Systemprogrammierung

Autor: Julian Kotzur

Grundlagen der C-Programmierung

Main-Funktion

→ Compilieren: gcc -o hello hello.c
→ Ausführen: ./hello

```
int main(int argc, char *argv[]){    }
```

Datentypen

→ Standard: char, short, int, long, long long
→ Kommazahlen: float, double, long double
→ Leerer Datentyp: void
→ Boolean: Muss selbst definiert werden
→ Vorangestellte Modifizierer:
  ⇒vorzeichenbehaftet: signed
  ⇒vorzeichenlos: unsigned
  ⇒konstant: const

Variablen

→ Globale Variablen:
  ⇒Definition außerhalb der Funktion
  ⇒Syntax: extern Datentyp name;
→ Lokale Variablen:
  ⇒innerhalb eines Blockes definiert
→ Lebensdauer von Variablen:
  ⇒static: Lebt über ganze Ausführzeit des Programms
  ⇒automatic: Lebt über die Ausführzeit des Blockes

```
int main(int argc, char *argv[]){
 int x = 420;
 printf("Der Wert von x ist: %i \n" , x);
}
```

Structs

→ Zusammenfassung mehrerer Daten zu einer Einheit
→ Auch als Array möglich

```
struct person { int Wert1; int Wert2; };
 int main(int argc, char *argv[]){
// --- Inizialisierung ---
 struct person p1;
 p1.alter = 21
 struct person p2 = { 42 , 1337};
}
```

Blöcke

→ Zusammenfassung mehrerer Anweisungen
→ Lokale Variablendefinition (Namen-neubelegung)

```
int main(int argc, char * argv[]){
 int x = 10; printf("x = %i\n" , x);
// x = 10 ------------------------
  { int x = 20;
// x = 20 ------------------------
    {    x = 30;
// x = 30 ------------------------
    } // x = 30 ------------------------
  } // x = 10 ------------------------
}
```

Kontrollstrukturen

→ Bedingte Anweisungen (if, else)
→ Einfache Verzweigungen (goto)
→ Fallunterscheidung (switch, case)
→ Schleifen (while , do-while, for)
→ Zusatzanweisungen:
  ⇒Schleifenabbruch: break;
  ⇒Nächste Iteration: Continue

```
int c = 0;
do {
 if (c == 2){ continue;}
 else if (c == 3) {goto label;}
 else{break;}
 c = c + 1;
} while ( c < 20);
label:
```

Funktionen

→ In Java Methoden genannt
→ Funktionen sind generell global definiert
→ main() normale Funktion, wird als erstes Aufgerufen
→ Rekursive Aufrufe zulässig
→ Funktionen müssen vor Verwendung deklariert werden!
  ⇒Möglich durch Kurzdeklaration vor der Main-Funktion
  ⇒Syntax: Typ Name (Parameterliste);
→ Parameterübergabe:
  ⇒Variablen: Call by Value
  ⇒Pointer: Call by Reference
  ⇒void-Pointer müssen deklariert werden, void nicht
→ static:
  ⇒Sichtbarkeitseinschränkung
  ⇒Funktionsaufruf nur innerhalb der Datei möglich
  ⇒Aus Gründen robuster Programmierung wichtig

```
// Funktionsdefinition:
<Rueckgabe> <Name> (<Parameter>){ ... }
// Fibonacci:
int fib (int n){
 if (n == 1) return (1);
 if (n == 2) return (2);
 else return( fib(n-1) + fib(n-2) );
}
```

Makros und Einfügen

→ Am Anfang des Dokumentes
  ⇒Syntax: # define Makroname Ersatztext

→ Einfügen von Datei Inhalt in C-Quellcode
  ⇒# include <Dateiname> bzw. # include "Dateiname"

Explizite Typumwandlung

→ Meist regelt das C selbst und automatisch
→ Teilweise explizite Typumwandlung nötig:
  ⇒Besonders bei Pointer wichtig
  ⇒Syntax: <Name> = <Typ> <Name2>

Programm richtig beenden

→ Syntax: exit(< int-Wert >)
→ EXIT_SUCCESS : Programm endet richtig
→ EXIT_FAILURE : Fehler im Programm

Pointer

Grundlagen:
→ Pointer: Enhält Adresse einer Variable
→ NULL-Pointer sollte Default-Pointer sein
→ call-by-reference zugriff
→ Variablendefinition:
⇒Syntax: Typ *Name;
→ Adressoperator &
⇒liefert Referenz auf den Inhalt einer Variable
→ Verweisoperator *
⇒ermöglicht Variablenzugriff (Dereferenzierung)
⇒!Unterschied! zu Variablendefinition
→ Void Pointer:
⇒Zur Übergabe an Funktionen sehr Praktisch
⇒Wertezugriff führt zu Fehler
⇒Mit Typumwandlung wieder nutzbar
Zeiger und Funktionen:

```
void *addadress(int *pointer);
int main(){
 int array[] = {1 , 2 , 3};
 int *a = place;
 a = addadress(a);
}
void *addaddress (int *pointer){
// --- Siehe Pointer Arithmetik ---
 return (int*) (((char*)pointer)+1);
}
```

Zeiger auf Strukturen:

```
struct person{int alter;};
struct person student;
struct person *pointer;
pointer = &student;
// --- Schwer Leserlich ---
(*pointer).alter = 21;
// --- Besser Leserlich ---
pointer -> alter = 42;
```

Pointer auf Pointer:

```
int x = 5;
int *ip = &x;
int **ipp = &ip;
```

Werte verändern:

```
int x = 5;
int *pointer = &x;
// --- x wird auf 10 gesetzt ---
*pointer = 10;
```

Pointer Arithmetik

→ Pointer kann man rechnerisch bearbeiten
→ Pointerdefinition-abhängige Rechnung:
⇒*char sind "kleinste" Pointer
⇒*char++ springt im Speicher symbolisch 1 weiter
⇒*int++ springt im Speicher symbolisch 4 weiter
→ Typumwandlung bei Pointer
⇒Problem: Int Pointer um einen Char verschieben
⇒Lösung: (int*) (((char*)pointer)+1);

```
int array[3]; int *ip = array;
// Zugriff auf naechste Adresse:
ip++;
// Zugriff auf naechsten Speicherblock:
ip+1;
```

Arrays

Arrays ohne Pointer

→ Damit wird Speicher auf dem Stack belegt

```
// Erstellen:
int prim[4] = { 2 , 3 , 5 , 7 };
// Zugriff:
printf("%i\n" , prim[0] );
// Mehrdimensional:
int feld[5][7]; feld[2][3] = 10;
```

Arrays und Pointer

Pointer auf ein Array:

```
int prim[4] = { 2 , 3 , 5 , 7 };
int *ptr = prim;     // pointer 1tes Element
int *ptr = &prim[2];// pointer 3tes Element
```

Pointer Array:

```
int* a[5];
```

Mehrdimensionale Arrays mit Pointern erstellen:
→ Damit wird Speicher auf dem Heap belegt

```
int ***array,a , b;
array=(int ***) malloc(256 *sizeof(int **));
for(int i=0;i<256;i++){
array[i]=(int **)malloc(256 *sizeof(int *));
for(int j=0;j<256;j++)
array[i][j]=(int *)malloc(256 *sizeof(int));
}
```

Arrays als Parameter:
→ Arrays werden nicht call by value übergeben
→ Arrays werden mit Pointern an Funktionen übergeben

Strings

→ Char Array mit '\ 0 ' am Ende
→ C füllt das Char Array Automatisch mit '\0' auf
→ Feldname ist Pointer auf ersten Buchstaben
→ Stringlänge Funktion:
⇒strlen(< String >);
⇒Abspeicherung meist in Ganzzahl-Datentyp
→ String Vergleich:
⇒strcmp und strncmp

```
// --- String erstellen ---
char str[50];
strcpy(str, "Cogito␣Ergo␣Sum");

// --- Laenge bestimmen: ---
int strlen(const char string[]){
 int i=0;
 while (string[i] != '\0') ++i;
 return(i);
}

// --- String kopieren: ---
void strcpy(char *to, const char *from){
 while ( *to++ = *from++ );
}

// --- String Array ---
char *sarray[2] = {"foo" , "Zuege"};
```

Parallelisierung

Kindprozesse starten:

→ Rückgabe:
⇒ElternProzess wird Kind-Prozess-ID zurückgegeben
→ Am Ende immer ein exit(0);
⇒Außer nach exec(): dann ein exit(1)
→ Waitpid:
⇒pid_t waitpid(pid_t pid, int* stat, int options)
⇒Wartet auf Beendigung von Kinderprozessen
⇒Zerstört bei Ausführung den Kinderprozess
⇒Kinderprozess sendet SIGCHLD nachdem terminieren
⇒SIGCHLD abfangen und waitpid ist sehr effizient

```
// --- Benoetigt ---
#include <unistd.h>
// --- Prozesserzeugung ---
pid_t pid = fork();
if( pid < 0 ) {
// --- Error ---
} else if( pid == 0 ) {
// --- child ---
...
exit(0);
} else {
// --- Parent ---
...
}
```

Zombies mit Signalbehandlung behandeln:

```
static void ZombieHandler(){
int tmp = errno;
pid_t pid;
while((pid=waitpid(0,&event,WNOHANG))>0){
// pid kann verwendet werden
}
errno = tmp;
}
// beliebigen Kindprozess behandeln:
while(waitpid(-1, NULL, WNOHANG) < 0)
```

Auf Terminierung eines Kindes warten:

```
while(counter == <maxKinder>){
 sigsuspend(&<maske>);
}
// mit leerer Maske moeglich
```

Threads starten:

Threads:
→ Benötigt: pthread.h
→ Threads Beenden mit return NULL;
→ Threads arbeiten auf Run-Methode

```
// --- Notwendige Funktion: ---
static <...> run(<...>);

// --- Ein Thread: ---
pthread_t thrd;
errno = pthread_create(&thrd,NULL,run,NULL);
if(errno != 0) die();
pthread_join(thrd,NULL);// wartet bis fertig

// --- Max. n Threads ---
int nThreads = 42;
semNThreads = semcreate(nThreads);
pthread thd;
while(1){
 P(semNThreads);
 errno = pthread_create(&thd,NULL,run,NULL);
 if(errno != 0) die();
}

// --- N Threads starten: ---
pthread_t tid;
errno = 0;
for(int i = 0; i<maxThreads; i++){
 errno = pthread_create(&thd,NULL,run,NULL);
 if(errno) die();
}
```

Semaphoren:

Semaphoren:
→ Locks für paralleles Arbeiten
→ P() dekrementiert Zähler und blockiert ggf. Aufrufer
→ V() inkrementiert Zähler und weckt ggf. Threads
→ Anzahl Locks:

```
// --- Semaphoren initialisieren ---
<semName> = semCreate(<Anzahl Locks>);
if(<semName> == NULL) die();
// --- Daten mit Semaphoren locken ---
P(<semName>); // Lockt
Datenaufruf: <name>++;
V(<semName>); // Unlockt
// --- Am Ende Semaphoren schliessen ---
semDestroy(<semName>);
```

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

// Creates a new semaphore.
SEM *semCreate(int initVal) {
 SEM *sem = malloc(sizeof(*sem));
 if (sem != NULL) {
  if(pthread_mutex_init(&sem->m,NULL)==0){
   if(pthread_cond_init(&sem->c,NULL)==0){
    sem->value = initVal;
    return sem;
   }
   pthread_mutex_destroy(&sem->m);
  }
  free(sem);
 }
return NULL;
}

// Destroys a semaphore.
void semDestroy(SEM *sem) {
 if (sem == NULL) return;
 pthread_mutex_destroy(&sem->m);
 pthread_cond_destroy(&sem->c);
 free(sem);
}

// P-operation.
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);
}

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

Ringpuffer

```
static volatile int read, write;
static SEM *full, empty, readL;
static int BB[<size>];

void bbCreate(){
 read = 0;
 write = 0;
 full = semCreate(0);
 empty = semCreate(<size>);
 readL = semCreate(1);
}

void bbDestroy(){
 semDestroy(readL);
 semDestroy(empty);
 semDestroy(full);
}

void bbPut(int value) {
 P(empty);
 BB[write] = value;
 write = (write + 1) %<size>;
 V(full);
}


int bbGet(){
 P(readL);
 P(full);
 int result = BB[read];
 read = (read +1)% <size>;
 V(empty);
 V(readL);
 return result;
}
```

ABA-Problem

→ Sei T1 = Thread 1 und T2 = Thread 2
→ 1. T1 liest eine Variable var mit dem Wert A.
→ 2. T1 bekommt CPU entzogen und T2 läuft.
→ 3. T2 ändert var von A nach B nach A.
→ 4. T1 bekommt CPU und kontrolliert die Variable A.
⇒T1 hat von der Änderung nichts mitbekommen

Dynamische Speicherverwaltung

→ Datenlebensdauert Segmentabhängig:
⇒Stack: Lebensdauer für den Block
⇒Daten: Lebensdauer für immer
⇒Heap: dynamische Lebensdauer mittels malloc

MALLOC

→ Benötigt: # include <stdlib.h>
→ Malloc setzt den Pointer neu
⇒Ab dem Pointer angeforderter Speicherplatz
⇒Falls Probleme: Malloc setzt NULL-Pointer
→ Immer NULL-Abfrage, ansonsten schlechtes Programm

```
int *feld;
feld=(int*) malloc(42 * sizeof(int));
if(feld==NULL){
 perror("malloc failed"); exit(1);
}
```

CALLOC

→ Benötigt: # include <stdlib.h>
→ Macht im Endeffekt das gleiche wie Malloc
⇒Unterschied: der komplette Speicher wird genullt

```
int *feld;
feld = (int*) calloc(42 , sizeof(int));
if(feld==NULL){
 perror("calloc failed"); exit(1);
}
```

REALLOC

→ Benötigt: # include <stdlib.h>
→ Vergroessert den mit Malloc/Calloc allokierten Speicher
⇒Returnt NULL-Pointer bei Fehler

```
int *feld;
feld = (int*) calloc(42 , sizeof(int));
feld = (int*) realloc(feld, 73 * sizeof(int));
// NULL-Abfrage nicht vergessen!
```

FREE

→ Benötigt: # include <stdlib.h>
→ Deallociert einen allocierten Speicherbereich
→ Werte stehen weiterhin in dem Speicher
→ Werte sind nicht mehr "gesichert" und Pointerlos

```
int *feld;
feld = (int*) calloc(42 , sizeof(int));
free(feld);
```

Signalbehandlung

Gängige Signale:

SIGINT	Beenden durchs Terminal (CTRL-C)
SIGKILL	Tötet Prozess, nicht behandelbar
SIGPIPE	Schreiben trotz geschlossener Gegenseite
SIGTERM	Standardsignal von kill(1)
SIGABRT	Signal nach Aufruf von abort
SIGFPE	Rechenfehler, z.b. teilen durch 0
SIGSEGV	Speicherzugrifffehler
SIGCHILD	Statusänderung eines Kinprozesses (Zombies)

Signalbehandlung Allgemein

Signalmaske:
Beinhaltet Signale, die in den Wartezustand versetzt werden, während eine Signalbehandlung läuft. (Meist wg. Vermeidung von Paralellitätsproblemen).
Es ist jeweils nur eine Maske gleichzeitig pro Prozess aktiv, d.h. jeder Prozess hat eigene Signalbehandlung, Kinderprozesse Signalbehandlung der Eltern und Masken können ersetzt werden, indem man eine neue aktiviert.

```
// sigaction-Aufbau: handler, flags, mask
struct sigaction sig;

// Signal Ignorieren:
sig.sa_handler = SIG_IGN;
// Standart Behandlung:
sig.sa_handler = SIG_DFL;
// Eigene Funktion aufrufen:
sig.sa_handler = <Funktionsname>;

// Optionen fuer Kinder und Systemaufrufe:
sa_flags = ... ;

// Signalmaske von sig aktivieren!
sigemptyset(&sig.sa_mask);

// Signalbehandlung starten:
if(sigaction(<Signal>, &sig, NULL)==-1)die();
// Weitere Signalmaske erstellen:
sigset_t mask;
// Signalmaske initialisieren:
if(sigemptyset(&mask)==-1) die();
// Signal zur Maske hinzufuegen:
if(sigaddset(&mask, <Signal>)== -1) die();
// Signalmaske aktivieren:
// Statt Null kann Maske eingetragen werden
// Darin wird dann alte Maske gespeichert
if(sigprocmask(SIG_BLOCK,&mask,NULL) == -1){
die();}
// Signalmaske aendern:
sigprocmask(SIG_SETMASK, &mask, NULL);

// Passives Warten auf Signal:
// Alle nicht in der Maske enthalten Signale
sigsuspend(&mask);

// Beispiel: Zombies ignorieren:
struct sigaction action;

action.sa_handler = SIG_DFL;
action.sa_flags = SA_NOCLDWAIT;

sigemptyset(&action.sa_mask);
sigaction(SIGCHLD, &action, NULL);
```

Client-Server: Clientseitig

Grundlagen:
→ IP-Adresse (Identifikation des Rechners im Web)
→ Port-Nummer (Identifiziert Dienst auf dem Rechner)
→ Unterschiedliche Byteorder bei differenten Systemen
⇒BigEndian: 11110000
⇒LittleEndian: 00001111

Kommunikationsarten:
→ Verbindungsorientiert (TCP):
⇒Gesichert gegen Verlust und Duplizieren von Daten
→ Paketorientiert (UDP):
⇒Schutz vor Bitfehler, kein Schutz vor Paketverlust

Name des Clients:

```
char fqdn[sysconf(_SC_HOST_NAME_MAX) + 1];
if(gethostname(fqdn, sizeof(fqdn))==-1){
 return(EXIT_FAILURE);
}
struct addrinfo hints = {
 // Adressauswahl einschraenken
 .ai_flags = AI_CANONNAME;
};
struct addrinfo *result;
int er=getaddrinfo(fqdn,NULL,&hints,&result);
if(er!=0) return(EXIT_FAILURE);
strncpy(fqdn, result->ai_canonname, size);
fqdn[size-1] = '\0';
freeaddrinfo(result);
```

Mit Host verbinden:

```
struct addrinfo hints = {
 // Adressauswahl einschraenken
};
struct addrinfo *result;
int er=getaddrinfo(host,port,&hints,&result);
if(er!=0) return(EXIT_FAILURE);
struct addrinfo *tmp;
int sock;
for(tmp=result;tmp!=NULL;info->ai_next){
 int family = tmp->ai_family;
 int socktype = tmp->ai_socktype;
 int protocol = tmp->ai_protocol
 sock=socket(family,socktype,protocol);
 if(sock == -1) continue;
 socklen_t addrlen = tmp ->ai_addrlen;
 er=connect(sock,tmp->ai_addr,addrlen);
 if (er == 0) break;
 close(sock);
}
FILE *stream = NULL;
if(tmp =NULL){ exit(EXIT_FAILURE);
}else{
 stream=fdopen(sock,"a+");
 if(stream==NULL) exit(EXIT_FAILURE);
}
freeaddrinfo(result);
// --- Kommunikation mit Host ---
fprintf(stream, "%s", <string>);
// --- Verbindung beenden ---
if(fclose(stream) = EOF){
 exit(EXIT_FAILURE);
}
```

Client-Server: Serverseitig

→ File* stream am Ende immer schließen!

```
// Create Listen Socket
int socket = socket(PF_INET6,SOCK_STREAM,0);
if(sock == -1) die();

// Fuer IPv4 die 6 weglassen
struct sockaddr_in6 adress;
memset(&address, 0 , sizeof(address));
address.sin6_family = AF_INET6;
address.sin6_port = htons(port);
address.sin6_addr = in6addr_any;

if(bind(socket,(const struct sockaddr *)
  &address, sizeof(address)) != 0) die();

if(listen(socket, SOMAXCONN) != 0) die();

// Accept and handle incoming connections
int clientSock = accept(socket, NULL, NULL);
if(clientSock == -1) die();

// Lesestream mit Client:
FILE* read = fdopen(clientSock, "r");
if(!read)die();
char tmp = '.';
if(!fgets(tmp, sizeof(char) , read));
fclose(read);
close(clientSock);

// Schreibstream mit Client:
int clientSock2 = dup(clientSock);
if(clientSock2 == -1) die();
File* write = fdopen(clientSock2, "w");
fprintf(write , "Nachricht\n");
fclose(write)
close(clientSock2);
```

Datei mit HTTP senden:

```
static void sendFile(FILE *client,
const char path[], const char  relPath[]) {

 FILE *file = fopen(path, "r");
 if (file == NULL) {
  httpNotFound(client, path, relPath);
  return;
 }
 httpOK(client);
 int c;
 while ((c = getc(file)) != EOF) {
  if (putc(c, client) == EOF) break;
 }
 fclose(file);
}
```

Dateisystem:

File erschaffen und beschreiben:

```
File* f;
if((f=fopen(<DateiName>,'w'))==NULL) die();
// Mit Eingabe von Client beschreiben:
char tmp = '.';
while(tmp != EOF){
 fgets(tmp,1,<ClientFileStream>);
 fprintf(f, %c, tmp);
}
close(f);
```

Jede Datei im Ordner besuchen:

```
// Zu Ausgangsordner gelangen:
Dir* d=opendir(<Pfad>);
if(!d) die();
// Ausgangsordner durchsuchen:
struct dirent *e;
while(errno=0, (e=readdir(d))!=NULL){
// Dateiname:
 e->d_name[<index>];
}
```

Verzeichnis ändern:

```
static void changeCwd(char * argv[]){
 if(argv[1] == NULL || argv[2] != NULL) die();
 if(chdir(argv[1]) != 0) die();
}
```

Umleiten von STD-IN oder STD-OUT:
→ open() und dup2()

```
// Umleiten von STD-OUT
int outFile =open(<Pfad>,<Rechte>|<Rechte>);
if(outFile==-1)die();
if(dup2(outFile,STDOUT_FILENO)==-1){
 close(outFile); die();
}
// Umleiten von STD-IN
int inFile = open(<Pfad>, <Rechte>);
if(inFile==-1)die();
if(dup2(inFile,STDOUT_FILENO)==-1){
 close(outFile); die();
}
```

Print Current Directory

```
void printcwd(){
// PATH_MAX benoetigt limits.h
 errno = 0;
 char cwd[PATH_MAX];
 getcwd(cwd , sizeof(cwd));
 if(cwd != NULL && (errno != ERANGE)){
  fprintf(stdout, "%s:" , cwd);
 }else{
  perror("Error");
  exit(EXIT_FAILURE);
 }
}
```

Öffnen und Schließen von Dateien

→ Funktion fopen benötigt *#include* <stdio.h>
→ Syntax: FILE *fopen(char *name, char *mode);
⇒name: Pfadname ausgewählter Datei
⇒mode = r: Lesen
⇒mode = w: Schreiben
⇒mode = a: Am Dateiende schreiben
⇒mode =rw: Lesen und Schreiben

Rekursive Verzeichnis Durchsuchung

```
void work(const char* path,int depth){
DIR *currentDir = opendir(path);
if(currentDir == NULL) die());

struct dirent *de;
while (1) {
 errno = 0;
 de = readdir(current_dir);
 if(de == NULL) break;
 struct stat s;
 int tmp=strlen(path)+1+strlen(de->d_name);
 char newPath[tmp];
 if (lstat(newPath, &s) != 0) die();
// --- Output und Rekursion ---
 if(maxdepth == -1 || maxdepth > depth){
  if(S_ISDIR(s.st_mode)){
   work(newPath, depth + 1);
  } else if (S_ISLNK(s.st_mode)) {
   ausgeben(path, de->d_name, s);
  } else if(S_ISREG(s.st_mode)){
   ausgeben(path, de->d_name, s);
  }
 }
}
if (closedir(current_dir) != 0)die();
}
```

Nützliche Funktionen:

opendir:
→ DIR *opendir(const char *name);
→ Öffnet ein Verzeichnis für den Prozess
→ Rückgabe: Pointer auf das Verzeichnis
→ Schließen mit: closedir(DIR *ptr);

readdir:
→ struct dirent *readdir(DIR *dirp);
→ DIR dirp repräsentiert Verzeichnisstream aus opendir
→ readdir liefert nächsten Eintrag
→ Gibt am Ende NULL zurück
→ struct dirent enthält Dateiinformationen siehe manpage

struct stat:
→ Initialisierung mit lstat:
⇒int lstat(char * path, struct stat *buf);
→ speichert Informationen über Datei: siehe manpage

Ein- und Ausgabe:

Argumente aus der Commando-Zeile

→ Wird der Main-Methode mitgegeben
→ argc enthält Anzahl der mitgegebenen Argumente
→ argv ist pointer auf die Argumente
→ Das erste Argument liegt bei argv[1]

```
main (int argc, char *argv[]){
// Anzahl der Argumente:
int anzahl = argc;
// Auf einzelnes Commando zugreifen:
int size = sizeof(argv[1]);
char tmp[size+1];
strcpy(tmp , argv[1]);
// Alle Commandos ausgeben:
int i;
for (i=1; i<argc; i++) {
 printf("%s\n", argv[i]);
}}
```

Pfad von der Kommandozeile lesen

→ Benötigt: sys/stat.h
→ struct stat speicher Pfad/Datei-Informationen
→ lstat inizialisiert den struct

```
char *pfade[n];
char *arg = getPath(); // von argv holen
struct stat s // speichert Pfadinformationen
errno = 0;
if(lstat(arg , &s) == -1){
 perror(); exit(EXIT_FAILURE);
}
if(!S_ISDIR(s.st_mode)) exit(EXIT_FAILURE);
// Test ob Pfad abgeschlossen
pfad[i] = arg;
```

Eingabe lesen:

```
// Von STDIN lesen:
long max = sysconf(_SC_LINE_MAX) + 2;
char *tmp =(char*)calloc(max, sizeof(char));
if(tmp==NULL) die();
fgets(tmp , max, stdin);
if(tmp==NULL) die();

// Lesen von Stream: siehe Server
```

Ausgabe schreiben:

Ausgabe:
→ int printf(char *format, /* Parameter */ ...);
  ⇒Standartausgabe
→ int fprintf(FILE *fp, char *format, /* Parameter */ ...);
  ⇒Dateikanal fp
→ int fputc(int c, FILE *stream);
  ⇒Schreibt den char c in den Stream
→ int fputs(const char *s, FILE *stream);
  ⇒Schreibt den String s in den Stream

Kanäle:

→ Jedes Programm hat 3 Standartkanäle:
  ⇒stdin: Standarteingabe
  ⇒stdout: Standartausgabe
  ⇒stdeer: Standart-Fehler-Ausgabe
→ Zusätzlich kann man Streams selbst erschaffen:
  ⇒File *name;
  ⇒Anwendung1: Siehe Client-Server
  ⇒Anwendung2: Dateisystem

Formatierung:

→ % d = int / % hd = short / % ld = long int
→ % lld = long long int / % f = float / % lf = double
→ % Lf = long double / % c = char / % s = string
→ nach einem % kann eine Zahl folgen
  ⇒gibt an, wie lange int, o.ä., ist

FGET

→ Benötigt: stdio.h
→ Syntax: char *fgets(char *str, int n, FILE *stream)
  ⇒Pointer auf Array zum speicher
  ⇒n ist Anzahl der zu lesenden Chars
  ⇒stream Pointer auf ein File objekt
  ⇒stream kann auch stdin sein
  ⇒Als Flag: True solange nicht EOF, sonst false
→ Syntax: int fgetc(File* stream);
  ⇒liest genau einen char
  ⇒Liest am Ende EOF und speichert dieses

FEOF

→ Benötigt stdio.h
→ Syntax: int feof(FILE *stream);
  ⇒Pointer auf File oder stdin
  ⇒0 wenn noch Daten vorliegen
  ⇒1 wenn End of File
→ Wird in einer Schleife verwendet
  ⇒Abbruch, wenn End of File

```
while(feof(filestream) == 0){}
```

strcmp

→ Benoetigt: string.h
→ Syntax: int strcmp(const char *s1, const char *s2);
  ⇒0 : s1 == s2
  ⇒-1: s1 vor s2
  ⇒1 : s1 nach s2
→ strncmp vergleicht die ersten n Zeichen
  ⇒int strncmp(const char *s1, const char *s2 , size_t n)

strlen

→ size_t strlen(const char *s);
→ Liefert Größe eines Strings

strcpy

→ char *strcpy(char *dest, const char *src);
→ char *strncpy(char *dest, const char *src, size_t n);
→ Kopiert von src nach dest
→ dest sollte gleich groß oder größer sein, sonst BOOOM

strchr

→ char *strchr(const char *s, int c);
→ Rückgabe: Pointer auf den ersten Eintrag der c ist

Nützliche Standart-Funktionen

Size-Of-Operator

→ Liefert die Größe eines Objektes in Bytes
→ benötigt: # include <stddef.h>

QSORT

→ Benoetigt stdlib.h
→ Syntax: qsort(*base , n , g , compare);
⇒*base: Pointer zum Anfang des Arrays
⇒n: Anzahl der Elemente des Arrays
⇒g: Groesse des Arrays
⇒compare: Vergleichsfunktion
⇒compare: a == b: 0
⇒compare: a>b: 1
⇒compare: a<b: -1;
⇒Tipp: STRCMP ist compare Funktion

```
int compare(const void *a , const void *b){
 return ( *(int*)a - *(int*)b );
}
// _______
int zahlen[] = {5 , 7 , 3 , 13 , 9};
int *base = zahlen;
qsort(base , 5 , 5*sizeof(int) , compare);
```

ABORT

→ Beendet ein Programm vollstaendig

```
abort();
```

SYSCONF

→ Benötigt: unistd.h
→ Syntax: long sysconf(int name);
→ Kann Überprüfen ob Optionen unterstützt werden
→ Kann Konfiguration-Werte des System ausgeben

EXECVP

→ Wird in einem Kindprozess gestartet
→ Benötigt: unistd.h
→ Syntax: execvp(char *a, *argv);
⇒a sollte ein Command sein. Bsp: "ls"
⇒argv sollte ein Array mit Argumenten sein

MEMCPY

→ Benötigt: # include <string.h>
→ Copy Paste fuer Speicherbereich
→ Deklaration: void *memcpy(void *p1, const void *p2, n)
⇒Speicher wird von p2 kopiert und bei p1 eingefuegt

```
dconst char src[6] = "Hallo";
dchar dest[6] = "abcde";
dmemcpy(dest , src , 5);
```

MEMSET

→ Benötigt: # include <string.h>
→ Ersetzung der ersten n Buchstaben eines Arrays
→ Ersetzt wird es jeweils mit dem gleichen Char → Deklaration: void *memset(void *p, int c, n)

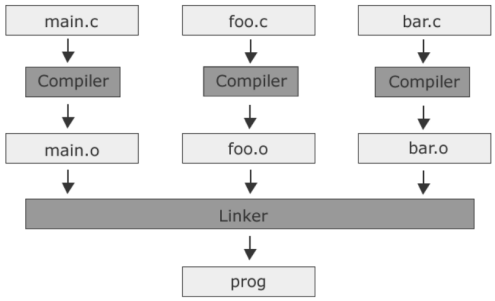
```
char p[6] = "Hallo";
memset(p , 'a' , 5);
```

EXEC

→ Startet einen Prozess
→ int execl(const char *path, const char *arg, ...);
→ int execlp(const char *file, const char *arg, ...);
→ int execv(const char *path, char *const argv[]);
→ int execvp(const char *file, char *const argv[]);

Zusaetzliche Kleinigkeiten

Make-File




```
.PHONY: all clean

all: prog clean

prog: main.o foo.o bar.o
gcc -o prog main.o foo.o bar.o

main.o: main.c
gcc -c main.c

foo.o: foo.c
gcc -c foo.c

bar.o: bar.c
gcc -c bar.c

clean:
rm -f prog main.o foo.o bar.o
```

→ all:
⇒Vorher meist .Phony: all
⇒Nach ":" alle ausführbaren gelinkten Programme
⇒Wird konventionell eingefügt
→ PHONY und Clean konventionell hinzufügen

Variablen:
→ Syntax: <Varname> = <Operation>
→ Verwendung: $ (Varname)
→ Konventionelle Variablen:
⇒ Compiler: CC = cc
⇒ CompilerOptionen: -std=c99 -pedantic
-D_XOPEN_SOURCE=600 -Wall -Werror -m32 -O3
⇒ LinkerOptinen: LDFLAGS = -m32
⇒ Löschen: RM = rm -f

Listen

```
// --- Listen-Struktur: ---
struct list{int wert; struct list *next};
static struct list *hauptliste = NULL;
// --- Element einfuegen: ---
int insertElement(int value){
 struct list *helper = hauptliste;
 if(value < 0) return -1;
 while(1){
  if(helper == NULL){
  helper = malloc(sizeof(struct list));
  if(helper == NULL) return -1;
  helper->wert = value;
  helper->next = NULL;
  return value;
 }else{
  if(helper->wert == value) return -1;
  helper = helper->next;
}}}
// --- Element loeschen: ---
int removeElement(void){
 if(hauptliste == NULL) return -1;
 struct list *merker = hauptliste;
 int tmp = hauptliste->wert;
 hauptliste = hauptliste->next;
 free(merker);
 return tmp;
}
```

Fehlerbehandlung

1. Variante:
→ Ausgabe auf der Error-Pipe
→ fprintf(stderr, "Text", Werte);

2. Variante:
→ Benötigt: *#include* <errno.h>
→ Syntax: void perror(const char *s);

ERRNO

→ Benötigt: errno.h
→ ist als globaler Integer definiert
→ Wenn ungleich 0, gab es einen Fehler
⇒Wichtig für einlesen: Errno erkennt ungültige Zeichen
⇒Daher: errno != 0 nach fgets

Rechnerorganisation

Übersetzung:

1. Schritt: Präprozessor
→ Kommentare werden entfernt
→ Bearbeitung von Include und Macros
2. Schritt: Compilieren
→ C wird in Assembler übersetzt
3. Schritt: Assemblieren
→ Maschienencode wird erstellt
4. Schritt: Binden
→ Ausführbare Datei wird erstellt
→ statisch und dynamisch möglich (betrifft Bib-Funktionen)

Grundbegriffe:

→ Programm: Folge von Anweisungen
→ Prozess: In Ausführung befindliches Programm
⇒Kann mehrere Programme ausführen
→ Kompilierer: Wandelt Quellsprache in Zielsprache um
→ Interpreter: Führt Programm direkt aus

Laden eines Programms:

→ Statistisch gebundene Programme:
⇒Von Ebene 5 auf Ebene 3
⇒Läd beim Compilieren alle Komponenten in den Speicher
⇒Alle Adressen werden zum Bindezeitpunkt aufgelöst
→ Dynamisch gebundene Programme:
⇒Von Ebene 5 auf Ebene 1
⇒Läd zur Laufzeit benötigte Komponenten
⇒Adressen werden beim Programmstart aufgelöst
⇒Auflösung von Adressbezüge beim Übersetzen möglich

Mehrebenenmaschine:

→ 5. Problemorientierte Programmierebene
→ 4. Assembliersprache
→ 3. Maschinenprogrammebene
→ 2. Befehlssatzebene
→ 1. Mikroarchitekturebene
→ 0. digitale Logikebene
→ Hinweise:
⇒Schichten 3-5 repräsentieren virtuelle Maschine
⇒Schichten 0-2 repräsentieren reale Maschine
⇒Schicht 4 Logisch existent aber unwichtig geworden
→ Durchlauf:
⇒5 → 4: Kompilation
⇒4 → 3: Assemblieren und Bindung
⇒3 → 2: Partielle Interpretation
⇒2 → 1: Interpretation

Programmhierachie:

→ Maschinenprogramme enthalten zwei Befehlsformen:
⇒Maschinenbefehle der Befehlssatzebene
⇒Systemaufrufe an das Betriebssystem
→ Maschinenprogramme (MaschProg):
⇒ohne Compiler vom Prozessor ausführbar
⇒Meist von Compiler generiert
⇒Grundlage bilden Hochsprachen und Assembler
→ Triumvirat:
⇒MaschProg: Anwendungsprogramm + Laufzeitsystem
⇒MaschProg: Benutzerebene
⇒Betriebssystem
⇒Zentraleinheit
⇒Ausführplattform: Betriebssystem + Zentraleinheit
⇒Ausführplattform: Systemebene

Betriebssystem

→ Menge von Programmen der Befehlssatzebene
→ Stellt ein nicht sequentielles Programm dar
→ Zählt nicht zur Klasse der Maschinenprogramme
→ interpretiert die eigenen Programme nur teils partiell
→ sollten unterbrechbar sein
→ Interpreter - Ausführung von Systemaufrufen:
⇒1. Prozessorstatus unterbrochener Programme sichern
⇒2. Systemaufruf interpretieren
⇒Prozessorstatus wiederherstellen

Maschinenprogrammebene:

Zwei Sorten von Befehlen:
→ unpriviligierte Befehle:
⇒Wird von der CPU direkt ausgeführt
→ priviligierte Befehle:
⇒Werden vom Betriebssystem ausgeführt
⇒explizit als Systemaufrufe codiert
⇒implizit als Ausnahmen ausgelöst

Ausführungsablauf bei priviligierten Befehlen:
⇒1. CPU interpretiert Maschinencode befehlsweise
⇒2. Bei Ausnahmen startet das Betriebssystem
⇒3. Interpretiert Programme des BS befehlsweise
⇒Folge von Punkt 3:
⇒4. BS interpretiert Maschinencode befehlsweise
⇒5. CPU wird zum weitermachen verleitet

Ausnahmebehandlung:

→ Ausnahmebehandlung ist zwingend
→ Es gibt zwei Varianten:
→ 1. Trap
⇒Abfangung für Ausnahmen von interner Ursache
⇒falsche Adressierungsart oder Rechenoperation
⇒Systemaufruf, Adressraumverletzung,unbekanntes Gerät
⇒Seitenfehler im Falle lokaler Ersetzungsstrategien
⇒synchron, vorhersagbar, reproduzierbar
⇒Behandlungsmodelle: Beendigung und Wiederaufname
→ 2. Interrupt:
⇒Unterbrechung durch Ausnahmen von externer Ursache
⇒Ein externer Prozess signalisiert einen Interrupt
⇒Z.b. Beendigung einer DMA- bzw. E/A-Operation
⇒Z.b. Seitenfehler im Falle globaler Ersetzungsstrategien
⇒Unterbrechungsbehandlung muss Nebeneffektfrei sein
⇒Aktiver Prozess wird unterbrochen, nicht abgebrochen
⇒asynchron, unvorhersagbar, nicht reproduzierbar
⇒Behandlungsmodell: Nur Wiederaufnahmemodell

Aktives Warten:
→ Prozess frägt beim warten immer wieder nach
→ Vergeudet nicht unbedingt CPU-Zeit
→ Benötigt keine Unterstützung durch das Betriebssystem

Passives Warten:
→ Prozess Schläft, bis er benachrichtigt wird

Betriebsarten

Stapelbetrieb:

Abgesetzter Betrieb:
→ Verwendung von Satellitenrechner und Hauptrechner
→ Satellitenrechner sendet Eingabe an Hauptrechner
→ Hauptrechner verarbeitet die Eingabe
→ Hauptrechner schickt Ergebnis an Satellitenrechner
→ +: Entlastung durch Spezialrechner

Überlappte Ein-Ausgabe:
→ Durch Interrupts gleichzeitige Prozesse
→ Speicherdirektzugriff (DMA) statt programmierte IO
→ nebenläufige Programmausführung möglich
→ Problem: Leerlauf beim Auftragswechsel

Überlappte Auftragsverarbeitung:
→ Verarbeitungsstrom auszuführender Programme
→ Technik: Prefetching oder Auftragseinplanung
→ Probleme: CPU Monopolisierung, IO Leerlauf

Abgesetzte Ein-Ausgabe: Spooling
→ Abwechselnd: CPU-Stoß und IO-Stoß
→ Entkopplung durch Pufferbereiche

Mehrprogrammbetrieb:
→ Multiplexen der CPU
→ Mehrere Programme gleichzeitig im Hauptspeicher
→ Nutzung von Aktiven/Passiven Warten
→ Wichtiges Tool: Adressraumschutz:
⇒Vor Laufzeit: Schutz durch Eingrenzung
⇒Zur Laufzeit: Schutz durch Segmentierung

Dynamisches Laden:
→ Problem: Programm zu groß für den Arbeitsspeicher
→ Lösung:
⇒Zerteilung des Programms in kleine Teile
⇒Das Nachladen ist Programmiert
⇒Nachteil: Programmierer manuell einzubauen

Echtzeitbetrieb:

→ System muss ständig betriebsbereit sein
→ Ergebnisrückgabe innerhalb vorgegebener Zeit
→ externe (physikalische) Prozesse definieren Verhalten für nicht termingerechte Ergebnisrückgabe

Terminvorgabe:
→ weich:
⇒Ergebnis weiterhin nutzbar
⇒Ergebnis wird nur mit der Zeit immer wertloser
⇒Terminverletzung ist tolerierbar
→ fest:
⇒Ergebnis ist wertlos
⇒Ergebnis wird verworfen
⇒Terminverletzung ist tolerierbar
→ hart:
⇒kein Ergebnis bedeutet großes Problem
⇒Terminverletzung ist nicht tolerierbar

Mehrzugangsbetrieb

→ Nur sinnvoll mit CPU- und Speicherschutz

Dialogbetrieb:
→ mehrere Benutzer gleichzeitig
→ Benutzereingaben und Verarbeitung wechseln sich ab
→ Zugang über Dialogstation (z.b. Terminal)
→ Problem: Monopolisierung der CPU möglich

Dialogorientiertes Monitorsystem (Hintergrundbetrieb):
→ Prozesse im Vordergrund starten
→ Prozesse im Hintergrund vollziehen
→ mehrere Aufgaben werden parallel bearbeitet
→ Problem: Hauptspeichergröße

Teilnehmerbetrieb:
→ eigene Dialogprozesse werden interaktiv gestartet
→ Zeitscheibe um CPU-Monopolisierung vorzubeugen

Teilhaberbetrieb:
→ Client-Server-System

Systemmerkmale

Symmetrische Simultanverarbeitung:
→ Mehrere Prozesse gekoppelt über gemeinsames Verindung
→ Stellt homogenes System dar

Speichergekoppelter Multiprozessor:
→ Alle Prozesse nutzen den Hauptspeicher
→ Stellt heterogenes System dar
→ Assymetrische Architektur:
⇒hardware bedingter assymetrischer Betrieb
→ Symmetrishce Architektur:
⇒Betriebssystem legt Multiprozessorbetrieb fest

Parallelverarbeitung:
→ N Prozessoren können:
⇒N Programme echt parrallel ausführen
→ Jeder dieser Prozessoren kann multiplexen
⇒Jeder Prozessor kann einzeln parallelisiert werden

Schutzvorkehrungen:
→ Jeden Prozessadressraum in Isolation betreiben
→ Prozessen eine Zugriffsbefähigung erteilen
→ Objekten eine Zugriffkontrolliste geben
→ 1. Methode: Schutz durch selektive Autorisierung
→ 2. Methode: Zugriffsrechtmatrix
→ 3. Methode: Schutzring names Multics
⇒Einzelne Ringabschnitte mit unterschiedlichen Rechten
⇒Von außen nach innen mehr Rechte
⇒ring fault bei Rechteverletzung
→ 4. Methode: Schutzgatterregister:
⇒Permanent benötigte Programme werden isoliert
⇒Abgrenzung von den Anwendungsprogrammen
⇒Schutzgatter kann manuell ausgeschalten werden

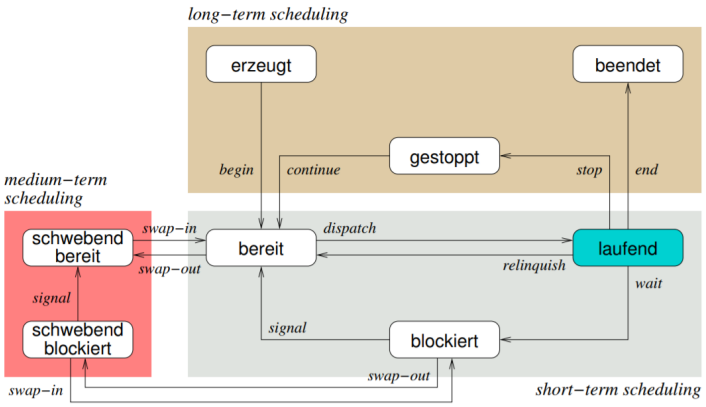
Umlagerung nicht ausführbereiter Programme:
→ auch swapping genannt
→ schafft Platz im Arbeitsspeicher
→ Probleme: Fragmentierung, Verdichtung

Umlagerung ausführbereiter Programme:
→ Nicht benötigte Teile werden ausgelagert
→ Zugriff auf ausgelagerte Teile unterbricht Prozess
→ Intensiver Wechsel zw. aus und einlagern

Prozessverwaltung

Begriffe:
→ Einplanung: Reihenfolenbildung von Aufträgen
→ Einlastung: Zuteilung der Betriebsmittel

Programmfaden:
→ Einplanungseinheit für die Prozessorvergabe ist der Faden
→ Lauf- und Wartephasen betreiben einen Rechner stoßartig
→ Fäden überdecken passives Warten anderer Fäden



Kurzfristige Planung (short-term scheduling):
→ Wichtig für Mehrprozessbetrieb
→ bereit: Prozess ist in der Bereitliste
→ laufend: Prozess vollzieht seinen CPU-Stoß
→ blockiert: Prozess erwartet Betriebsmittelzuteilung

Mittelfristige Planung (medium-term scheduling):
→ Anhand der Umlagerung kompletter Programme
→ schwebend bereit:
⇒Das Exemplar eines Prozesses ist augelagert
⇒Die Einlastung des Prozesses ist außer Kraft
→ schwebend blockiert:
⇒ausgelagerter ereigniserwartender Prozess

Langfristige Planung (long-term scheduling):
→ Nutzung von Lastkontrolle
→ erzeugt: fertig zur Programmverarbeitung
→ gestoppt: erwartet Fortsetzung / Beendigung
→ beendet: Prozess erwartet seine Entsorgung

Gütemerkmale:
→ Benutzer:
⇒Antwort-/Durchlaufzeit, Termine, Vorhersagbarkeit
⇒Prozessausführung unabhängig von der Systemlast
→ System:
⇒Durchsatz, Auslastung, Gerechtigkeit
⇒Dringlichkeit, Lastausgleich
⇒Gleichmäßige Betriebsauslastung

User-Threads:
→ Threads mit nichtpriviligiertem Code
→ blockierende syscalls blockieren andere User-Threads
→ Schedulingstrategie vom Programmierer definiert
→ Können effizient umgeschalten werden
→ Nicht für Multiprozessorbetrieb ausgelegt

Kernel-Threads:
→ Threads mit priviligiertem Code
→ Erzeugung teurer als User-Threads
→ blockierende syscalls blockieren keine anderen Threads
→ Schedulingstrategie durch Betriebssystem vorgeben
→ Umschalten findet immer im Systemkern statt
→ Kann für Multiprozessorbetrieb sinnvoll sein

Klassifikation der Prozesseinplanung:
→ kooperative Planung (FCFS):
⇒für voneinander abhängige Prozesse
⇒CPU-Entzug nicht zugunsten anderer Prozesse
⇒laufender Prozess gibt CPU nur mit Systemaufruf ab
⇒CPU Monopolisierung möglich
→ preemptive Planung (RR, VRR, SRTF):
⇒für voneinander unabhängige Prozesse
⇒CPU-Entzug zugunsten anderer Prozesse möglich
⇒ereignisbedingte Verdrängung laufender Prozesse
⇒CPU Monopolisierung nicht möglich

→ deterministische Planung:
⇒Alle Prozesszeiten sind bekannt
⇒Einhaltung von Zeitgarantien sichergestellt
⇒CPU-Auslastung vorhersagbar
→ probabilistische Planung (SPN, SRTF, HRRN):
⇒Prozesszeiten unbekannt und nur approximierbar
⇒Keine Zeitgarantie möglich
⇒CPU-Auslastung nicht vorhersagbar

→ statische Planung:
⇒Vor Betrieb des Prozesses
⇒Keine Planung im laufenden Betrieb
⇒Ergebnis der Vorberechnung ist kompletter Ablaufplan
⇒Begrenzt auf strikte Echtzeitsysteme
→ dynamische Planung:
⇒während des Betriebs des Prozesses
⇒Stapelsysteme, interaktive Systeme, verteilte Systeme
⇒schwache und feste Echtzeitsysteme

→ assymetrische Planung:
⇒wichtig bei assymetrischen Multiproessorsystemen
⇒optional auf symmetrischen Multiprozessorsystemen
⇒ungleiche Verteilung mehrer Prozesse auf CPUs
→ symmetrische Planung:
⇒identische Prozessoren
⇒Lastausgleich bei Verteilung von Prozessen auf CPUs

Bekannte Verfahrensweisen:
→ FCFS (First Come First Serve):
⇒Prozesseinplanung nach Reihenfolge der Ankunft
⇒Prozesse mit langen Rechenstößen werden begünstigt
⇒Prozesse mit kurzen Rechenstößen werden benachteiligt
⇒−: Konvoieffekt (Abwechseln kurze/lange Prozesse)
→ RR und VRR: (Round Robin)
⇒Verdrängende Variante von FCFS
⇒Verringerung der Benachteiligung aus FCFS
⇒Prozesse werden nach ihrer Ankuftszeit eingeplant
⇒Prozessumplanung in regelmäßigen Zeitabständen
⇒Durch Zeitschreibe entsteht CPU-Schutz
⇒Weiterhin Problem des Konvoieffekts
⇒VRR: Variable Zeitscheibe, nicht voll-verdrängend
⇒VRR: Zusätzlich Vorzugsliste für IO
→ SPN , HRRN , SRTF: (Shortest Process next)
⇒SPN: Schnellster Prozess als nächstes
⇒SPN: Prozessverhungern möglich
⇒HRRN: Hungerfreies SPN durch Alterungswichtung
⇒SRTF: Prozesseinplanung nach Bedienzeit
⇒SRTF: unregelmäßige Prozessumplanung
⇒SRTF: Verdrängt wird in eine Bereitliste
→ MLQ, MLFQ:
⇒Kombination der obrigen Strategien
⇒Prozesseinplanung nach Typ und Ankuftszeit
⇒Unregelmäßige Prozessumplanung
⇒Jede Queue hat eigene Einplanungstrategie
⇒Gut für kurze Prozesse, schlecht für lange

Prozesssynchronisation

Grundbegriffe:
→ Kausalität: Beziehung zw. Ursache und Wirkung
→ Gleichzeitige Prozesse:
  ⇒vertikal: time sharing (multiplexen)
  ⇒horizontal: multiprocessing
→ Gekoppelte Prozesse: Zugriff auf gleiche Daten
→ Sequentialisierung:
  ⇒Koordinierung durch atomare Operationen
→ unsicherer Zustand:
  ⇒früher oder später kommt es zu Verklemmung
→ Einseitige Synchronisation:
  ⇒Nur ein beteildigter Prozess
  ⇒erfolgt logisch oder bedingt
→ Mehrseitige Synchronisation:
  ⇒alle der beteildigten Prozesse betroffen
  ⇒blockierend oder nicht-blockierend möglich
Nebenläufigkeit:
→ Keine Abhängigkeit zw. Ereignissen
  ⇒Datenabhängigkeit darf nicht verletzt werden
  ⇒Zeitbedingung anderer darf nicht verletzt werden
→ Ursachen:
  ⇒Interrupts / preemptives Scheduling
  ⇒Mono- Multiprozessor-Threads / Unix-Prozess-Signale
Verklemmung:
→ Deadlock: gutartig, Prozesse sind blockiert
→ Livelock: bösartig, Prozesse zw. laufend und bereit
→ notwendige Bedingung:
  ⇒1. wechselseitiger Ausschluss
  ⇒2. Nachforderung eines oder mehrerer Betriebsmittel
  ⇒3. Unentziehbarkeit der zugeteilten Betriebsmittel
→ notw. und hinreichende Bedingung:
  ⇒4. zirkuläres Warten
  ⇒reicht vorzubeugen um Dealocks zu vermeiden
  ⇒kann mittels Virtualisierung realisiert werden
Philosophenproblem:
→ 5 Philosophen 5 Gablen
→ Philosoph benötigt 2 Gabeln zum Essen
→ Alle nehmen gleichzeitig linke Gabel
→ Alle Philosophen verhungern
Blockierende Synchronisation:
→ Locking Techniken wie Semaphore
→ Kritische Abschnitte, vor welchen blockiert wird
→ −: Verklemmung, Effizienzverlust
Nicht-blockierende Synchronisation:
→ Keine kritischen Abschnitte
→ Verwendung von atomaren Operationen
  ⇒Prozessor-Befehle wie Compare-and-Swap
  ⇒nebenläufigkeitsfreie Variablenmodifikation ohne Locks
→ −: hohe Komplexität und Verhungern möglich

Semaphoren:
→ Krit. Abschnitt mit wechselseitigem Ausschluss sichern
→ Eignen sich für ein- und mehrseitige Synchronisation
→ Ganzzahlige Variable mit zwei Operationen:
  ⇒P : wait();
  ⇒V : release();
→ Operationen sind logisch und physisch unteilbar
→ Auf P und V maximal ein Prozess gleichzeitig:
  ⇒Prozesse werden in Warteschlangen gehalten
  ⇒Nichtpassierende Prozesse werden schlafengelegt
→ Atomarität wird auf Befehlssatzebene umgesetzt
  ⇒nicht durch "normale" C-Anweisungen umsetzbar
→ Mutex: eine Semaphorenspezialisierung
  ⇒Prüft Eigentümerschaft bei Freigabe
  ⇒lässt Freigabe damit bedingt zu
  ⇒für einseitige und mehrseitige Synchro. geeignet

Monitor:
→ Monitore sind abstrakte Datentypen
→ Monitor Behandelt Parallelitätsprobleme automatisch
→ Bei Blockierung muss Prozess den Monitor verlassen
→ Konzept:
  ⇒Mehrere Warteschlangen (WS)
  ⇒Prozesse warten immer außerhalb des Monitors
  ⇒MonitorWS: Prozesse warten auf Monitoreintritt
  ⇒EreignisWS:Prozesse warten auf Wartebedingungsende
  ⇒Verwendet bei Blockierung Bedingungsvariablen
  ⇒nicht blockierend: Vorrang Signalgeber
  ⇒blockierend: Vorrang Signalnehmer
→ Hansen-Methode (blockierend):
  ⇒Programmier-Primitive: wait() / broadcast()
  ⇒Signalisierung lässt Signalgeber den Monitor verlassen Nachdem er alle Signalnehmer bereit gesetzt hat
→ Hoare-Methode (blockierend):
  ⇒Programmier-Primitive: signal()
  ⇒Signal 1: Versetzt laufenden Prozess ins Warten
  ⇒Signal 2: Prozess aus Warteschlange wird fortgesetzt
  ⇒Signalisierung lässt Signalgeber den Monitor verlassen und genau einen Signalnehmer fortsetzen (atomar)
→ Mesa-Methode (nicht blockierend):
  ⇒Programmier-Primitive: signal()
  ⇒signal unterbricht laufenden Prozess nicht
  ⇒signal verschiebt Prozess in Monitorwarteschlange
  ⇒Signalisierung lässt Signalgeber im Monitor fortfahren nachdem bereit-setzen eines oder aller Signalnehmer

Betriebsmittelverwaltung

Betriebsmittelklassifikation:
→ wiederverwendbare: (begrenzt)
  ⇒Anforderung durch mehrseitige Synchronisation
  ⇒Zusätzliche Unterscheidung: teilbar und unteilbar
  ⇒Werden belegt, benutzt und freigegeben
  ⇒Beispiel: CPU, RAM, GPU
→ konsumierbare: (unbegrenzt):
  ⇒Anforderung durch einseitige Synchronisation
  ⇒Werden produziert, empfangen, benutzt und zerstört
  ⇒Beispiel: Signale und Traps

Ziele:
→ Durchsetzung der vorgegebenen Betriebsstrategien
→ Optimale Realisierung in Bezug auf relevante Kriterien
→ Wichtig: Keine Verhungern und keine Verklemmung:

Aufgaben:
→ Buchführung über vorhandene Betriebsmittel
→ Steuerung der Verarbeitung von Anforderungen
→ Betriebsmittelentzug bei fehlerhaften Prozessen

Verfahrensweisen:
→ Statisch:
  ⇒Aufgabenbewältigung vor der Laufzeit
  ⇒Risiko: suboptimale Betriebsmittelauslastung
→ Dynamisch:
  ⇒Aufgabenbewältigung während der Laufzeit
  ⇒Risiko: Verklemmung abhängiger Prozesse

Speicherverwaltung: Adressräume

Reale Adressen:
→ lückenhafter, wirklicher Hauptspeicher
→ Die realen Adressen, von der Hardware gegeben
→ Vorgegebene Adressen und Adressräume
Logische Adressen:
→ lückenloser, wirklicher Hauptspeicher
→ Getrennte reale Adressbereiche werden linearisiert
→ Zweck:
⇒Sicherheit, Virtualisierung, bessere Verwaltung
Virtuelle Adresse:
→ lückenloser, scheinbarer Hauptspeicher
→ entkoppelt von der Lokalität im Arbeitsspeicher
→ Kann größer als der Arbeitsspeicher sein (paging)
→ Adresse wird auf den Hauptspeicher abgebildet
→ Dafür gibt es dann Abbildungstabellen
→ Navigation zum Speicher durch Abbildungstabellen

Eindimensionaler Adressraum:
→ Typische Seitengröße bei Seitenadressierung: 4096
→ Adressaufbau: Seitennummer p und Offset o (Versatz)
→ Zusätzlich wird eine Seitentabelle benötigt
→ Adressbildung:
⇒Mit p Adresse in Seitentabelle Suchen
⇒p in Zahl umrechnen und damit Eintragsindex suchen
⇒Bei mehrstufigen: p gleichmäßig Aufteilen
⇒o wird so wie es ist weiterhin übernommen
Zweidimensionaler Adressraum:
→ Aufteilung des Adressraums in Segmente
→ Zweikomponenten Adresse:
⇒Segmentname S und Adresse A mit p und o (siehe oben)
→ Adressbildung (ohne p und o):
⇒Mit Segmentname auf Segmenttabelle zugreifen
⇒Tabellen Eintrag mit Adresse verrechnen
→ Adressbildung (mit p und o):
⇒Mit Segmentname auf Segmenttabelle zugreifen
⇒Mit diesen Eintrag auf die Seitentabelle zugreifen

Private Adressräume:
→ Illusion eigenes Adressraums für BS und Maschinenprogs.
→ Spezialhardware verhindert ausbrechen aus Adressraum
→ Sinn: strikte Isolation ganzer Adressräume

Seitentabelle Berechnung bei Seitenadressierung:
→ Adressraum : (Speicherseite : Eintrag)
⇒Jeder Seite muss adressiert werden
⇒Speicherseite : Eintrag → Einträge pro Seite
⇒Adressraum : Einträge pro Seite → Ergebnis

Seitendeskriptor:
→ Von MMU vorgegebener Verbund von Attributen:
⇒seitenausgerichtete reale Adresse
⇒Schreibschutzbit / Präsensbit
⇒Referenzbit / Modifikationsbit
→ Seitendeskriptor des BS in shadow page table
Filedeskriptor:
→ Prozesslokale Integerdatei
→ Wird von einem Prozess verwendet
→ Zugriff auf Dateien, Geräte, Pipes, Sockets
Segmentdeskriptor:
→ Von MMU vorgegebener Verbund von Attributen:
⇒Basis: Segmentanfang im Arbeitsspeicher
⇒Limit: Segmentlänge als Anzahl der Granulate
⇒Typ (Text, Daten, Stapel)
⇒Zugriffsrechte (lesen, schreiben, ausführen)
⇒Expansionsrichtung / Präsensbit

Speicherbereiche

Code-Segment:
→ Auszuführenden Programmcode
Data-Segment:
→ inizialisierte Globale und statische Variablen
Heap-Segment:
→ Von malloc reservierter Speicher
Block-Storage-Segment (BSS):
→ nicht inizialisierte globale/statische Variablen
Stack-Segment:
→ lokale (Pointer-) Variablen
→ Übergebene Argumente
→ Platz für Zwischenergebnisse
→ Rücksprungadresse / Frame-Pointer

Adressraumschutz durch Eingrenzung:
→ Begrenzungsregister legen Adressbereich fest
→ Dieser ist im physikalischen Adressraum
→ Auf diesen werden Speicherzugriffe beschränkt

Seitennummer und Versatz berechnen:
→ Bits der Seitengröße berechnen
→ logische Adresse unterteilen:
⇒Hintere Bits sind für Seitengröße
→ beide Adressen mit 0er vorne Auffüllen

Platzierungsstrategien:

→ Bitkarte:
⇒für Hohlräume fester Größe
→ Lochliste:
⇒für Hohlräume variabler Größe

Zuteilungsverfahren:
→ Alle folgenden sind Listenbasiert
→ Bei allen kann externer Verschnitt auftreten
→ Freispeicherverschmelzung: first-fit schneller als worst-fit
→ worst-fit:
⇒absteigend nach Lochgröße sortiert
⇒Größtes passendes Loch wird gesucht
⇒+: Suchaufwand ist klein
⇒−: Speicherverschnitt ist groß
→ best-fit:
⇒aufsteigend nach Lochgröße sortiert
⇒kleinstes passendes Loch wird gesucht
⇒+: Speicherverschnitt ist klein
⇒−: Suchaufwand ist groß
→ first-fit:
⇒Erstes passendes Loch nutzen
⇒+: Suchaufwand ist klein
⇒−: Speicherverschnitt ist groß
→ next-fit:
⇒Round-Robin-Variante von first-fit
⇒Beginnt Suche beim zuletzt zugeteilten Loch

→ Interne Fragmentierung:
⇒Speicherblöcke nur teilweise befüllt
⇒Ist (durch das Betriebssystem) unvermeidbar
⇒Buddy: Entstehung durch das Aufrunden
⇒Kein Zugriffsfehler bei Zugriff auf ungenutzen Bereich
→ Externe Fragmentierung:
⇒Angeforderte Größe zu groß für jedes Loch
⇒Dadurch Zerstückelung eines Speicherraums
⇒Verringerung durch Speicherblock-Verschmelzung
⇒Auflösung durch Kompaktifizierung möglich

Buddie-Verfahren:
→ **Datenblock kann vorinitialisiert sein!!!**
→ Hier Beschreibung des Binären Buddyverfahrens
→ Neuer Speicher wird zu nächster 2er Potenz aufgerundet
→ 2^n wird in 2^n großem Block gespeichert!
→ Passender Speicher wird itterativ gesucht und verwendet
→ Ist kein Speicher vorhanden:
⇒Halbieren des ersten größeren Speichers, bis er passt
→ Aufeinanderfolgende Buddyadressen um ein Bit different
→ Lochlisten Einträge:
⇒Links mit Zweierpotenzen sind Blockgrößen
⇒Daneben: Adresse/n mit freiem Block jeweiliger Größe
→ Bemerkungen:
⇒Es gibt keinen externen Verschnitt

1. p1 = Malloc(200) / 2. p2 = Malloc(128) / 3. p3 = Malloc(500) / 4. free(p1) / 5. free(p2)

<table>
<tr><td colspan="4">1024</td></tr>
<tr><td colspan="3">512</td><td>512</td></tr>
<tr><td>p1</td><td colspan="2">256</td><td>512</td></tr>
<tr><td>p1</td><td>p2</td><td>128</td><td>512</td></tr>
<tr><td>p1</td><td>p2</td><td>128</td><td>p3</td></tr>
<tr><td>256</td><td>p2</td><td>128</td><td>p3</td></tr>
<tr><td>256</td><td colspan="2">256</td><td>p3</td></tr>
<tr><td colspan="3">512</td><td>p3</td></tr>
</table>

Ladestrategien:

→ Umsetzung durch MMU (Memory Management Unit)
→ Der TLB (Tranlation Lookaside Buffer) untersützt MMU
⇒spezieller Cache mit Infos vom Seitendeskriptor
⇒Speichert Ergebnis: Logisch → physikalisch

Arbeitsweisen:
→ Demand-Paging:
⇒Läd Adresse erst nach Ansprechen in den RAM
⇒Nichtbenutzte Seiten werden ausgelagert
→ Anticipatory:
⇒Vorausladen bzw. Prefetching
⇒Zur Vermeidung von Folgefehler
⇒Heuristik liefert Hinweise über zukünftige Zugriffe

Fehler:
→ Page/Segment-fault:
⇒Speicher/Segment nicht im Hauptspeicher
⇒Entstehen z.b. durch ungültige Zeiger
⇒Present-Bit des Adressraums ist nicht gesetzt
⇒Adresse ist somit vmtl. ausgelagert (Festplatte)
⇒MMU löst einen Trap aus
⇒Zugriffsfehler: Schwerer Seitenfehler

Ersetzungsstrategien:

Lokale vs. globale Seitenersetzung:
→ lokal:
⇒Suchraum: Menge residenter Seiten des Prozesses
⇒ein Seitenfehler ist vorhersag-/reproduzierbar
→ global:
⇒Suchraum: Menge alles residenter Seiten aller Prozesse
⇒ein Seitenfehler ist unvorhersag-/unreproduzierbar

Lokalitätsprinzip:
Wenn ein Prozess eine Stelle in seinem Adressraum referenziert, dann wird er wahrscheinlich dieselbe Stelle oder eine andere Stelle in direkter Umgebung referenzieren

Seitenflatter:
→ Seitenein-/auslagerungen dominiert Systemaktivität
→ Sofortiges Einlagern kürzlich ausgelagerter Seiten
→ ein mögliches Phänomen der globalen Seitenersetzung
→ Verschwindet meist von alleine wieder

Freiseitenpuffer:
→ Um schneller einen freien Seitenrahmen zu finden
→ FIFO mit Cache für potentiell zu ersetzende Seiten
→ Seiten im Cache werden wie bei SSD überschrieben
→ Zugriffsfehler auf Seite im Cache:
⇒Reklamierung und Neusetzen des Präsensbits

Arbeitsmenge:
Die kleinste Sammlung von Programmtext und -daten, die in einem Hauptspeicher vorliegen muss, damit effiziente Programmausführung zugesichert werden kann.

Strategien:
→ FIFO (First in First out):
⇒Ersetzt wird zuletzt eingelagerte Seite
⇒Implementierung: Verkettete Liste
⇒Mehr Seitenrahmen führen zu mehr Seitenfehlern
→ LFU (Least frequently used):
⇒Ersetzt wird die am seltensten referenzierte Seite
⇒Implementierung durch Zähler
→ LRU (Least recently used):
⇒Ersetzt wird am längsten nicht referenzierte Seite
⇒Siehe: Verschiedenes

Dateisysteme:

Kontinuierliche Speicherung:
→ Dateispeicherung in aufsteigend nummerierten Blöcken
→ Zur Identifizieren: Anfangsblock und Größe
→ +: Schnelles Lesen, da keine externe Fragmentierung
→ +: Geeignet für Echtzeitsysteme
→ +: Freien Speicher finden ist schwierig
→ −: Interne Fragmentierung
→ −: dynamische Erweiterung sehr aufwändig

Verkettete Speicherung:
→ Speicherung von Daten in verketteten Blöcken (Liste)
→ +: Dynamische Erweiterung einfach möglich
→ −: hohe Fehleranfälligkeit / externe Fragmentierung

Indiziertes Speichern:
→ Spezieller Platten enthält Blocknummern (wie Adressen)
→ Datenblock einer Datei über Blocknummer identifizierbar
→ Bei großen Dateien mehrere Datenblöcke
→ Extends: größerer Datenblock → kontinuierlich teils mögl.
→ −: Interne Fragmentierung

Freispeicherverwaltung:
→ Speicherbereich wird in gleich Große Blöcke unterteilt
→ Bitvektor speichert pro Block belegt / nicht belegt
→ verkettete Liste repräsentiert freie Blöcke
→ Besser: Aufeinanderfolgende Blöcke Zusammenfassen
→ Besser: Statt Liste: Datenblöcke zeigen auf freien Speicher

Unix Blockstruktur:

Bootblock	Superblock	Inodeliste	Datenblock

→ Boot: Infos zum Laden eines initialen Programmes
→ Super: Anzahl Blöcke / Inodes und Attribute
→ Inode: Dateiart,UserId,rights,größe,linkzähler,hard links
⇒Zusätzlich Zeiten: Erstellung, Zugriff, letzte Änderung

Journaling-File-Systems:
→ Änderungen als Transaktionen dokumentiert im Log-File
→ Log wird vor der Transaktion beschrieben
→ Inkonsistenz wird beim Booten durchs Log-File vermieden
→ −: Undo nach erkanntem Transaktionsproblem
→ Optimiert: Checkpoint (Platte in konsistentem Zustand)
⇒Log-File-Einträge bis checkpoint löschen

Raid 0: Gestreifte Platte (Paritätsfrei):
→ Daten werden über mehrere Platten gespeichert
→ +: Schnelles Lesen und schreiben
→ −: Systemausfall bei Plattenausfall / keine Fehlertoleranz

Raid 1: Gespiegelte Platten (Paritätsfrei)
→ Zwei oder n Platten mit gleichen Daten
→ Verknüpfung von Raid 0 und Raid 1 möglich
→ +: n-1 Platten können ausfallen
→ +: Schnelles Lesen (Nicht Schreiben)
→ −: Hoher Speicherbedarf

Raid 4: Paritätsplatte:
→ Daten werden über mehrere Platten verteilt gespeichert
→ Eine Paritätsplatte P speichert Paritätsinformationen
→ +: Eine Platte kann ausfallen und schnelles Lesen
→ +: mind. 3 Platten, beliebig viele mehr möglich
→ −: P hoch belastet, da bei jedem Schreiben einbezogen

Raid 5: Verstreuter Paritätsblock:
→ Paritätsblöcke werden über alle Platten verteilt
→ Vor und Nachteile wie Raid 4
→ +: Last der Paritätsplatte wird verteilt
→ Raid 6: zwei Paritätsblöcke pro Platte

Verschiedenes:

Hacking:

→ strcpy() und strcat() als Sicherheitslücken

Least Recently Used:

→ Periodische Unterbrechung → Hintergrundrauschen
→ Altersstruktur:
⇒Benötigt in Software implementiertes Schieberegister
⇒Zusätzlich Zeitgeber für periodische Unterbrechung
⇒Referenzbitprüfungs nach jedem Zeitintervall
⇒Ersetzt: Global älteste Seite im Aging Register

tick	Ref.	aging register
		00000000
n	1	10000000
n+1	0	01000000
n+2	1	10100000

→ Zweite Chance:
⇒Ähnlich zum FIFO-Prinzip
⇒Nutzt Zeitintervall zur periodischen Unterbrechung
⇒Ersetzung der Ersten Seite bei der Referenzbit 0 ist
⇒Wenn alle Referenzbits 1: FIFO Prinzip nutzen!
→ Dritte Chance:
⇒Übernimmt alle Merkmale von Second Chance
⇒Zusätzliche Nutzung des Modifikationsbits
⇒(0,0): unbenutzt = beste Wahl
⇒(1,0): beschrieben = Naja
⇒(0,1): gelesen = Naja
⇒(1,1): kürzlich beschrieben = Schlecht
⇒Zwei Umläufe für jede eingelagerte Seite

Wertetabelle:

Hex	Dez	Binär	Name	Name	2er	Dez	Pot
0	0	0000	KiB	KibiB	2^{10}	2	2^1
1	1	0001	MiB	MibiB	2^{20}	4	2^2
2	2	0010	GiB	GibiB	2^{30}	8	2^3
3	3	0011	TiB	TebiB	2^{40}	16	2^4
4	4	0100	PiB	PebiB	2^{50}	32	2^5
5	5	0101				64	2^6
6	6	0110				128	2^7
7	7	0111				256	2^8
8	8	1000				512	2^9
9	9	1001				1024	2^{10}
A	10	1010	Name	10er	2er	2048	2^{11}
B	11	1011	kB	10^3	2^{10}	4096	2^{12}
C	12	1100	MB	10^6	2^{20}	8192	2^{13}
D	13	1101	GB	10^9	2^{30}	16384	2^{14}
E	14	1110	TB	10^12	2^{40}	32768	2^{15}
F	15	1111	PB	10^15	2^{50}	65536	2^{16}

Gewichtsklasse von Prozessen:

→ Schwergewichtiger Prozess:
⇒Eigener Adressraum und Ausführungsfaden
→ Leichtgewichtiger Prozess:
⇒teilt sich Adressraum mit anderes Prozessen
→ Federgewichtiger Prozess:
⇒weder eigener Adressraum noch Ausführungsfaden
⇒wird nicht vom Kernel verwaltet

Kleinigkeiten zum Einfügen:

Inode-Aufgaben:

Gegeben:
/Desktop/Inode:

71	d...	3	...	4096	...	.
66	d...	9	...	4096	...	..
58	d...	2	...	4096	...	deeper
27	l...	1	...	53	...	file.txt → deeper/file.txt
93	-...	1	...	25	...	prog.c

/Desktop/Inode/deeper:

58	d...	2	...	4096	...	.
71	d...	3	...	4096	...	..
94	-...	1	...	6	...	file.txt

Lösung:
→ Inode mit: Ordner, Datei, Verknüpfung
→ Ordner: Inhalt + st_ino + . + ..
→ Datei: Inhalt unbekannt: ???
→ Verknüpfung: Verknüpfte Datei
→ Alle Inode (st_ino) Nummer müssen abgearbeitet werden
→ Jede Inode Nummer nur einmal

Inode			Inhalt Datenblöcke	
st_ino:	71		71 .	66 ..
st_nlink:	3	⟶	58 deeper	27 file.txt
st_size:	4096		93 prog.c	

st_ino:	66		66 .	71 Inode
st_nlink:	9	⟶		
st_size:	4096			

st_ino:	58		58 .	71 ..
st_nlink:	2	⟶	94 file.txt	
st_size:	4096			

st_ino:	27		
st_nlink:	1	⟶	deeper/file.txt
st_size:	53		

st_ino:	93		???	
st_nlink:	1	⟶		
st_size:	25			

st_ino:	94		???	
st_nlink:	1	⟶		
st_size:	6			

Bemerkung:
→ Root Knoten zeigt mit . und .. auf sich selbst
→ Inode auf Datei wird getrennt von Datei gespeichert

UNIX-UFS-Dateisystem:

→ Pfadnamen ohne '/' am Anfang:
⇒Werden relativ zum aktuellen Verzeichnis interpretiert

→ Hierarchisch organisierter Namensraum:
⇒Die Absteigende Baumstruktur bei Dateisystemen
⇒Kontext (Verzeichnis) als flachen Namensraum
⇒Gleiche Namen in unterschiedlichem Kontext möglich
⇒Pro Verzeichnis mehrmals gleicher Name Nicht möglich

Dateisystem:
→ Symbolic-Links:
⇒Können existieren, obwohl Ziel bereits gelöscht
⇒Kann auf Verzeichnisse verweisen
→ Hard-Link:
⇒Pro Reguläre Datei mind. einer im selben Dateisystem
⇒Pro Verzeichnis mind. 2 Hard-Links
⇒Können nur auf Dateien verweisen
⇒Auf Datei: Anlegen nur im selbem Dateisystem

Prozesskontrollblock:
→ Als Tabelle vorstellbar mit folgenden Einträgen:
⇒Zustande des Prozesses / Befehlszähler
⇒CPU-Register / Stack-Pointer
⇒Zustande der geöffneten Dateien / aktuelles Verzeichnis
⇒Verwaltungsinformationen / UID Eigentümer