Zusammenfassung Unix -System-API

0. Unix

- Anfänge von Unix 1965
- ab 1979/80 Entwicklung in 2 Strängen (BSD und AT&T)
- ist ein **Betriebssystem**
  = Gesamtheit aller Programme, die sich zu einem bestimmten Zeitpunkt in einem Rechnersystem befinden

- **Manuals**:
  1. Programme und Shellbefehle
  2. Systemrufe
  3. Bibliotheksrufe (C-Funktionen)
  4. spezielle Dateien
  5. Treiber
  6. Spiele
  7. Makropakete, Konventionen
  8. Systemadministration
  9. Kernelroutinen (nicht Standard)

- **Ordnerstruktur**:

/etc/	Files für Systemverwaltung
/home/	Homedirectories
/kernel/	Kernel-Module
/lib/	-> usr/lib/
/mnt/	Mountpunkt
/proc/	Prozessfilesystem
/usr/	Systemfiles für User
/usr/bin/	Binaries
/usr/lib/	Libraries und Konfigfiles
/usr/include/	Headerfiles
/var/	veränderbare Systemfiles

1. Prozesse

- Prozess = Programm, das ausgeführt wird (uasführbare Instanz eines Programmes)
  - Startzustand in einem Filesystem
  - hat Systemressourcen (Speicher, CPU...)

- besteht aus:
  - Textsegment: - Programmcode (auch von mehreren Programmen nutzbar => Shared Libraries)
    - schreibgeschützt
  - Datensegment: - Benutzerdaten eines Prozesses
  - Stacksegment: - alle lokale Variablen einer Funktion
  - Share-Memory-Segement:
    - für mehrere Prozesse geteilter Speicherbereich

- **als aktives Element:**
  - zur Beschreibung des Organisationsprinzips eines BS oder eines Algorithmus
  - Eigenschaften:
    - geboren, leben, sterben
    - sind in ihrer Zahl variabel
    - belegen Ressourcen und geben diese frei
    - beeinflussen sich
    - können zusammenarbeiten
    - können sich blockieren
    - können sich Ressourcen teilen
    - können voneinander un/abhängig sein
    - können hierarchisch voneinander abhängig sein

- **als passives Element:**
  - auf das Prozessor und Peripherie wirken
  - ist eine Datenstruktur:

Codesegment	Anfangszustand im File
Datensegment	-"-
Stacksegement	im Speicher oder geswapt
Eintrag in Proc-Liste	immer im Speicher
user-Struktur	im Speicher oder geswapt

- **proc-Struktur:**
  - ständig benötigte Prozessdaten
  - Identifier:

p_pid	PID
pp_pid	ParentPID
p_uid	UID
p_gid	GID
p_suid	effektiver UID (Wert mit dem tatsächlich versucht wird auf eine Ressource zuzugreifen)
p_sgid	effektiver GID
p_flags	Eigenschaften des Prozesses (im Hauptspeicher, ausgelagert...)
p_stat	Statusinfos des Prozesses (SSLEEP, SRUN, SZOMB...)
p_pri	aktuelle Priorität
p_nice	User nice-Priorität
p_cptr	1. Kind
p_pptr	Vaterprozess
p_ysptr	linker Bruder
p_osptr	rechter Bruder

- **user-Struktur:**
  - Ausführungsstatus (user-Modus/kernel-Modus, Register...)
  - Ressourcensteuerung

- **Statuswechsel eines Prozesses:**
  - im **user-mode:** kein Zugriff auf Systeminterne Daten (jederzeit unterbrechbar)
  - im **kernel-modus:** Zugriff auf Systeminterne Daten

- Prozess kann sein:
  - ready — wartet auf Prozessorzuteilung (im Speicher oder ausgelagert) (running oder runnable)
  - sleep — wartet auf Ereignis (im Speicher oder ausgelagert)
  - stopped — gestoppt oder unterbrochen
  - Zombie — beendet, aber noch nicht aus Tabelle entfernt

- **Prozessscheduling:**
  - Kriterien: Fairplay (jeder ist mal dran), Effektivität (CPU-Auslastung 100%), Antwortzeitverhalten (möglichst schnell), Durchsatz (möglichst viel), Verweilzeit (möglichst wenig Zeit, die der Batchjob braucht)

- **Round Robin Scheduling:**
  - basiert auf Prozessqueue FIFO
  - gleiches Zeitquantum für alle
  - wenn verbraucht, dann hinten anstellen
  - wenn blockiert, vorne bleiben, bei nächstem weiter, bis Blockierung aufgehoben
  - je größer Zeitquantum, desto grösser Reaktionszeit
  - je geringer Zeitquantum, desto geringer Effektivität (große Verwaltungszeit)

- **multiple Queues**
  - es ist sinnvoll CPU-intensive Prozesse länger im Speicher zu halten und größere Zeitquanten zu verteilen
  - mehrere Schlangen mit unterschiedlicher Priorität
  - unterschiedliche Zeitquanten für die Queues
  - nach Zeitverbrauch rutscht Job in nächste Priorität
  - Bevorzugung von kurzen Jobs
  - Swapvorgänge werden dadurch minimiert

- **Shortest Job First**
  - aus beobachteten Werten wird geschätzt, welcher Job am kürzesten ist
  - Einbeziehung der Vergangenheit (gewichtete Summen)

  => Bevorzugung kurzer Jobs

- **Zeit überwachte Steuerung**
  - Priorität = verbrauchte CPU-Zeit/(Login-Zeit/Anzahl Nutzer)

  => früh einloggen verschafft Vorteil

- **Zweistufiges Scheduling**
  - 2 Scheduler
  - einer für Prioritätssteuerung im HS (normales Scheduling mit kleinen Quanten)
  - einer fürs Swappen (große Quanten)
    - Wie lange ist der Prozess im HS?
    - Verbrauchte CPU-Zeit?
    - Größe des Prozesses?
    - Priorität des Prozesses?

Systemrufe zur Prozesssteuerung
- Prozessidentifikation:
    - jeder Prozess hat PID (0 = sched, 1 = init)
    - Header: sys/types.h unistd.h

    pid_t getpid/getppid (void) Nummer eigenen Prozess/Vater
    uid_t getuid/geteuid (void) real UID/effektive UID
    gid_t getgid/getegid (void) real GID/effektive GID

Erzeugung eines Prozesses
- **pid_t fork(void)**
- erzeugt neuen Prozess

- Kindprozess ist eine exakte Kopie (des Prozesses) der Umgebung seiner Vorfahren (auch geöffnete Dateidiskriptoren, Arbeitsverzeichnis, Signale, UID, EUID, GID, EGID, PGID, Nice-Value, S-Bit)

- copy on write Verfahren! erst kopieren, wenn Pages geändert werden
- hat eigene PID und PPID auf Vater
- beide Prozesse teilen sich gleichen Speicherbereich und führen gleiches Programm aus

- Unterschiede zw. Vater- und Sohnprozess:
    - Rückkehrcode child: 0
    - im Parentprozess: PID des Child
    - eingenes Stack-Segment und Datensegment
    - child:
        - diverse Lockaufhebungen

- Abwandlung davon fork1() kopiert nur aktuellen Thread (bei POSIX: fork=fork1)

Prozessbeendigung
- normales Prozessende: return im Hauptprogramm, void (_)exit(int Exitcode)
- abnormales Prozessende: void abort(void) => Signal SIGABRT oder Ende über Signalbehandlung

- Probleme bei Exitcode-Übergabe:
    - Parent wartet mit wait => kein Problem
    - Parent wartet nicht:
        - Parent existiert nicht:
            => Code an init übergeben
        - Parent existiert:
            => Code in Proc-Tabelle, alle anderen Infos gelöscht
            => Zombie (wird gestrichen, wenn Parent Resultat mit wait holt)

Warten auf Prozessende:
- beim Beenden wird Terminationscode gebildet (aus Exitcode oder Signalnummer, die stoppen bewirkt)

Systemrufe:

Header: sys/wait.h

pid_t wait(int *status)

- wartet auf erste Ende eines Kindprozesses
= wait3 + options + struct rusage *rusage

pid_t waitpid(pid_t pid, int *status, int options)

- wartet auf Ende eines bestimmten Kindprozesses
= wait4 + struct rusage *rusage

pid > 0 warte auf Ende von PID
pid == -1 warte auf irgendeins (= wait)
pid == 0 warte auf Ende von Child eigene Prozessgruppe
pid < -1 warte auf jeden Prozess mit PGID = |pid|

Options:

WCONTINUED	- warten bis Child beendet
WNOHANGUP	- kehrt sofort zurück (Elter nicht blockiert)
WUNTRACED	- auch wenn Child gestoppt

rusage = Ressourcenangaben über CPU-Zeit...

- wenn Status nicht ausgwertet werden soll: NULL

- WIFEXITED(status)	Child normal beendet
- WIEXITSTATUS(status)	+ Status des Kindes abfragen
- WIFSIGNALED(status)	Child durch Signal beendet
- WTERMSIG(status)	Nummer des Signals
- WIFSTOPPED(status)	Child angehalten
- WSTOPSIG(status)	Nummer des Signals

Rückkehrwert: >= 0 = Prozessnummer des Child
< 0 = Fehler

Programmausführung:

- **exec()** zum Laden und Starten eines neuen Programms in einem Prozess
  - überlagert Code- und Stacksegment
  - Stacksegment und private Signalbehandlungen werden zurückgesetzt
  - geöffnete Dateien und ignorierte Signale bleiben so
  - gleich bleibt auch: nice-Wert, PID, PPID, PGID...

- Rückgabewert: - keiner, wenn alles OK
<0 bei Fehlern

- **Versionen:**

int execl(const char *pathname, const char *arg0...,(char*)0)

- l für Liste (immer mit NULL beendet)
Bsp.: execl(Ordner, Prozessname, Parameter1...2, NULL)

int execv(const char *pathname, char *const argv[])
- v für Vektor-Paramter

int execve(const char *pathname, char *const argv[], char *const envp[])
- e für Enironmentvariablen als Vektor

int execlp(const char* filename...
- p für Dateinamen statt Pfad

int execvp(const char* filename...
wie execl/execv filename wird aber gesucht

Änderung von ID's in Prozessen

Systemrufe:

int setuid(uid_t uid) setzt UID für aktuellen Prozess

int setgid(gid_t gid) setzt GID für aktuellen Prozess

- wenn aktueller UID=0 (root), dann setuid UID auf uid und EUID auf uid (denn root kann alle ändern)
- wenn UID!=0 (nicht root) und (uid=UID oder uid=S(aved)UID, dann EUID auf uid (also wird nur die effektive UID gesetzt)

(gilt analog für setgid)

Rückgabewerte: 0 = OK <0 = Fehler

int seteuid(uid_t uid) setzt EUID für aktuellen Prozess

int setegid(gid_t gid) setzt GUID für aktuellen Prozess

- wenn aktueller UID=0, EUID = uid
- wenn aktueller UID!=0 und uid=UID oder uid=SUID, dann EUID=uid

(gilt analog für setegid)

int setreuid(uid_t uid, uid_t euid)

int setregid(gid_t gid, gid_t egid)

- root: setzt beide, andere: hin- und herschalten zwischen privilegiertem und User-Modus

pid_t setpgid(pid_t pid, pid_t pgid)
- ProzessgruppenID für PID auf pgid

pid_t setpgrp(void)
- aktuelle Prozess wird Prozessgruppenführer (PGID SID ergeben sich aus PID des Prozesses)
- Rückgabe: Nummer aktuelle Prozessgruppe

pid_t setsid(void)
- aktueller Prozess wird Prozessgruppenführer einer neuen Gruppe
- Rückgabe: Nummer aktuelle Sessiongroup

- **Änderung der Prioritäten**

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

- which gibt an, wie who interpretiert wird:
  - PRIO_PROCESS who ist PID
  - PRIO_PGRP who ist Prozessgruppe

- prio wird zu aktuellem nice-Wert addiert (negativ ist höhere Priorität)

2. Threads

- einzelner, sequentieller Steuerungsfluss in einem Programm (Leichtgewichtsprozess, weil Umschaltung zwischen Threads nicht soviel Aufwand bedeutet, wie bei Prozessen) (meisten Programme sind single Threads)
- neu: Multithreadunterstützung (ein Programm besteht aus mehreren Threads)
- Threads eines Prozesses teilen sich Heap, Code und Static-Data
- jeder Thread hat Stack-Register
- Threads eines Prozesses sind für Prozessor nur ein Prozess

Unterschiede Threads und Prozesse:

- jeder Prozess besteht aus einem Thread
- Aufwand für Interprozesskommunikation (Syncchhronisation) entfällt bei Threads fast völlig
- kein Aufwand zur Duplizierung des Namensraums, da Threads eines Prozesses den gleichen Adressraum nutzen
- wenn aber Thread abstürzt, reißt er alle anderen mit (weil kein Speicherschutz)

Arbeitsmodelle

- **Boss/Worker Modell**
  - ein Thread (Boss) übernimmt Steuerung und übergibt anderen Threads (Worker) Aufgaben
  - Worker melden Boss Arbeitszustand
  - Ableitung: Workqueue (dort holen sich Worker die Arbeit)

- **Work Crew Modell**
  - alle Threads arbeiten an einer Aufgabe

- **Pipelining Modell**
  - Fließbandarbeit
  - für jede Teilarbeit ein Thread (Eingaben eines Threads sind Ausgaben eines anderen)

Probleme mit Threads

- Threads haben Overhead, also gut überlegen, ob eingesetzt werden muss
- machen Quelltexte lesbarer

- Synchronisation durch Codelocking (Thread anhalten) oder Datalocking (Mutex oder Semaphoren)

- Deadlocks

- Nutzung von nicht reenteranter Software (kann nicht von mehreren Threads benutzt werden!)

- Faustregeln für Locks:
  - keine locks für I/O-Operationen
  - keine Locks beim Ruf von nichtreenteranten Funktionen
  - keine überzogenen Prozessoranforderungen während Locks
  - immer gleiche multiple Locks verwenden

Thread-Operationen
- Header: thread.h (pthread.h für POSIX-Threads)

- **Starten von Threads**

  int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void * arg);

  - neue Variable pthread_t thread;
  - Attribut ist normalerweise 0 (default)
  - neuer Thread, Abbarbeitung wird bei start_routine mit Parameter arg begonnen

- **Ändern der Eigenschaften von Threads**
  - möglicherweise andere Rechte benötigt!

  int pthread_attr_setdetachestate(pthread_attr_t *attribute, int detachstate);

  PTHREAD_CREATE_JOINABLE Exitstatus des Threads erfragbar

  PTHREAD_CREATE_DETATCHED beim Beenden sofort gestrichen (auch Rückkehrwerte nicht mit join holbar)

  int pthread_attr_setschedpolicy(pthread_attr_t *attribute, int policy);

  SCHED_OTHER nicht Echtzeit Scheduling (default)

  SCHED_RR Echtzeit (Round Robin)

  int phtread_attr_setinheritsched(pthread_attr_t *attribute, int inherit);

- **Beenden von Threads**

  void pthread_exit(void *value_ptr)

Exitcode

- aktueller Thread beendet
- wenn nicht Detached, dann speichern in ptr von Thread-ID und Exit-Status bis anderer Thread per join es holt

- **Warten auf Ende von Threads**

  int pthread_join(pthread_t thread, void **value_ptr);

  Zielthread Exitcode

  - aktueller Thread blockiert, bis angegebene Thread (muss im aktuellem Prozess sein und nicht Detached) beendet wird
  - Zusammenführung zum aktuellen Thread
  - wenn Rückgabe des abgeholten Threads nicht benötigt, dann auf NULL setzen

- **Holen der eigenen Thread-Identität**

  pthread_t pthread_self(void);

  - Rückgabe eigene Thread-ID

- **Initialisierung von Threads**

  int pthread_once(pthread_once_t once_control, void (*init_routine) (void));

  - init_routine wird nur einmal abgearbeitet von einem Thread

- **Manipulation der Threadpriorität**

  int pthread_getschedparam(pthread_t target_thread, int *policy, struct sched_param *param)

  int pthread_setschedparam(pthread_t target_thread, int policy const struct sched_param *param)

  - in *policy wird aktuelle Priorität gespeichert (get), bzw. aus policy gelesen (set)

- Synchronisation mit Mutexen

- Threads bezüglich Zugriff auf kritische Ressourcen synchronisiert
- = gegenseitiger Ausschluss (Mutex gehört einem oder keinem Thread)

=> Deadlockgefahr!

- Thread kann Ressourcen anfordern, obwohl er schon welche hat
- Thread gibt Ressourcen nicht frei

Systemrufe

- Header: pthread.h
- Mutex sollte als globale Variable für alle Threads sichtbar

sein

Mutex anlegen

pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutex_attr)
wenn Null, dann Standard

- legt neuen Mutex in mutex an, mit Attributen
- attr:
  - (meist) fast: Thread muss warten
  - error checking: return EDEADLK
  - recursive: mehrfach Lock möglich, aber genausoviele unlocks nötig, bis wieder frei

pthread_mutexattr_init(pthread_mutexattr_t *attr)
-setzt Attribute

pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
-entfernt Attribute

Mutex sperren

pthread_mutex_lock(pthread_mutes_t *mutex)
pthread_mutex_trylock(pthread_mutes_t *mutex)

- sperrt mutex, wenn frei
- wenn nicht frei, dann blockiert erste so lange, bis mutex frei, zweite kehrt mit Fehler EBUSY zurück, aber locked trotzdem

Mutex entsperren

pthread_mutex_unlock(pthread_mutex_t *mutex)

- Sperrung wird aufgehoben

Mutex löschen

pthread_mutex_destroy(pthread_mutex_t *mutex)

- löscht mutex

- bedingte Mutexobjekte
- erlauben Mehrfachzugriff auf ein Mutexobjekt
- Synchronisation erfolgt über Bedingungen, die an das Objekt gebunden sind

initialisiert/entfernen bedingtes Mutexobjekt

pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr)

- Attribute ist unter Linux ignoriert

pthread_cond_destroy(pthread_cond_t *cond)
- löscht die Condition

warten auf Condition

pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
- wartet beliebig lange auf Erfüllung der Condition

pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *exp)
- wartet nur bestimmte Zeit (exp, exp.tv_sec = Sekunden, exp.tv_nsec = Nanosekunden)

wieder aufwecken

pthread_cond_signal(pthread_cond_t *cond)
- weckt zufällig einen Thread auf

pthread_cond_broadcast(pthread_cond_t *cond)
- weckt alle Threads auf

- Ablauf:
  globale Variable
    pthread_cond_t cond und pthread_mutex_t mutex
  T1 macht was, was T2 braucht:
  T2 wartet an kritischer Stelle
    pthread_cond_wait(&cond, &mutex)
  T1 setzt lock
  T1 macht alles
  T1 pthread_cond_signal(&cond)
  T1 hebt lock auf
  T2 kann alles machen

3. Signalbehandlung

- für Kommunikation zwischen Prozessen
- sind asynchrone Ereignisse (Interruptanforderungen auf Prozessebene)
- für Prozesssynchronisation

- **Signalquellen:**
  1. Terminalsignale: Strg + Z ...
  2. Hardwareausnahmen: Division durch Null, Speicherfehler
  3. Kill-Systemrufe
  4. spezielle Softwarezustände: Pipeende, Nutzerzeit abgelaufen

- **Reaktionen:**
    - ignorieren (außer SIGKILL/SIGSTOP)
    - Nutzerspezifische Signalbehandlung
    - Default-Behandlung durch Kernel

- **einige Signale:**
    - SIGCHLD (Kindprozess beendet)
    - SIGCONT angehaltenen Prozess fortsetzen
    - SIGSTOP Prozess angehalten
    - SIGXCPU maximale CPU-Zeit überschritten
    - SIGUSR1/2 frei zur eigenen Benutzung

- früher wurde nach jedem Auftreten eines Signals die Routine auf SIG_DFL gesetzt (=> im Signalhandler wieder auf denselbigen setzen!)
- später blieb die Routine, bis man per signal() umgestellt hat
- heute wie früher

- Wirkung auf wartende Systeme:
    - früher: alle wartenden wurden unterbrochen, kein Restart des Systemrufs
    - heute: systemabhängig (unterbrochen, (default: kein) Restart (möglich, aber nicht nötig)

- atomare Operationen/Variablen nutzen, um Ausführung des Programms von Signalen zu lösen (Programm kann wichtige Sachen erledigen bevor es durch Signal beendet wird)

- Signalmaske enthält für jeden Prozess die Signale, die blockiert werden, also nicht zugestellt werden können

- **Systemrufe:**
    - Header: signal.h unistd.h

Initialisieren/Hinzufügen/Entfernen von Signalen aus Signalmenge
- beschreibt Zulassung und Behandlung von Signalen

int sigemptyset(sigset_t *sig_set);
- entfernt alle Signale auf die sig_set zeigt aus Signalmenge

int sigfillset(sigset_t *sig_set);
- fügt alle Signale auf die sig_set zeigt zur Signalmenge hinzu

int sigaddset(sigset_t *sig_set, int signal_nummer)

int sigdelset(sigset_t *sig_set, int signal_nummer)
- entfernen/hinzufügen von Signalnummer bezüglich sig_set

int sigismember(sigset_t *sig_set, int signal_nummer)
- testet, ob Signalnummer in sig_set enthalten

(1=ja,0=nein)

sigset_t sigsetmask(sigset_t mask)
- setzt aktuelle Signalmaske auf mask

sigset_t sigblock(sigset_t mask)
- fügt mask zu gesperrten Signale in aktueller Signalmaske hinzu (=add)

int sigprocmask(int how, sigset_t *set, sigset_t *oset)
- auslesen und setzen von aktueller Signalmaske
- how:

SIG_BLOCK	gesperrte Signale von *set zur aktuellen Signalmaske
SIG_UNBLOCK	gesperrte Signale von *set werden in aktueller Signalmaske als nicht gesperrt gesetzt
SIG_SETMASK	aktuelle Signalmaske = *set

- (set + oset) !=0 oset = set, dann set nach how setzen
- set!=0 + oset =0 set entsprechend how geschrieben
- set=0 + oset !=0 oset liefert aktuelle Signalmaske

int pthread_sigmask(int how, sonst sigset_t *set, sigset_t *oset)
- wie sigprocmask, aber für Threads

int sigpause(sigset_t sigmask)
- aktuelle Signalmaske auf sigmask
- wartet auf Signal
- ursrpüngliche Signalmaske nach Signal wiederhergestellt

int sigsuspend(sigset_t sigmask)
- wie sigpause

int sigpending(sigset_t* set)
- alle hängenden Signale (die in set aufnehmen

int sighold(int sig)
- fügt sig zur aktuellen Signalmaske hinzu
- Signal in Zukunft gesperrt

int sigrelse(int sig)
- entfernt sig aus aktuellen Signalmaske
- kann in Zukunft wieder auftreten

int sigignore(int sig)
- sig für aktuellen Prozess auf SIG_IGN

int sigpause(int sig)
- entfernt sig aus aktuellen Signalmaske und wie pause

Funktionen um Signalroutinen aufzusetzen

struct sigaction:

sighandler_t sa_handler	Adresse der Funktion, die Signal behandeln soll (auch konstante Werte: SIG_DFL, SIG_IGN...)
sigset_t sa_mask	Signale die blockiert werden, während Signalhandler ausgeführt wird (Signale, die abgefangen werden)
sa_flags	Signaloptionen (SA_RESTART für Restart von unterbrochenem Systemcall nach Signal, SA_RESETHAND = auf SIG_DFL setzen))

alt **void (*signal(int signo,void(*func)(int)))(int)**
- func wird bei Signal signo aktiviert (int Parameter für Signalnummer)

= **int sigvec(int signo, struct sigvec *vec, struct sigvec *ovec)**

struct sigvec:

void (*ssv_handler)	= sighandler_t sa_handler
int sv_mask	= sigset_t sa_mask
int sv_flags	= sa_flags (SV_INTERRUPT = SA_RESTART, SV_RESETHAND = SA_RESETHAND)

= **void (*sigset(int sig, void(*disp)(int)))(int)**
- wie signal, aber wenn sig auftritt, wird es in Signalmaske des aktuellen Prozesses aufgenommen => nochmals sig unterdrückt

neu **int sigaction(int signr, const struct sigaction *handler_neu, struct sigaction *handler_alt)**
wenn nicht gewünscht NULL (gibt sonst alten zurück)

Signale senden

int kill(pid_t pid, int signo)

- sendet Signal signo an PID
  - (>0 an PID
  - ==0 an eigene Prozessgruppe
  - == -1 Nutzer: an alle Nutzerprozesse
    SU: an alle Prozesse ausser 0 und 1
  - <-1 an alle Prozesse aus Prozessgruppe |pid| == killpg(int pgrp, int sig)
- signo == 0 => checken, ob PID gültig ist und Signal

gesendet werden kann
- gleichen Rechte wie Zielprozess nötig

int pthread_kill(pthread_t thread, int sig)
- sendet Signal an einen bestimmten Thread

int raise(int signo)
- sendet signo an eigenen Prozess
= kill(getpid(), signo)

- Rückkehrwerte: 0 = OK, -1 = Fehler

unsigned int alarm(unsigned int seconds)
- Kern sendet nach seconds SIGALRM an aktuellen Prozess
- überschreibt eventuellen alten Alarmwert

int pause(void)
- stoppt aktuellen Prozess, bis Signal eintrifft

unsignet int sleep(unsigned int seconds)
- Rückgabe der eventuell nicht geschlafenen Sekunden

void usleep(unsignet long microseconds)

4. Prozesskommunikation (IPC)

- Messages, Semaphoren, Shared Memory(System V spezifische)
- Sockets (BSD spezifisch)
- es gibt Speicherschutzmechanismen (Prozesse wissen nichts über andere Prozesse), wird von Viren unterlaufen
- Gründe für Kommunikation:
  - mehrere Prozesse müssen Daten gemeinsam nutzen
  - Prozesse sind voneinander abhängig und müssen aufeinander warten
  - Daten von einem zum anderen Prozess reichen
  - Koordinierung von Systemressourcen

- Header: sys/ipc.h

Ressourcenidentifikation
- extern über long int Schlüssel (muss allen Prozessen bekannte sein)
- intern numerische ID (erhält man beim Eröffnen)

Pipe
- garantiert auf jedem System
- verbindet Ausgabe eines Kommandos mit Eingabe eines anderen (unidirektional)
- Shell kümmert sich um Synchronisation (erst lesen, wenn geschrieben wurde)
- **unbenannte Pipe: |**
  - direkt über Pufferbereich des Kernels
  - nur mit verwandten Prozessen (über fork)

    - gut zur Flusskontrolle, da nur
- benannte Pipes: FIFO Dateien   (mittels mkfifo Name)
    - mit völlig fremden Prozessen

Pipe einrichten   (unistd.h)

int pipe(int fd[2])
- eine Pipe mit fd[0] als Filediscriptor zum Lesen und fd[1] zum Schreiben

- dann fork(), denn Kind erbt Pipe
- nun muss noch klargemacht werden, wer schreiben bzw. lesen darf
- Schließen von fds
- lesen und schreiben mit read(fd, puffer, bytes) und write(fd, puffer, bytes), blockieren solange, bis etwas in der Pipe steht oder wieder Platz zum schreiben ist, wenn voll

Messages
- Austausch von Nachrichten zwischen Prozessen
- Messegequeue im Kern
- Nachrichten bestehen aus Text und Nachrichtentyp

```
struct msgbf{
    long mtype;
    char mtext[1];
}
```

- FIFO-Prinzip
- Daten bleiben selbst nach Ende des Erzeugers noch erhalten

- Header: sys/types.h   sys/ipc.h   sys/msg.h

Messegequeue erstellen

int msgget(key_t key, int msgflg)
- gibt msqid für Messegequeue key zurück
- msgflg sind Zugriffsrechte
    IPC_PRIVATE   öffnen Privaten Schlüssel
    IPC_CREATE   erstellen, wenn nicht vorhanden
    IPC_EXCL   + CREATE, dann Fehler, wenn vorhanden

Rückgabe: ID neue Messegequeue

Nachricht senden

int msgsend(int msgid, struct msgbuf *msgp, int msgsz, int msgflg)

- sendet Message an msgid (mit msgget erfragen)
- msgsz gibt länge von mtext in msgbuf (msgp) an
- wenn msgflg IPC_NOWAIT, dann macht Prozess gleich weiter und wartet nicht, bis Empfängerprozess die Nachricht abgeholt hat oder mit msgctl() die Messegequeue

entfernt wurde

Rückgabe: msgsz, wenn erfolgreich <0 Fehler

Nachrichten empfangen

int msgrcv(int msgid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)

- liest Messeage in msgp
- msgsz ist maximale Länge von mtext
- msgtyp sagt, welcher Typ geholt werden soll
  - ANY alle
  - >0 1. Message mit mtype = msgtyp
  - <0 1. Message mit mtype <= |msgtyp|

- wenn msgflg MSG_NOERROR, dann können auch Nachrichten länger als msgsz geholt werden

Rückgabe: Anzahl Zeichen in mtext, wenn erfolgreich
<0Fehler

Messegequeue steuern

int msgctl(int msgid, int cmd, struct msgid_ds *buf)

- Steuerfunktionen
- cmd = gewünschte Steuerfunktion
  - IPC_RMID entfernt Messegequeue aus System
  - IPC_STAT liest Struktur msgid_ds in *buf (Status)
  - IPC_SET setzt Werte aus msgid_ds im Kern (Zugriffsrechte, Größe...)

struct msgid_ds verschiedene Angaben über Status der MQ

Semaphoren

- Synchronisation der Arbeit von verschiedenen Prozessen
- Feld von Semaphoren im Kern
- eine Variable, wenn >0 dann frei, sonst gesperrt

- Header: sys/types.h sys/ipc.h sys/sem.h

Erzeugen

int semget(key_t key, int n_sems, int flag)

- n_sems = Anzahl Semaphore im Satz (wenn 0, dann bereits existierende Semaphore öffnen)
- flag wie bei MQ + PERM (muss gesetzt werden, damit man auch Zugriff darauf bekommt)

Abfrage, ändern, löschen

```
int semctl(int sem_id, int semnum, int cmd, union semun arg)
```

- cmd:

IPC_RMID	entfernt Messegequeue aus System
IPC_STAT	liest Struktur semid_ds in *buf (in arg) (Status)
IPC_SET	setzt Werte aus semid_ds im Kern (Zugriffsrechte, Größe...)
GETVAL	abfragen Semaphorenvariable
SETVAL	setzt Semaphorenvariable
GETPID	PID des zuletzt zugreifenden Prozesses
GETNCNT	#Prozesse, die warten, bis VAL > 0
GETZCNT	#Prozesse, die warten, bis VAL = 0

Ändern Semaphorenvariable

`int semop(int semid, struct sembuf *buf[], size_t n_op)`

- buf ist Array von Operationen
- n_op Anzahl Argumente in buf
- sembuf:

unsigned_short semnum	(Semaphorennummer)
short semop	(Semaphorenoperation) wenn 1, dann Ressource für alle anderen Prozesse freigegeben, -1 gesperrt, wenn 0, dann setzt Prozess seine Arbeit fort
short semflg	(IPC_NOWAIT, SEM_UNDO (setzt Semaphor zurück, was von beendetem Prozess gesetzt wurde) ...)

Shared Memory

- Prozesse greifen auf gleiche Daten zu
- Shared Memory Segment
- schnellstte Möglichkeit der Kommunikation, da nix kopiert werden muss

- Header: sys/types.h sys/ipc.h sys/shm.h

Erzeugen

`int shmget(key_t key, int size, int flag)`

- size ist Mindestgröße (wenn vorhandes öffnen, dann 0)
- flag wie bei MQ

Abfragen, Ändern, Löschen

int shmctl(key_t key, int cmd, struct shmid_ds *buf)

- cmd:

IPC_STAT	Status abfragen (liest Struktur shmid_ds in *buf)
IPC_SET	setzt Werte aus shmid_ds im Kern (Zugriffsrechte, Größe...)
IPC_RMID	entfernt SHM aus System
SHM_(UN)LOCK	(ent)sperren des SHM (Superuser)

Anbinden (attach)

void *shmat(int shm_id, const void *adresse, int flag)
(NULL für Automatik)

- flag:

SHM_RDONLY	nur für Lesen

Lösen (detatch)

int shmdt(void *adresse)

- löst SHM vom Prozess (wird nicht gelöscht)

Sockets
- für lokale und globale Kommunikation über Protokolle
- TCP - sichere Übertragung (verbindungsorientiert) (Stream)
- UDP - unsicher, aber schneller als TCP (verbindungslos) (Datagramm)
- global IP-Adresse, Einteilung in Klassen A, B, C, D
- lokal Portadresse + Protokoll

- Header: sys/types.h sys/socket.h

- TCP/IP:

Server:	socket()	bind()	listen()	accept() read/write()	close()
Client:	socket()			connect() read/write()	close()

-UDP/IP:

Server:	socket()	bind()	recfrom()/sendto()	close()
Client:	socket()	bind()	sendto()/recfrom()	close()

Anlegen

int socket(int family, int type, int protocol)

- family:

AF_INET	Internetadressen
AF_UNIX	interne Unixadressen

- type:

SOCK_STREAM	stream-socket (TCP)
SOCK_DGRAM	datagram-socket (UDP)

SOCK_RAW raw-devices

- protocol:
  IPPROTO_UDP
  IPPROTO_TCP
  0 das zu type passende Protokoll wird gewählt

Rückgabe: Filedescriptor(SockID)=OK <0 Fehler

Verbindungsaufbau (Client)

int connect(int sockfd, struct sockaddr *adresse, socklen_t adressen_laenge)

- structs:

sockaddr

u_short	sa_family	Adressfamilie
char	sa_data[128]	Adresse

in_addr

u_long	s_addr	32-bi-HostID

sockaddr_in

short	sin_family	Adressfamilie
u_short	sin_port	16-bit-Portnummer
struct	in_addr sin_addr	
char	sin_zero[8]	unused

Socket mit Port verknüpfen (Server)

int bind(int sockfd, struct sockaddr *adresse, socklen_t adressen_laenge)

- bei Clients nicht nötig, da dann automatisch ne Ausgangsadresse genommen wird
- bei Servern Muss!

Auf Verbindung warten (Server)

int listen(int sockfd, int warteschlange)

- belauscht sockfd
- warteschlange = maximale Verbindungszahl (es kann nur eine Verbindungsanfrage abgearbeitet werden, der Rest wird gecached und danach abgearbeitet)

int accept(int sockfd, struct sockadr *adresse, socklen_t adressen_laenge)

- erzeugt neuen Filedescriptor, über den Datenübertragung laufen kann, alte wird für neue accepts genutzt
- erfolgreiches accept führt bei Client zu erfolgreichem connect()

Senden und empfangen

ssize_t read(int sockfd, const void *data, size_t data_len)
ssize_t write(int sockfd, const void *data, size_t data_len)

- Puffergröße beachten!

ssize_t send(int sockfd, const void *data, size_t data_len,
unsigned int flags)
meist 0

- sendet data an sockfd
- testen, ob wirklich schon alles gesendet wurde!!!
- vorher connect, wenn für UDP genutzt

int sendto(int sockfd, const void *data, size_t data_len,
unsigned int flags, struct sockaddr *to, int addrlen)

- wie send, aber für UDP
- mit Adressangaben

ssize_t sendmsg(int sockfd, struct msghdr *buf, size_t
data_len, unsigned int flags)

- egal ob TCP oder UDP

ssize_t recv(int sockfd, const void *data, size_t data_len,
unsigned int flags)
meist 0

- liest daten von sockfd in data

int recvfrom(int sockfd, const void *data, size_t data_len,
unsigned int flags, struct sockaddr *to, int addrlen)

- wie rcv, aber für UDP

ssize_t recvmsg(int sockfd, struct msghdr *buf, size_t
data_len, unsigned int flags)

- egal ob TCP oder UDP

msghdr:

```
caddr_t   msg_name
int       msg_len
struct iovec *msg_iov
u_int     msg_iovlen
caddr_t   msg_control
int       msg_controllen
int       msg_flags
```

flags:

MSG_OOB	out-of-band Daten empfangen/senden
MSG_PEEK	Daten besichtigen, ohne aus Datenstrom zu entfernen (rcv, rcvfrom)

Verbindung schließen

int close(int sockfd)

Optionen

int getsockopt(int sockfd, int level, int optionname, char *option_value, int *option_len)

int setsockopt(int sockfd, int level, int optionname, char *option_value, int *option_len)

- setzen oder lesen von Optionen für sockfd
- level = Protokoll
- option_name:

SO_SNDBUF	Puffergröße send
SO_DEBUG	
SO_KEEPALIVE	Connection aktiv bleiben
SO_TYPE	Sockettype (get)

int getsockname(int sockfd, struct sockaddr *adresse, int *addr_len)

- holt zu Socket sockfd gehörende lokale Adresse

int getpeername(int sockfd, struct sockaddr *adresse, int *addr_len)

- holt zu Socket sockfd gehörende remote Adresse

Hilfsfunktionen

Header: netinet/in.h inttypes.h arpa/inet.h

int inet_aton(const char *ptr, struct in_addr *inp)

- String in punktierte Dezimaldarstellung
- = inet_addr() (veraltet)

char *inet_ntoa(struct in_addr *in)

- Gegenteil von inet_aton
- network-byte-order
- nicht Thread-Safe (wegen statischen Puffer des Strings)

in_addr_t inet_network(const char *ptr)

- macht aus String ne Netzwerknummer (Netzanteil einer IP-Nummer N.N.N.H)

in_addr_t inet_netof(struct in_addr in)

- macht aus numerischen 32-Bit eine Netzwerknummer
- Rückgabe als host-byte-order

in_addr_t inet_lnaof(struct in_addr in)

- extrahiert lokalen Teil aus numerischen 32-Bit-Adresse
- Rückgabe als host-byte-order

struct in_addr inet_makeaddr(int net, int host)
- net + host => numerische 32-bit Netzwerknummer
- Übergaben als host-byte-order

struct hostent *gethostbyname(const char *name)
- Header: netdb.h
- macht aus namen (www.google.de) eine IP-Adresse
- hostent enthält Namen, Aliase, Adresstyp etc.

struct hostent *gethostbyaddr(const char *ip_addr, int len, int type)
- Gegenteil von gethostbyname

htonl long Integer von Host- nach Netzdarstellung
htons unsigned short int von Host- nach Netzdarstellung
ntohl long int von Netz- nach Hostdarstellung
ntohs unsigne short in von Netz- nach Hostdarstellung

5. E/A-Geräte unter Unix

- Hardware

E/A-Geräte:
- blockorientierte Geräte (Speicher...)
  - feste physische Blocklänge
  - direkter oder sequentieller Zugriff
  - Seekoperationen möglich
- zeichenorientierte Geräte (Terminal, Drucker, Modem)
  - Ströme von Zeichen
  - sequentieller Zugriff
  - keine feste Satzstruktur
  - keine Seekoperation
- Sondergeräte (Uhren, hauptspeicher)
  - nicht für direkten Datenaustausch

Controller:
- haben Standardschnittstellen
- realisieren Verbindung zwischen herstellerspezifischen Busstrukturen und den standardisierten Geräteschnittstellen
- wandeln Bit- in Byteströme
- Geräteauswahl
- Fehlerkorrektur
- Interrupterzeugung
- Ausführung von Steueroperationen

- Software

Grundanforderungen
- Geräteunabhängig
- einheitliche Namensgebung von Geräten und Dateien
- einheitliche Fehlerbehandlung
- (a)synchrone E/A-Operationen

- Synchronisation und Organisation des Zugriffs auf E/A-Geräte
- Struktur: Interrupthandler -> Gerätetreiber -> geräteunabhängige Systemsoftware -> nutzerorientierte Software

Interrupthandler

1. rettet aktuellen Prozessorzustand
2. Memorymanagement (blendet aktuellen Prozess aus und Gerätetreiber ein)
3. aktiviert Gerätetreiber
4. aktiviert einen Prozess nach Interruptbehandlung

Gerätetreiber

- C-Routine zum Steuern einer Gerätefamilie
- geräteunabhängige Funktionen: xx_open, xx_close, xx_read, xx_write, xx_ioctl (Steueroperationen), xx_intr (Interruptroutine), xx_strategy (lesen/schreiben blockieren)

- es gibt low-Level (open, close...) und high-Level (fopen, fclose...) Rufe

Systemrufe

- Header: sys/types.h sys/stat.h fcntl.h

int open(const char *pfadname, int flags)

int open(const char *pfadname, int flags, mode_t permission)

- wenn pfadname neu eröffnet wird, dann setzen der Zugriffsrechte von permission
  S_I[RWX][USR, OTH, GRP] [user, other, group]-[read, write, execute]

- flags:
  O_RDONLY
  O_WRONLY
  O_RDWR
  O_APPEND
  O_CREAT
  O_EXCL erzeugt Fehler, wenn File ex. und O_CREAT

Rückgabe: Filedescriptor=OK -1=Fehler

int close(int fd)

- schließt Filedescriptor fd
- gibt alle Ressourcen frei

ssize_t read(int fd, void *buf, size_t nbytes)

- Rückgabe = Anzahl gelesener Zeichen != nbytes, wenn EOF/lesen aus Socket/Terminal oder Blockgröße vom E/A-Gerät anders

ssize_t write(int fd, void *buf, size_t nbytes)

- Rückgabe = Anzahl geschriebener Zeichen

in ioctl(int fd, int operation, ...)
- Header: sys/ioctl.h
- Steueroperation operation über File fd

6. Filesystem

Allgemeines:
- Möglichkeiten der Datenspeicherung
  1. Bildung von **Segmenten**
     Startprozess verwaltet Segemente
     Programme haben feste Segmentnummern
     manche Segmente als Directories (Verweise auf andere Segemente)
     Daten werden in Segementen gespeichert
     +: Daten als Teil des Adressraumes => schnell
     -: unpraktisch, statisch, störanfällig
  2. **Files** = benannte Objekte (enthalten Daten, Programme...)
     gehören nicht zum Adressraum des Prozesses
     +: dynamisch

Fileorganisation
- Bytefolge (UNIX), Blockfolge(CP/M), Baumstruktur
- möglichst geräteunabhängig (Anwenderprogramme soltlen sich nicht darum kümmern wo und wie File gespeichert wird)

Directories
- enthalten mehrere Files
- pro File ein Record im D
- = Files

Filespeicherung
1. als zusammenhängende Folge von Bytes => problematische Speicherzuweisung
2. als Folge von Blöcken (nicht notwendigerweise zusammenhängend)
   Probleme: Größe der Blöcke (Zugriffszeit vs. Plattennutzung)
   Optimum 512 Platte, 4096HS

Freispeicherverwaltung
1. verkettete Liste (enthalten Blocknummern der freien Blöcke)
   je weniger freie Blöcke, desto weniger Platz, schnell
2. Bitmapping (pro Block 1 Bit 0=belegt, 1=frei)
   es wird viel Platz benötigt und langsam, aber sicher und einfach zu verwalten

Organisation Filespeicherung
Problem: File = Blöcke => Speicherung der Reihenfolge der Blöcke?

1. verkettete Liste
   - Blocklänge 1024 (Datenlänge 1022 + 2 Byte Pointer zum nächsten Block)

- Nachteil: Blocklänge keine 2er Potenz
  Direktzugriff schwer
  Probleme bei E/A-Fehler

2. FAT – File Allocation Table (DOS)
   - Pro Block ein Pointer zum nächsten in einer Tabelle
   - Nachteil: für ein File immer ganze FAT nötig (bei großen Medien viel HS nötig)

3. i-node (UNIX)
   - enthält Dateinummer, Besitzer, Zugriffsrechte, Zugriffszeiten, Typ und Größe der Datei, Blöcke, die die Datei ausmachen

Filesystemwiederherstellung

1.	Badblocks	finden und sperren
2.	dynamisches Backup	Spiegeln von Platten (aufwendig/teuer, sicher!)
3.	statisches Backup	zu festen Zeiten auf Backupmedium
4.	Filesystemkonsistenz	doppelt freie Blöcke, doppelt benutzte Blöcke finden

Systemrufe

- Header: sys/types.h sys/stat.h fcntl.h

int creat(const char *pathname, mode_t mode)
- = open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode)
- existierende Files werden vorher gelöscht
- Rückgabe ist Fd

int dup(int fd)
- verdoppelt fd
- Rückgabe ist neuer Fd (nächste freie FD)

int dup2(int fd, int fd2)
- verdoppelt fd, Ziel fd2 (wenn voerher geöffnet => close)
- nicht in POSIX

off_t lseek(int fd, off_t offset, int whence)
- positioniert Zugriffspunkt in fd an durch whence und offset spezifizierte Stelle
- physisches Positionieren erst bei nächstem read/write
- whence

SEEK_SET	offset = Anzhal Bytes vom Fileanfang
SEEK_CUR	offset = Anzahl Bytes von aktueller Position
SEEK_END	offset = Anzahl Bytes vom Fileende

int pipe(int fd[2])
- öffnet 2 neue FD (lesen[0], schreiben[1])
- wenn lesende Prozess beendet, SIGPIP an schreibenden

Prozess
- EOF erst, wenn schreibende close()

int fcntl(int fd, int cmd, long arg)
- Ausführung von Steuer- und Abfrageoperationen über fd
- cmd = Kommando, arg = Informationen für cmd
  - F_DUPPD newfd(oldfd, F_DUPPD, newfd)
  - F_GETFL RKW: Zugriffsrechte
  - F_SETPL Zugriffsarten wie bei open

int stat(const char *pathname, struct stat *buf)
- Infos über pathname

int fstat(int fd, struct stat *buf)
- infos über eröffnete File fd

int lstat(const char *pathname, struct stat *buf)
- wie stat, aber symbolische Links nicht verfolgt

stat enthält Infos über i-node, UID, GID, letzter Zugriff...

int access(const char *pathname, int mode)
- testet, ob mode-Zugriffsrechte für pathname vorhanden
- mode:
  - R_OK Test auf Lesen
  - W_OK Schreiben
  - X_OK Ausführen
  - F_OK Existenz

mode_t umask(mode_t cmask)
- setzt Filecreationmaske
- es werden Bits gesetzt, wofür später kein Zugriff erlaubt sein soll
- cmast wie bei open

int chmod(const char *pathname, mode_t mode)
int fchmod(int fd, mode_t mode)
- setzen Zugriffsrechte aus mode für fd/pathname
- mode wie bei umask + s-Bits
  - S_ISUID set-user-ID on execution
  - S_ISGID set-group-ID on execution
  - S_ISVTX saved-text

int chown(const char *pathname, uid_t owner, gid_t group)
int fchown(int fd, uid_t owner, gid_t group)
int lchown(const char *pathname, uid_t owner, gid_t group)
- ändern von UID und GID
- wenn pathname = symbolischer Link, dann mit lchown nur dessen UID und GID geändert
- nur vom Eigentümer oder SU

int truncate(const char *pathname, off_t length)
int ftruncate(int fd, off_t length)

- Länge eine Files festlegen (0en anhängen)

int link(sonst char *altpath, const char *neupath)
- für altpath neuer Directoryeintrag neupath

int unlink(const char *pathname)
- löscht Link
- wenn letzte, werden auch Daten gelöscht

int rmdir(const char *pathname)
- löscht Directory (muss leer sein)

int rename(const char *altname, const char *neuname)

int readlink(const char *pathname, char *buff, int buffsize)
- liest symbolischen Link in buff
- open, read, close

int utime(const char *pathname, const struct utimebuf *time)
- setzt Zugriffs- und Modifikationszeit

int mknod(const char *pathname, int mode, int dev)
int mkfifo(const char *pathname, int mode)
- erzeugt Specialfiles (nur SU, USR nur fifo oder normal)
- mode:

S_IFDIR	directory
S_IFCHR	charakter-special
S_IFBLK	block special
S_IFREG	regular
S_IFIFO	fifo

int mkdir(const char *pathname, mode_t mode)
- neues Verzeichnis mit mode-Zugriffsrechten (umask wird beachtet!)

int chdir(const char *pathname)
int fchdir(int fd)
- setzt aktuelle Arbeitsverzeichnis für Dateien, die nicht mit / anfangen

int chroot(const char *pathname)
int fchroot(int fd)
- SU ändert root-verzeichnis für Prozess

int mount(char *type, char *dir, int flags, caddr_t data)
- Eingliedern eines blockorientierten Gerätes
- dir ist Mount-Point
- data = Gerät
- type = filesystem des Gerätes
- flags = M_RDONLY...

int umount(char *dir)

- gliedert blockorientiertes Gerät aus

int sync(void)
- synchronisiert Filesystem, alles was noch nicht geschrieben wurde, kommt auf Platte

int fsync(int fd)
- synchronisiert alle Blöcke von fd

int flock(int fd, int operation)
- schützt eröffnetes File fd vor Zugriff
- Operation
  - LOCK_SH shared Lock (mehrer Prozesse teilen sich lock lesend)
  - LOCK_EX exclusive Lock (kein weiterer Lock möglich)
  - LOCK_NB nicht blockieren, wenn Lock
  - LOCK_UN unlock

int select(int maxfd, fd_set *readfd, fd_set writefd, fd_set exceptfd, struct timeval *tvptr)
- fragt mehrere fd ab, ob Daten angekommen sind (ob gelesen, geschrieben werden kann oder ob aussergewöhnliches Ereigniss auftrat)
- vor jedem Aufruf FD_ZERO(fd_set *fdset) (löscht Menge)
- FD_SET(int fd, fd_set *fdset) fügt fd zur Menge hinzu
- FD_CLR(int fd, fd_set *fdset) löscht fd aus Menge
- FD_ISSET(int fd, fd_set *fdset) testet, ob fd noch in Menge
- mit tvptr kann festgelegt werden, wie lange gewartet werden soll

caddr_t mmap(caddr_t addr, size_t len, int prot, int flag, int fd, off_t off)
- blendet Teil des Adressraums eines Files fd in Adressraum des Prozesses
- Gegenteil ist int munmap(caddr_t addr, int len)

7. Hauptspeicherverwaltung

Monoprogrammierung:
- einfachste Art
- nur ein Prozess kann laufen (bei einfachen Mikrocomputern)
- feste Stelle im Speicher
- meist kein Speicherschutz!

Multiprogrammierung:
- bei größeren Speichern zur besseren Nutzung der CPU-Zeit
- HS in Partitionen aufgeteilt
- Tasks je nach Speicher und CPU-Bedarf in Klassen (die Patitionen zugeordnet werden)

1. **feste Partitionen:**

1. gleichgroße: + gut für Verwaltung
   - schlecht für HS-Auslastung
2. unterschiedliche:
   mit Eingabewarteschlange für alle Partitionen
   + gute HS-Auslastung, kleine Tasks können überall laufen
   - Hoher Verwaltungsaufwand
   mit Eingabewarteschlange für jede Partition
   + einfache Verwaltung
   - schlechte HS-Auslastung

Probleme: Speicherschutz und Verschieblichkeit der Tasks

2. **variable Partitionen:**
   - Anzahl je nach Bedarf
   - Swapping (Prozess wenn größer auslagern und wieder einlagern)
   -

Probleme: wieviel Anfangsspeicher?, Vergrößerung der Partition (vorher etwas mehr einplanen? oder von Begin an Maximum?)

3. **Möglichkeiten der Speicherverwaltung**
   Bitmaps:
   - Speicher in kleine gleichgroße Teile zerlegt (pro Einheit ein Eintrag in Tabelle)
   - je länger Bitmap, desto länger Suchzeiten

   Listen:
   - Speicher in kleine gleichgroße Teile zerlegt
   - Einheiten zu Gruppen, wenn sie zusammenhängend im Speicher liegen
   - pro Gruppe ein Eintrag (Kennzeichen, Anfang, Länge, Verweis zum Prozess (Nutzer))
   - Problem: finden eines Passenden Stückes

   Buddy-Systeme:
   - Speicher entsprechend 2er-Potenzen aufteilen
   - daraus passende Stücke wählen
4. **Swappingalgorithmen**
   - Wann? Wohin?

Virtueller Speicher
- Prozess bekommt virtuellen Speicher (kann kleiner oder größer als realer Speicher sein)
- immer benötigten Adressbereiche im HS durch Algorithmus

Paging
- HS in Frames, kann genau eine Seite aufnehmen
- virtuell wird über Seitennummer adressiert
- soft- oder hardwareseitig
- nur benötigte Seiten in HS

Segmentation

- ähnlich Paging
- mehrere Seiten logisch zu Segment

Probleme: Replacementalgorithmus (Wann kann Seite ersetzt werden? -> Verweildauer, Nutzungshäufigkeit, Einfachheit der Auslagerung (Größe))

- FIFO
- LRU

Systemrufe

- Header: unistd.h sys/mman.h sys/types.h

implizite Speicherverwaltung

fork, exec, exit mmap, munmap (Speicherbereich in Prozess ein und ausblenden)

explizite Speicherverwaltung

int brk(void *addr)
- setzt Ende des Datensegmentes von Prozess auf addr gesetzt
- wird auf nächste vielfache Seite gerundet
- addr muss größer als aktuelle Ende sein

Rückgabe: 0=OK -1=Fehler (kein Speicher...)

size_t gepagesize(void)
- Größe eine Seite in Byte

int mprotect(const void *addr, size_t len, int prot)
- setzt Zugriffsrechte von Adressbereich durch mmap eingebunden

int mctl(caddr_t addr, size_t len, int function, void *arg)
- SU
- Steueroperationen über die Seite im Speicher

MC_LOCKAS	festhalten aller Seiten im HS
MC_UNLOCKAS	freigeben zum Swappen
MC_LOCK	Speicherbereich wird im HS fixiert
MC_UNLOCK	Fixierung aufheben
MC_SYNC	Synchronisation der Seiten mit Hintergrundspeicher

int plock(int operation)
- Header: sys/lock.h
- SU kann Auslagerung des Daten/Textsegmentes verbieten
- operation:

PROLOCK	Prozesslock (lock Text- und Datensegement)
TXTLOCK	Textsegment
DATLOCK	Datensegment
UNLOCK	hebt lock auf