Betriebssystem SINIX

CES Buch 2

Werkzeuge zur C-Programmierung Systemaufrufe, C-Funktionen und Makros

Ausgabe Februar 1986 (CES V1.0B, 1.0C, 2.0)

Bestell-Nr. U2581-J-Z95-1
Printed in the Federal Republic of Germany
8600 AG 7863. (10800)

SINIX ist der Name der Siemens-Version des Softwareproduktes XENIX.
XENIX ist ein Warenzeichen der Microsoft Corporation.
XENIX ist aus dem UNIX System III unter Lizenz der Firma AT & T entstanden.

Copyright © an der Übersetzung Siemens AG, 1984, alle Rechte vorbehalten.

Vervielfältigung dieser Unterlage sowie Verwertung ihres Inhalts unzulässig, soweit nicht ausdrücklich zugestanden.

Im Laufe der Entwicklung des Produktes können aus technischen oder wirtschaftlichen Gründen Leistungsmerkmale hinzugefügt bzw. geändert werden oder entfallen. Entsprechendes gilt für andere Angaben in dieser Druckschrift.

Siemens Aktiengesellschaft

RECHNER:	PC-X, PC-MX, PC-MX2, PC-MX4
Prozessor:	80186, 8086, NS32016

BETRIEBSSYSTEM: SINIX	PROGRAMMIERSPRACHE: C

CES VERSION: 1.0B, 1.0C, 2.0	
CES BUCH 1	CES BUCH 2
Schnittstelle zur C-Programmierung: SINIX-Kommandos	Schnittstelle zum Betriebssystem: Systemaufrufe, Funktionen
- Programmerstellung cb,...	- Ein/Ausgabe printf, putchar,...
- Programmtest lint, adb,...	- Dateiverwaltung fopen, close,...
- Programmübersetzung cc, ld,...	- Speicherverwaltung malloc, calloc,...
- Programmausführung a.out	- Prozeßsteuerung signal, kill, fork,...
- Programmgenerierung lex, yacc,...	- Zeichenreihenbearbeitung strcat, strcpy,...
- Programmhaltung sccs,ar,...	- mathematische Funktionen abs, sin,...
	- Zeitfunktionen ctime, meztime,...
	- Bildschirmprogrammierung termcap . . .

Bild 1-1 Manualübersicht CES Buch1 und CES Buch 2

Vorwort

Buch 2 der CES-Dokumentation ist ein Nachschlagewerk für den C-Programmierer.

Es besteht aus einem Einführungsteil, dem eigentlichen Nachschlageteil und einem Anhang.

Der Einführungsteil

- beschreibt die formalen Konventionen von Buch 2
- erklärt wichtige Begriffe, die Ihnen geläufig sein müssen
- ordnet die Funktionen nach inhaltlichen Gesichtspunkten, was das Auffinden der richtigen Funktion erleichtert
- gibt einige Ratschläge, wie Sie Fehler in Ihrem Programm verhindern können.

Der Nachschlageteil ist ein alphabetisches Register aller Systemaufrufe, C-Funktionen und Makros, die Sie als Benutzer verwenden können, wenn Sie das C-Entwicklungssystem auf Ihrem SINIX-Rechner haben.
Diese Zusammenstellung ist neu und entspricht nicht der Einteilung in der Standard-Unix-Literatur(/3/, /4/). Sie finden dort eine Aufteilung in Systemaufrufe (üblicherweise Kapitel 2) und Funktionen (üblicherweise Kapitel 3).
Außerdem sind innerhalb der beiden Gruppen die Funktionen vorrangig inhaltlich zusammengestellt.
Wir haben uns aus folgenden Gründen für eine neue Einteilung entschieden:

- Die Aufteilung in Systemaufrufe und Funktionen ist nicht zwingend, sie kann daher in verschiedenen Implementierungen unterschiedlich ausfallen.

- Die Benutzerschnittstelle ist bei Systemaufruf und C-Funktion dieselbe, d.h die Deklarations- und Aufrufsyntax von Systemaufruf und C-Funktion unterscheiden sich nicht.

- Die alphabetische Anordnung erleichtert das Suchen nach Funktionen erheblich.

In dem Abschnitt **Typ** einer Funktionsbeschreibung ist vermerkt, ob die jeweilige Funktion ein Systemaufruf, eine C-Funktion oder ein Makro ist. Wenn die Unterscheidung nicht erforderlich ist, verwenden wir den Begriff "Funktion" als Oberbegriff.

Buch 2 können sowohl C-Anfänger als auch Profis benutzen.
Der Anfänger wird erfreut sein über die vielen Beispiele, die ohne Umschweife zum Kern der Sache kommen. Der Profi - insgeheim über das ein oder andere Beispiel schmunzelnd - wird sich hoffentlich über die übersichtliche Darstellung freuen und die vielen Hinweise schätzen lernen.

Eine Bitte an Sie

Keine erklärende Dokumentation kann perfekt sein. Eine Dokumentation lebt. Sie lebt auch von Ihren Anregungen, Ideen und Verbesserungsvorschlägen. Helfen Sie uns, indem sie uns Ihre Stolpersteine mitteilen, damit wir sie aus dem Weg räumen können.

Manualredaktion K D ST QM2
Otto-Hahn-Ring 6, 8 München 83

Inhalt

		Seite
1	**Einführung**	**1-1**
1.1	Darstellung und Formatkonventionen	1-1
1.1.1	Versionsunterschiede	1-1
1.1.2	Darstellungsformate	1-2
1.1.3	Beschreibungsstruktur	1-4
1.2	Begriffsdefinitionen	1-7
1.3	Thematische Zusammenstellung der Funktionen	1-14
1.3.1	Dateibearbeitung	1-14
1.3.1.1	Dateizugriff	1-14
1.3.1.2	Dateiverwaltung	1-15
1.3.1.3	Dateiinformation	1-15
1.3.1.4	Ein/Ausgabe	1-16
1.3.2	Prozesse	1-17
1.3.2.1	Prozeßverwaltung	1-17
1.3.2.2	Programmtest	1-17
1.3.2.3	Programmablauf-Steuerung	1-17
1.3.2.4	Zusammenwirken von Prozessen	1-18
1.3.3	Speicherverwaltung	1-18
1.3.4	Systemorganisation	1-19
1.3.5	Makros zur Zeichenbearbeitung	1-19
1.3.6	Zeichenreihen bearbeiten	1-20
1.3.7	Fehlermeldungen	1-20
1.3.8	Zeitfunktionen	1-20
1.3.9	Mathematische Funktionen	1-21
1.3.10	Konvertierung von Größen	1-22
1.3.11	Manipulation einer seriellen Schnittstelle	1-22
1.3.12	Termcap	1-22
1.4	Bibliotheken	1-23
1.5	Weise Ratschläge	1-24
1.5.1	C-Funktion statt Systemaufruf	1-24
1.5.2	Vorsicht: Fehler!	1-24
1.5.3	Ergebnistyp Zeiger	1-25
1.5.4	Ergebnisparameter Zeiger	1-25
1.5.5	Konstante oder symbolische Konstante?	1-25

Seite

2	**Nachschlageteil**		**2-1**
	abort	Prozeß abbrechen	2-2
	abs	Absolutbetrag einer ganzen Zahl	2-3
	access	Zugriffsrechte für Dateien oder Dateizeichnisse überprüfen	2-4
	acos	arcus cosinus	2-8
	alarm	Alarmuhr stellen	2-10
	asctime	Datum mit Uhrzeit in Englisch	2-12
	asin	arcus sinus	2-14
	atan	arcus tangens	2-16
	atof	Umwandlung einer Zeichenreihe in eine Gleitkommazahl	2-18
	atoi	Umwandlung einer Zeichenreihe in eine ganze Zahl	2-20
	atol	Umwandlung einer Zeichenreihe in eine ganze Zahl (Typ long)	2-22
	brk	Größe des Datensegmentes verändern	2-24
	cabs	Absolutbetrag einer komplexen Zahl	2-29
	calloc	Speicherplatz reservieren	2-31
	ceil	Aufrunden	2-33
	chdir	Aktuelles Dateiverzeichnis wechseln	2-35
	chmod	Zugriffsrechte ändern	2-37
	chown	Eigentümer und Gruppe einer Datei ändern	2-40
	chroot	Root-Dateiverzeichnis ändern	2-42
	clearerr	Lese- oder Schreib-Fehleranzeige löschen	2-43
	clos	Datei schließen	2-44
	cos	Cosinus	2-45
	cosh	Cosinus Hyperbolicus	2-47
	creat	Datei neu anlegen	2-48
	crypt	Verschlüsselung	2-52
	ctime	Datum mit Uhrzeit (MEZ) in Englisch	2-53
	dup	Zusätzliche Dateikennzahl einrichten	2-55
	dup2	Zusätzliche Dateikennzahl einrichten	2-57
	ecvt	Umwandlung in ASCII für die Ausgabe	2-59
	encrypt	Ver- oder Entschlüsseln	2-62
	endgrent	"Gruppendatei" /etc/group schließen	2-64

Seite

endpwent

"Passwortdatei" /etc/passwd schließen . . 2-65

execl Programmaufruf: 2-70

execle Programmaufruf: 2-73

execlp Programmaufruf: 2-76

execv Programmaufruf: 2-78

execve Programmaufruf: 2-81

execvp Programmaufruf: 2-83

_exit Prozeßbeendigung 2-85

exit Prozeßbeendigung 2-86

exp Exponentialfunktion 2-88

fabs Absolutbetrag einer Gleitkommazahl . . . 2-90

fclose Datei schließen 2-92

fcntl Kontrollfunktionen auf eine geöffnete Datei . 2-94

fcvt Umwandlung in ASCII Ziffern entsprechend Fortran F-Format 2-98

fdopen Dateizeiger zuweisen 2-100

feof Test auf Dateiende 2-103

ferror Test auf Dateifehler 2-105

fflush Puffer leeren 2-107

fgetc Zeichen einlesen 2-108

fgets Zeichenreihe aus einer Datei einlesen . . 2-110

fileno Dateikennzahl abfragen 2-112

floor Abrunden 2-113

fopen Datei öffnen 2-114

fork Neuen Prozeß erzeugen 2-118

fprintf Formatierte Ausgabe in eine Datei . . . 2-122

fputc Zeichen ausgeben auf Datei 2-126

fputs Zeichenreihe auf Datei ausgeben . . . 2-127

fread Blockweise einlesen 2-129

free Speicherplatz freigeben 2-132

freopen Dateizeiger neu zuweisen 2-133

frexp normierte Darstellung zur Basis 2 einer Gleitkommazahl 2-135

fscanf Formatiertes Einlesen von einer Datei . . 2-137

fseek Lese/Schreibzeiger positionieren . . . 2-143

fstat Dateiinformationen ausgeben 2-145

ftell Aktuelle Lese/Schreibposition abfragen . 2-148

ftime Aktuelle Zeit 2-150

fwrite Blockweise ausgeben 2-152

gctime Datum mit Uhrzeit (MEZ) in Deutsch . . 2-154

gcvt Umwandlung in ASCII für die Ausgabe . 2-156

		Seite
getc	Zeichen von Datei einlesen	2-158
getchar	Ein Zeichen von Standardeingabe lesen .	2-160
getegid	Effektive Gruppennummer abfragen . .	2-161
getenv	Umgebungsvariable abfragen	2-162
geteuid	Effektive Benutzernummer abfragen . .	2-164
getgid	Reale Gruppennummer abfragen . . .	2-165
getgrent	Nächste Zeile von /etc/group ausgeben .	2-166
getgrgid	Gruppennummer in /etc/group suchen .	2-168
getgrnam	Gruppennamen in /etc/group suchen . .	2-170
getlogin	Login-Namen abfragen	2-172
getpass	Passwort einlesen	2-174
getpgrp	Prozeßgruppe abfragen	2-176
getpid	Prozeßnummer abfragen	2-177
getppid	Prozeßnummer des Vaters abfragen . . .	2-178
getpw	Eintrag in /etc/passwd suchen	2-179
getpwent	Nächste Zeile von /etc/passwd ausgeben .	2-181
getpwnam	Benutzernamen in /etc/passwd suchen .	2-183
getpwuid	Benutzernummer in /etc/passwd suchen .	2-185
gets	Zeichenreihe von Standardeingabe einlesen	2-187
getuid	Reale Benutzernummer abfragen . . .	2-189
getqw	Wort von Datei einlesen	2-190
gmtime	Aktuelle Zeit (GMT) als Struktur . . .	2-192
gtty	Bildschirmeigenschaften abprüfen . . .	2-194
index	Erstes Auftreten eines Zeichens in einer Zeichenreihe	2-199
ioctl	Geräteschnittstelle	2-200
isalnum	Buchstabe oder Ziffer?	2-202
isalpha	Buchstabe?	2-204
asiscii	ASCII Zeichen?	2-206
isatty	Dateikennzahl mit Gerätedatei verbunden?	2-207
iscntrl	Kontrollzeichen?	2-208

Seite

isdigit	Ziffer?	2-210
islower	Kleinbuchstabe?	2-212
isprint	Abdruckbares Zeichen?	2-214
ispunct	Sonderzeichen?	2-216
isspace	Zwischenraum?	2-218
isupper	Großbuchstabe?	2-220
isxdigit		
	Hexadezimale Ziffer?	2-222
j0, j1, jn		
	Besselfunktionen der ersten Art	2-224
kill	Signal an Prozesse schicken	2-225
l3tol	Umwandlung in long Integer	2-228
ldexp	Wert im Zweiersystem berechnen . . .	2-229
link	Verweis auf eine Datei einrichten . . .	2-231
localtime		
	Datum und Uhrzeit (MEZ) berechnen . .	2-233
locking	Teilbereiche einer Datei sperren oder freigeben	2-235
log	Natürlicher Logarithmus	2-240
log10	Logarithmus zur Basis 10	2-241
longjmp	Nicht lokaler Sprung	2-242
lseek	Lese/Schreibzeiger positionieren . . .	2-245
ltol3	Umwandlung in 3 Bytes lange Integer . .	2-248
malloc	Speicherplatz reservieren	2-249
meztime	Datum mit Uhrzeit in Deutsch	2-251
mknod	Gerätedatei oder Dateiverzeichnis einrichten	2-253
mktemp	Eindeutiger Dateiname	2-257
modf	Aufspalten einer Zahl in ihren ganzzahligen und gebrochenen Teil	2-259
monitor	Statistische Auswertung einer Programmausführung	2-261
mount	Dateisystem einhängen	2-263
nice	Priorität eines Prozesses ändern	2-266
nlist	Werte aus der Symboltabelle suchen . .	2-268
open	Datei öffnen (elementar)	2-271
pause	Prozeß anhalten bis Signal eintrifft . . .	2-276
pclose	Dateiverbindung zu einem Kommando schließen	2-278
perror	Fehlermeldung ausgeben	2-280
pipe	Pipe einrichten	2-283

		Seite
popen	Kommando aufrufen und Pipe einrichten	2-288
pow	allgemeine Exponentialfunktion	2-291
printf	Formatierte Ausgabe auf Standard-ausgabe	2-293
profil	Histogramm der Prozeßausführung	2-297
ptrace	Prozeßüberwachung	2-298
putc	Zeichen ausgeben	2-304
putchar	Zeichen auf Standardausgabe schreiben	2-306
puts	Zeichenreihe auf Standardausgabe ausgeben	2-307
putw	Wortweise in eine Datei schreiben	2-309
qsort	Quicksort	2-311
rand	Zufallszahlgenerator	2-313
rdchk	Auf Eingabedaten abprüfen	2-314
read	Elementare Einleseoperation	2-316
realloc	Speicherplatz verändern	2-319
rewind	Auf Dateianfang positionieren	2-320
rindex	Letztes Auftreten eines Zeichens in einer Zeichenreihe	2-322
sbrk	Größe des Datensegmentes verändern	2-323
scanf	Formatiertes Einlesen von Standard-eingabe	2-325
setbuf	Ein/Ausgabepuffer zuordnen	2-331
setbuffer	Ein/Ausgabepuffer zuordnen	2-334
setgid	Gruppennummer des Prozesses setzen	2-336
setgrent	Auf den Dateianfang von /etc/group positionieren	2-337
setjmp	"Marke" für nicht-lokale Sprünge setzen	2-339
setkey	Einstellen des "DES-Algorithmus"	2-340
setlinebuf	Ein/Ausgabepuffer für zeilenweise Pufferung zuordnen	2-341
setpgrp	Prozeßgruppe definieren	2-342
setpwent	Auf den Dateianfang von /etc/passwd positionieren	2-343
setuid	Benutzernummer des Prozesses setzen	2-345
shutdn	CPU anhalten	2-346
signal	Signalbearbeitung	2-347

		Seite
sin	Sinus	2-353
sinh	Sinus Hyperbolicus	2-355
sleep	Prozeß für festgesetzte Zeitspanne anhalten	2-357
sprintf	Formatierte Ausgabe in eine Zeichenreihe	2-359
sqrt	Quadratwurzel	2-363
srand	Initialisierung des Zufallszahlgenerators .	2-365
sscanf	Formatiertes Einlesen aus einer Zeichenreihe	2-366
stat	Dateiinformationen ausgeben	2-371
stime	Systemuhr stellen	2-376
strcat	Konkatenation von Zeichenreihen . . .	2-377
strcmp	Vergleich von zwei Zeichenreihen . . .	2-378
strcpy	Zeichenreihe kopieren	2-380
strdup	Temporäre Kopie einer Zeichenreihe . .	2-382
strlen	Länge einer Zeichenreihe	2-383
strncat	Konkatenation von Zeichenreihen . . .	2-384
strncmp	Vergleich von zwei Zeichenreihen . . .	2-386
strncpy	Zeichenreihe kopieren	2-388
stty	Eigenschaften einer seriellen Schnittstelle festlegen	2-390
swab	Bytes vertauschen	2-393
sync	Superblock aktualisieren	2-394
system	Shell Kommando ausführen	2-395
tan	Tangens	2-397
tanh	Tangens Hyperbolicus	2-399
tell	Aktuelle Position des Lese/Schreib-zeigers	2-401
termcap	Programmierhilfe für bildschirmorientierte Anwendungen	2-402
Mögliche	Felder für einen termcap-Eintrag	2-407
tgetent	Eintrag aus /etc/termcap suchen . . .	2-411
tgetflag	Boolsches Feld im Eintrag suchen . . .	2-413
tgetnum	Numerisches Feld im Eintrag suchen . .	2-415
tgetstr	Steuerzeichenfolge im Eintrag suchen . .	2-417
tgoto	Steuerzeichenfolge mit aktuellen Werten versorgen	2-420
tputs	Ausgabe von Steuerzeichenfolgen . . .	2-422
time	Aktuelle Zeitangabe	2-425

Seite

			Seite
	times	Laufzeit eines Prozesses	2-427
	times	Laufzeit eines Prozesses	2-429
	timezone	Name der Zeitzone	2-431
	toascii	Zeichen in ASCII-Zeichen umwandeln .	2-433
	tolower	Großbuchstabe in Kleinbuchstabe umwandeln	2-434
	toupper	Kleinbuchstabe in Großbuchstabe umwandeln	2-436
	ttyname	Name einer Gerätedatei	2-437
	ttyslot	Eintrag in /etc/ttys bestimmen	2-439
	ulimit	Kontrollfunktionen auf einen Benutzerprozeß	2-440
	umask	Schutzbitmaske setzen	2-442
	umount	Dateisystem abhängen	2-444
	uname	Name des aktuellen SINIX Systems . .	2-446
	ungetc	Zeichen zurückstellen	2-448
	unlink	Verweis auf eine Datei löschen	2-450
	utime	Zeiten einer Datei neu setzen	2-452
	wait	Auf Prozeßbeendigung warten	2-454
	write	Elementare Schreiboperation	2-457
	y0, y1, yn	Besselfunktionen der zweiten Art . . .	2-460
A	**Anhang**		**A-1**
	Version 1.0B		A-2
	Version 1.0C		A-8
	Version 2.0		A-13
	ASCII-Tabelle		A-17

Fachwörter deutsch-englisch
Fachwörter englisch-deutsch
Literatur
Bestellung

1 Einführung

1.1 Darstellung und Formatkonventionen

Dieses Kapitel erklärt die formalen Konventionen von Buch 2.

1.1.1 Versionsunterschiede

Das C-Entwicklungssystem wird in drei Versionen (1.0B, 1.0C und 2.0) ausgeliefert (siehe Bild 1-1).
Version 2.0 ist umfangreicher als Version 1.0C, Version 1.0C ist umfangreicher als Version 1.0B.
In diesem Buch sind alle drei Versionen beschrieben.
Da die Versionen aufwärtskompatibel sind, haben wird folgende Kennzeichnung der Unterschiede gewählt:

- Teile, die erst ab Version 1.0C gelten, sind mit hellgrauem Raster unterlegt,
- Teile, die erst ab Version 2.0 gelten, sind mit dunkelgrauem Raster unterlegt.

Für jede Version gibt es zusätzlich einen Anhang (siehe Anhang), in dem die Besonderheiten der jeweiligen Version, soweit sie Buch 2 betreffen, zusammengestellt sind.

Bevor Sie mit CES arbeiten, müssen Sie feststellen, welche Version auf Ihrem Rechner installiert ist.

1.1.2 Darstellungsformate

Funktionsdefinition (grün unterlegt)

Fettdruck: alle konstanten Teile, z.B **int fstat (...)**
Normaldruck: alle Variablen, z.B. **int fstat(**dk,dinf**)**

Beschreibungsteil

Kursiv: alle Variablen, z.B. die Datei *dname*

Fettdruck: wichtige Hervorhebungen, z.B. **reale** Benutzernummer

"..." : zitierte Begriffe, z.B. der "break"

'...' : (konstante) Zeichen, die so anzugeben sind, z.B. '='

{GROß}: symbolische Konstanten, die implementierungsabhängig sind, z.B. {PDAT_MAX}

GROß: sonstige symbolische Konstanten, z.B. EOF

Normaldruck: sonst, insbesondere werden im Beschreibungsteil vordefinierte Namen (für Dateien, Funktionen, Strukturen, Datentypen, Varaiablen) nicht hervorgehoben.

[...] : wahlweise Angabe,
z.B. **int open(**d_name,modus [, zugriff]**)** heißt, daß der Parameter *zugriff* nicht immer angegeben wird.

x... : das vorangehende Argument *x* kann wiederholt werden z.B. ziffer... steht für eine Folge von Ziffern

Abkürzungen

<d_name.h>
<sys/d_name.h>

<d_name.h> <sys/d_name.h>	Abkürzende Schreibweise für den Pfadnamen /usr/include/d_name.h bzw. /usr/include/sys/d_name.h
bzw.	beziehungsweise
d.h.	das heißt
i.a.	im allgemeinen
o.g.	oben genannt
s.	siehe
s.o.	siehe oben
u.a.	unter anderem
usw.	und so weiter
u.U.	unter Umständen
vgl.	vergleiche
z.B.	zum Beispiel

1.1.3 Beschreibungsstruktur

Alle Funktionen sind in einer einheitlichen Struktur beschrieben. Die einzelnen Abschnitte dieser Struktur sind nachfolgend erklärt. Jeder Abschnitt hat einen Namen (fettgedruckt). Es gibt folgende Abschnitte:

- Typ
- Parameter
- Ergebnis
- Fehlermeldung
- Achtung
- Hinweis
- Beispiel
- Dateien
- Externe Größen

In einer Funktionsbeschreibung können Abschnitte fehlen, falls sie für die Funktion keine Bedeutung haben.

Auf den nächsten Seiten ist die Beschreibungsstruktur vollständig angegeben und für jeden Abschnitt allgemein aufgelistet, was dort stehen kann.

Funktionsname

Deutsche Kurzbezeichnung

include-Dateien, die angegeben werden müssen
externe Größen, die die Funktion verwendet

Funktionskopf: **<typ> funkt(**par1, par2,...**)**
Parameterdeklarationen;

Text, der die Arbeitsweise der Funktion erklärt.

Typ

Systemaufruf oder C-Funktion oder Makro
Der Zusatz '(s)' besagt, daß die Funktion zur Standardein/ausgabe Bibliothek gehört, z.B. bei fopen:
C-Funktion (s)

Parameter

Es gibt zwei Arten von Parametern: Eingabe- und Ergebnisparameter. Die Ergebnisparameter sind durch ein vorangestelltes '←' gekennzeichnet:

<typ> par	Eingabeparameter
konst	Beschreibung, was ein Aufruf mit aktuellem Wert konst bewirkt
← <typ> par	Ergebnisparameter
konst	Beschreibung, was ein Aufruf mit aktuellem Wert konst bewirkt.

Ergebnis

Mögliche Funktionsergebnisse und ihre Bedeutung

Fehlermeldung

Für jeden Systemaufruf sind Fehlercodes eines fehlerhaften Aufrufes wie folgt aufgelistet:

FEHLERCODE : Kurze Standardfehlermeldung

Es gibt für jeden Fehler, der mit vorangestelltem '-' unter **Ergebnis** erfaßt ist, einen entsprechenden Fehlercode. Fehler und Fehlercode entsprechen sich in der Reihenfolge der Aufschreibung.

Achtung

Hinweise, die Sie auf alle Fälle berücksichtigen sollten!

Hinweis

- Begriffserklärungen
- Anwendungstips
- Zusammenhang mit anderen Funktionen

Beispiel

Kleines Beispiel, das eine Anwendung der beschriebenen Funktion zeigt.

Dateien

Auflistung der Dateien, die die Funktion verwendet mit vollem Pfadnamen und kurzer Inhaltsangabe.

Externe Größen

Beschreibung von externen Größen, die für die Funktion eine Bedeutung haben.

> > > > verwandte Funktionen

1.2 Begriffsdefinitionen

Dieses Kapitel erklärt Begriffe, die Ihnen häufig in den Funktionsbeschreibungen begegnen.

Dateikennzahl

Eine Dateikennzahl ist eine positive ganze Zahl aus dem Intervall [0, {PDAT_MAX}], die dazu dient, die Verbindung zu einer Datei herzustellen. Elementare Zugriffsoperationen (read, write,...) benutzen die Dateikennzahl als Dateiargument.

Dateiname

Ein Dateiname ist eine Zeichenreihe aus 1 bis {NAME_MAX} Zeichen. Außer '\0' und '/' ist jedes ASCII-Zeichen erlaubt. Allerdings ist es ratsam, nur sinnvolle Zeichen zu verwenden d.h., keine nicht druckbaren Zeichen, keine Sonderzeichen der Shell usw.

Dateistatus

Unter Dateistatus versteht man die Verwaltungsinformationen über eine bestehende Datei. Dazu gehören die Art der Datei, die Zugriffsrechte, die Änderungszeiten, der Eigentümer, usw. Den Dateistatus können Sie mit stat oder fstat abfragen.

Dateiverzeichnis

Dateiverzeichnisse sind die Knotenpunkte in dem baumartigen Dateisystem. Sie enthalten Verweise (Indexeinträge) auf Dateien und andere Dateiverzeichnisse. Es gibt zwei ausgezeichnete Einträge:

.	der Verweis auf das Dateiverzeichnis selbst
..	der Verweis auf das übergeordnete Dateiverzeichnis

(siehe SINIX Buch1, /2/).

Jedem Prozeß ist ein **aktuelles Dateiverzeichnis** und ein **Root-Dateiverzeichnis** zugeordnet. Ein Prozeß kann (mit Einschränkung) beide verändern (siehe chdir, chroot).

Dateizeiger

Der Dateizeiger ist neben Dateiname und Dateikennzahl eine weitere Möglichkeit, eine Datei anzusprechen. Der Dateizeiger verweist auf die FILE-Struktur, die der Datei für gepufferte Ein/Ausgabe zugeordnet wurde. Standardein/ausgabefunktionen verwenden den Dateizeiger, um die Verbindung zur Datei herzustellen.

Effektive Benutzer- und Gruppennummer

Zusätzlich zu der realen Benutzer- und Gruppennummer erhält jeder Prozeß eine effektive Benutzer- bzw. Gruppennummer. Die effektiven Nummern sind dieselben wie die realen außer, wenn folgender Fall eintritt:
Im Indexeintrag der Programmdatei ist das s-Bit für Eigentümer bzw. Gruppe gesetzt.
Dann wird die Eigentümer- bzw. Gruppenkennung dieser Datei zur effektiven Benutzer- bzw. Gruppennummer des Prozesses.

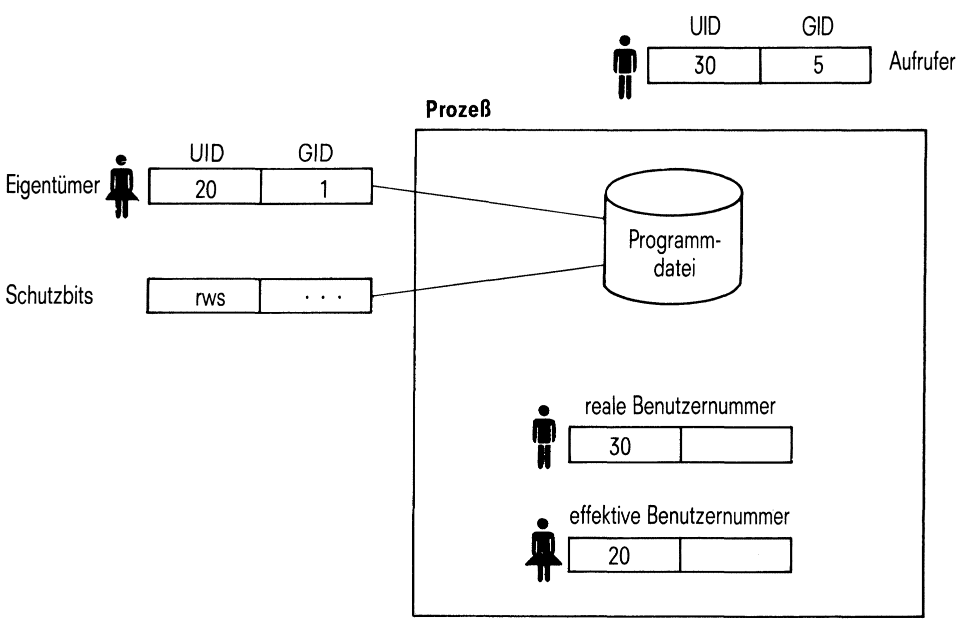


Bild 1-2 s-Bit Einfluß auf die effektive Benutzernummer

Die effektive Benutzer- und Gruppennummer vererbt ein Vaterprozeß an seine Söhne (siehe fork).
Die effektive Benutzer- und Gruppennummer bestimmen die Zugriffsrechte eines Prozesses (siehe Zugriffsrechte). Sie können daher über den s-Bit-Mechanismus den kontrollierten Zugriff auf Dateien realisieren (siehe auch SINIX Buch 1, /2/).
Mit setuid bzw. setgid können Sie die effektive Benutzer- bzw. Gruppennummer neu setzen.

Fehlermeldungen

Für jeden Systemaufruf sind die Fehlercodes aufgezählt, die ein fehlerhafter Aufruf auslösen kann. Ein Fehlercode wird in der externen Variablen extern int errno abgelegt. Diese Variable können Sie in Ihrem Programm abfragen (siehe perror). Endet ein Systemaufruf mit Fehler (meist Ergebnis -1), können mehrere Fehlerursachen vorliegen. Wir haben nicht ausgetestet, welche Fehlerursachen vorrangig abgeprüft werden. Folglich ist die Reihenfolge der Aufschreibung bedeutungslos.

FILE-Struktur

Einer Datei, die mit fopen, fdopen oder freopen geöffnet wird, ist automatisch ab diesem Zeitpunkt eine FILE-Struktur zugeordnet. Die Struktur ist in <stdio.h> definiert. Die Struktur enthält folgende Information über die Datei:

- Zeiger auf den Ein/Ausgabe-Puffer, der bei den Funktionen aus der Standardein/ausgabe-Bibliothek automatisch zugeordnet wird.
- Position des Lese/Schreibzeigers
- Länge der Datei

Der durch fopen, fdopen oder freopen zugeordnete Dateizeiger verweist auf diese FILE-Struktur.

Gerätedatei

Es gibt zeichen- und blockorientierte Geräte.
Zeichenorientierte Geräte sind z.B. Bildschirm und Drucker.
Blockorientierte Geräte sind z.B. Festplatte und Diskette.
Beide Gerätearten werden über Gerätedateien angesprochen.
Eine Gerätedatei hat eine **major-Nummer**, die den Typ des Gerätes angibt und eine **minor-Nummer**, die innerhalb eines Typs die Geräte unterscheidet. Beide Nummern sind positive ganze Zahlen.

Pfadnamen

Pfadnamen (Dateinamen) in einem C-Programm haben dieselben Konventionen wie Shell-Pfadnamen.
Denken Sie daran, daß bei einem Pfadnamen, der mit '/' beginnt, der Pfad im Root-Dateiverzeichnis startet, ansonsten im aktuellen Dateiverzeichnis.

Prozeß

Ein Prozeß ist ein Programm, das gerade ausgeführt wird. Ein Prozeß besteht aus dem ablauffähigen Programmcode und einer Reihe von Prozeß-spezifischen Verwaltungsdaten, die zum Ablauf des Programmes erforderlich sind. Jeder Prozeß wird in der Prozeßtabelle des Systems geführt.

Prozeßgruppe

Jeder aktive Prozeß ist Mitglied einer Prozeßgruppe. Die Prozeßgruppe ist gekennzeichnet durch eine kleine positive ganze Zahl, die **Prozeßgruppennummer**. Sie ist die Prozeßnummer des **Gruppenchefs**. Gruppenchef ist der Prozeß, der die Gruppe gebildet hat. Ab Version 1.0C kann jeder Prozeß mittels setpgrp eine neue Gruppe aufmachen. Die Prozeßgruppennummer wird bei exec und fork vererbt.

Prozeßnummer

Jeder Prozeß hat eine vom System zugeordnete Prozeßnummer. Sie ist eine positive ganze Zahl aus dem Intervall [0, {BPROZ_MAX}].
Die Prozeßnummern 0 und 1 sind für Systemprozesse reserviert.

Reale Benutzer- und Gruppennummer

Jedes SINIX System identifiziert einen Benutzer an Hand seiner Benutzer- und Gruppenkennung.
Ein Prozeß erhält als reale Benutzer- bzw Gruppennummer die Benutzer- bzw. Gruppenkennung desjenigen, der den Prozeß gestartet hat. Mit setuid bzw. setgid können Sie die reale Benutzer- bzw. Gruppennummer neu setzen.

Systemverwalter

Ein Prozeß gilt als Systemverwalter-Prozeß, wenn seine effektive Benutzernummer 0 ist.

Umgebungsvariable

Mit Beginn der Programmausführung steht dem Prozeß ein externes Feld von Zeichenreihen zur Verfügung, die Variablen definieren. Das Minimum an Variablen, die bereitgestellt werden, ist:

Variable	Wert der Variablen
HOME	Gesamt-Pfadname des Home-Dateiverzeichnisses
PATH	Eine Folge von Pfadnamen, die durch ':' getrennt sind. Diese Pfadnamen legen die Suchpfade fest, mit denen die Shell nach Dateien sucht.
TERM	Typ der angeschlossenen Datensichtstation

Vaterprozeß

Mit dem Systemaufruf fork könen Sie einen neuen Prozeß erzeugen.
Der aufrufende Prozeß ist der Vaterprozeß, der neue Prozeß ist der **Sohnprozeß**. Die **Vaterprozeßnummer** ist die Prozeßnumer des Vaterprozesses. Ab Version 1.0C können Sie diese Nummer mit getppid abfragen.

Zeichenreihen

Der Begriff "Zeichenreihe" bedeutet C-Zeichenreihe:

konstante Zeichenreihe :
"..." , z.B. "Sonnenschein"

variable Zeichenreihe :
Zeiger auf einen Vektor von Zeichen (char *p), dessen letztes Element das Nullbyte (^{6}0) ist.

Dateinamen in einem C-Programm sind C-Zeichenreihen. Wenn eine Funktion einen Dateinamen als Argument verlangt, können Sie entweder einen Namen in " " angeben oder einen Zeiger, der auf das erste Zeichen des Dateinamens zeigt. Das letzte Element des Vektors muß dann das Nullbyte sein.

Zugriffsrechte auf eine Datei

Ein Prozeß erhält ein Zugriffsrecht auf eine Datei (lesen, schreiben, ausführen/durchsuchen), wenn einer oder mehrere der folgenden Punkte erfüllt sind:

- Die effektive Benutzernummer des Prozesses ist die Kennung des Systemverwalters.
- Die effektive Benutzernummer des Prozesses stimmt mit der Kennung des Dateieigentümers überein und der Dateieigentümer hat das entsprechende Zugriffsrecht.
- Die effektive Benutzernummer des Prozesses stimmt nicht mit der Kennung des Dateieigentümers überein und die effektive Gruppennummer des Prozesses stimmt mit der Gruppenkennung der Datei überein. Die Gruppe muß dann das entsprechende Zugriffsrecht auf die Datei haben.
- Effektive Benutzernummer und effektive Gruppennummer stimmen nicht mit Eigentümer- bzw. Gruppenkennung der Datei überein. Dann muß das entsprechende Zugriffsrecht für Andere gewährleistet sein.

1.3 Thematische Zusammenstellung der Funktionen

In diesem Kapitel finden Sie eine Zusammenstellung der Funktionen nach inhaltlichen Gesichtspunkten. Jede Funktion kommt genau einmal vor. In der Spalte "Typ" bedeutet:

S : Systemaufruf
F : C-Funktion
F(s) : C-Funktion aus der Standardein/ausgabe Bibliothek
M : Makro
M(s) : Makro aus include-Datei <stdio.h>

Die mit '(s)' gekennzeichneten Funktionen aus der Standardein/ausgabe Bibliothek sprechen eine Datei über ihren Dateizeiger an. In Programmen, die diese Funktionen verwenden, sollte die Datei <stdio.h> am Anfang eingefügt werden.

1.3.1 Dateibearbeitung

1.3.1.1 Dateizugriff

Typ	Name	Kurzbeschreibung
S	close	Datei schließen
S	dup	zusätzliche Dateikennzahl einrichten
S	dup2	bestimmte Dateikennzahl zusätzlich einrichten
F(s)	fclose	Datei schließen
F(s)	fdopen	einer Dateikennzahl einen Dateizeiger zuordnen
F(s)	fflush	Dateipuffer leeren
F(s)	fopen	Datei öffnen
F(s)	freopen	Dateizeiger neu zuweisen
F(s)	fseek	Lese/Schreibzeiger positionieren
F(s)	ftell	aktuelle Position des Lese/Schreibzeigers abfragen
S	locking	Teilbereiche einer Datei sperren oder freigeben
S	lseek	Lese/Schreibzeiger positionieren
S	open	Datei öffnen
S	rdchk	prüfen, ob Datei gelesen werden kann ohne zu blockieren
F(s)	rewind	auf Dateianfang positionieren
S	tell	aktuelle Position des Lese/Schreibzeigers abfragen

1.3.1.2 Dateiverwaltung

Typ	Name	Kurzbeschreibung
S	chdir	aktuelles Dateiverzeichnis wechseln
S	chmod	Zugriffsrechte ändern
S	chown	Dateieigentümer ändern
S	chroot	Root-Dateiverzeichnis wechseln
F(s)	clearerr	Lese/Schreibfehler-Anzeige löschen
S	creat	neue Datei erstellen
S	fcntl	Kontrollfunktionen auf eine geöffnete Datei
S	link	Verweis auf eine Datei einrichten
S	mknod	neues Dateiverzeichnis oder Gerätedatei anlegen
F	mktemp	eindeutigen Dateinamen erzeugen
S	mount	Dateisystem einhängen
S	umask	Prozeßmaske setzen
S	umount	Dateisystem aushängen
S	unlink	Verweis löschen
S	utime	Dateizeiten (letzter Zugriff, letzte Änderung) setzen

1.3.1.3 Dateiinformation

Typ	Name	Kurzbeschreibung
S	access	Zugriffsrechte überprüfen
M(s)	feof	auf Dateiende abprüfen
M(s)	ferror	auf Dateifehler abprüfen
M(s)	fileno	Dateikennzahl abfragen
S	fstat	Dateistatus abfragen
S	stat	Dateistatus abfragen

1.3.1.4 Ein/Ausgabe

Typ	Name	Kurzbeschreibung
F(s)	fgetc	ein Zeichen einlesen
F(s)	fgets	Zeichenreihe einlesen
F(s)	fprintf	formatierte Ausgabe auf Datei
F(s)	fputc	ein Zeichen in eine Datei schreiben
F(s)	fputs	Zeichenreihe in eine Datei schreiben
F(s)	fread	blockweise von Datei einlesen
F(s)	fscanf	formatierte Eingabe von Datei
F(s)	fwrite	blockweise auf Datei ausgeben
M(s)	getc	ein Zeichen einlesen
M(s)	getchar	ein Zeichen von Standardeingabe einlesen
F(s)	gets	Zeichenreihe von Standardeingabe einlesen
F(s)	getw	Maschinenwort-weise einlesen
F(s)	putc	ein Zeichen ausgeben
M(s)	putchar	ein Zeichen auf Standardausgabe ausgeben
F(s)	printf	formatierte Ausgabe auf Standardausgabe
M(s)	puts	Zeichenreihe auf Standardausgabe ausgeben
F(s)	putw	Maschinenwort-weise ausgeben
S	read	aus einer Datei lesen
F(s)	scanf	formartierte Eingabe von Standardeingabe
F(s)	setbuf	Ein/Ausgabepuffer zuordnen
F(s)	setbuffer	Ein/Ausgabepuffer zuordnen
F(s)	setlinebuf	Zeilenweise Pufferung
F(s)	sprintf	formatierte Ausgabe in eine Zeichenreihe
F(s)	sscanf	formatierte Eingabe von einer Zeichenreihe
F(s)	ungetc	ein Zeichen in den Puffer zurückstellen
S	write	in eine Datei schreiben

1.3.2 Prozesse

1.3.2.1 Prozeßverwaltung

Typ	Name	Kurzbeschreibung
S	alarm	Alarmuhr stellen
S	getegid	effektive Gruppennummer abfragen
S	geteuid	effektive Benutzernummer abfragen
F	getenv	Umgebungsvariable abfragen
S	getgid	reale Gruppennummer abfragen
S	getpgrp	Prozeßgruppe abfragen
S	getpid	Prozeßnummer abfragen
S	getppid	Vaterprozeßnummer abfragen
S	getuid	reale Benutzernummer abfragen
S	kill	Signale schicken
S	pause	Prozeß auf Eis legen
S	setpgrp	Prozeßgruppe eröffnen
S	setgid	reale und effektive Gruppennummer setzen
S	setuid	reale und effektive Benutzernummer setzen
S	signal	Signalbearbeitung
F	sleep	Prozeß für festgesetzte Zeitspanne anhalten
S	times	Prozeßzeiten abfragen
S	ulimit	Kontrollfunktionen auf einen Benutzerprozeß

1.3.2.2 Programmtest

Typ	Name	Kurzbeschreibung
F	nlist	Werte aus der Symboltabelle suchen
S	ptrace	Programmüberwachung

1.3.2.3 Programmablauf-Steuerung

Typ	Name	Kurzbeschreibung
S	longjmp	Nicht lokaler Sprung
S	nice	Prozeßpriorität einstellen
S	profil	Zeitauswertung
F	monitor	Auswertung der Programmausführung
S	setjmp	Programmzustand retten

1.3.2.4 Zusammenwirken von Prozessen

Typ	Name	Kurzbeschreibung
S	abort	Prozeß abbrechen
S	execl	Programmaufruf
S	execle	Programmaufruf
S	execlp	Programmaufruf
S	execv	Programmaufruf
S	execve	Programmaufruf
S	execvp	Programmaufruf
S	_exit	Prozeß beenden
F	exit	Prozeß beenden
S	fork	Prozeß erzeugen
F	system	Shell-Kommando aufrufen
S	wait	auf Prozeßbeendigung warten

Pipe

Typ	Name	Kurzbeschreibung
F(s)	pclose	Pipe zu einem Kommando schließen
S	pipe	Pipe einrichten
F(s)	popen	Pipe zu einem Kommando einrichten

1.3.3 Speicherverwaltung

Typ	Name	Kurzbeschreibung
S	brk	den "break" neu setzen
F	calloc	Speicherplatz für ein Feld reservieren
F	free	Speicherplatz freigeben
F	malloc	Speicherplatz reservieren
F	realloc	Speicherplatz verändern
S	sbrk	den "break" verändern

1.3.4 Systemorganisation

Typ	Name	Kurzbeschreibung
F	crypt	Passwort verschlüsseln
F	encrypt	ver/entschlüsseln
F	endgrent	Gruppendatei schließen
F	endpwent	Passwortdatei schließen
F	getgrent	Eintrag aus der Gruppendatei
F	getgrgid	Gruppennummer in der Gruppendatei suchen
F	getgrnam	Gruppennamen in der Gruppendatei suchen
F	getlogin	Login-Namen abfragen
F	getpass	Passwort einlesen
F	getpw	Eintrag aus der Passwortdatei suchen
F	getpwent	nächste Zeile aus der Passwortdatei
F	getpwnam	Namen in der Passwortdatei suchen
F	getpwuid	Benutzernummer in der Passwortdatei suchen
F	setgrent	auf den Anfang der Gruppendatei positionieren
F	setpwent	auf den Anfang der Passwortdatei positionieren
F	setkey	DES-Algorithmus einstellen
S	shutdn	System anhalten
S	sync	Systempuffer leeren
S	uname	Name des aktuellen SINIX Systems abfragen

1.3.5 Makros zur Zeichenbearbeitung

Typ	Name	Kurzbeschreibung
M	isalnum	alphanumerisches Zeichen?
M	isalpha	Buchstabe?
M	isascii	ASCII-Zeichen?
M	iscntrl	Kontrollzeichen?
M	isdigit	Ziffer?
M	islower	Kleinbuchstabe?
M	isprint	abdruckbares Zeichen?
M	ispunct	Sonderzeichen?
M	isspace	Zwischenraum?
M	isupper	Großbuchstabe?
M	isxdigit	hexadezimales Zeichen?
M	toascii	Umwandlung in ASCII-Zeichen
M	tolower	Umwandlung in Kleinbuchstaben
M	toupper	Umwandlung in Großbuchstaben

1.3.6 Zeichenreihen bearbeiten

Typ	Name	Kurzbeschreibung
F	index	erstes Vorkommen eines Zeichens in einer Zeichenreihe
F	rindex	letztes Vorkommen eines Zeichens in einer Zeichenreihe
F	strcat	Verkettung von zwei Zeichenreihen
F	strcmp	Vergleich zweier Zeichenreihen
F	strcpy	Zeichenreihe kopieren
F	strdup	Temporäre Kopie einer Zeichenreihe
F	strlen	Länge einer Zeichenreihe abfragen
F	strncat	Verkettung bis zur Länge n
F	strncmp	Vergleich bis zur Länge n
F	strncpy	Kopie bis zur Länge n

1.3.7 Fehlermeldungen

Typ	Name	Kurzbeschreibung
F	perror	Standardfehlermeldung ausgeben

1.3.8 Zeitfunktionen

Typ	Name	Kurzbeschreibung
F	asctime	Datum mit Uhrzeit in Englisch
F	ctime	Datum mit Uhrzeit(MEZ) in Englisch
S	ftime	Aktuelle Zeit(GMT) als Struktur
F	gctime	Datum mit Uhrzeit(MEZ) in Deutsch
F	gmtime	aktuelle Zeit(GMT) als Struktur
L	localtime	aktuelle Ortszeit(MEZ) als Struktur
F	meztime	Datum mit Uhrzeit in Deutsch
S	stime	Systemuhr stellen
S	time	Aktuelle Zeit(GMT) in Sekunden
F	timezone	Name der Zeitzone

1.3.9 Mathematische Funktionen

Typ	Name	Kurzbeschreibung
F	abs	Absolutbetrag
F	acos	Arcus Cosinus
F	asin	Arcus Sinus
F	atan	Arcus Tangens
F	cabs	Absolutbetrag einer komplexen Zahl
F	ceil	ganzzahlig aufrunden
F	cos	Cosinus
F	cosh	Cosinus Hyperbolicus
F	exp	Exponentialfunktion
F	fabs	Absolutbetrag einer Gleitkommazahl
F	floor	ganzzahlig abrunden
F	frexp	normierte Darstellung zur Basis 2
F	hypot	Euklidischer Abstand
F	j1	Besselfunktion
F	jn	Besselfunktion
F	jo	Besselfunktion
F	ldexp	Wert im Zweiersystem berechnen
F	log	natürlicher Logarithmus
F	log10	Logarithmus zur Basis 10
F	modf	Aufspalten in Ganzteil und Bruchteil
F	pow	allgemeine Exponentialfunktion
F	qsort	Quicksort
F	rand	Zufallsgenerator
F	sin	Sinus
F	sinh	Sinus Hyperbolicus
F	sqrt	Quadratwurzel
F	srand	Zufallsgenerator initialisieren
F	tan	Tangens
F	tanh	Tangens Hyperbolicus
F	y1	Besselfunktion
F	yn	Besselfunktion
F	yo	Besselfunktion

1.3.10 Konvertierung von Größen

Typ	Name	Kurzbeschreibung
F	atof	Zeichenreihe in Gleitkommazahl
F	atoi	Zeichenreihe in integer
F	atol	Zeichenreihe in long integer
F	ecvt	Gleitkommawert in ASCII-Zeichenreihe
F	fcvt	Gleitkommawert in ASCII-Zeichenreihe
F	gcvt	Gleitkommawert in ASCII-Zeichenreihe
F	l3tol	3-Byte-integer in long integer
F	ltol3	long integer in 3-Byte-integer
F	swab	Kopieren mit vertauschen jeweils benachbarter Bytes

1.3.11 Manipulation einer seriellen Schnittstelle

Typ	Name	Kurzbeschreibung
S	gtty	Eigenschaften der seriellen Schnittstelle abfragen
S	ioctl	Kontrollfunktionen auf eine serielle Schnittstelle
F	isatty	serielle Schnittstelle?
S	stty	Eigenschaften einer seriellen Schnittstelle setzen
F	ttyname	Name einer Gerätedatei abfragen
F	ttyslot	Eintrag in /etc/ttys suchen

1.3.12 Termcap

Typ	Name	Kurzbeschreibung
F	tgetent	Termcap
F	tgetflag	Termcap
F	tgetnum	Termcap
F	tgetstr	Termcap
F	tgoto	Termcap
F	tputs	Termcap

1.4 Bibliotheken

Die hier beschriebenen Funktionen finden Sie in folgenden Bibliotheken:

/lib/libc.a — Standard-C-Bibliothek; ist automatisch dabei, wenn Sie cc aufrufen.

/lib/libm.a — Mathematische Funktionen

/usr/lib/libtermcap.a — Funktionen zur Bildschirm-Programmierung

/lib/libffp.a — Gleitkomma-Arithmetik (nur Version 1.0B und 1.0C)

1.5 Weise Ratschläge

Dieses Kapitel gibt einige hilfreiche Hinweise, die C-Programmierer berücksichtigen sollten, insbesondere, wenn ihr Programm auf verschiedenen Rechnern laufen soll.

1.5.1 C-Funktion statt Systemaufruf

Bietet sich zur Lösung eines Problems ein Systemaufruf und eine C-Funktion an, sollten Sie immer die C-Funktion verwenden. Dies gilt insbesondere für die Funktionen zur Dateibearbeitung. Verwenden Sie, wenn möglich, die Funktionen aus der Standardein/ausgabe-Bibliothek statt den Systemaufrufen open, close, read und write.
Das gleiche gilt für die Funktionen zur Speicherverwaltung. Arbeiten Sie lieber mit malloc, calloc und free statt mit brk und sbrk.
Folgende Systemaufrufe sind sehr maschinenabhängig und sollten daher nur verwendet werden, wenn es ausdrücklich erforderlich ist:

ioctl
nlist
profil
ptrace

1.5.2 Vorsicht: Fehler!

Guter Programmierstil schreibt vor, daß nach jedem Funktionsaufruf abgeprüft wird, ob ein Fehler vorliegt, etwa

```
if ( fkt(...) == fehlerergebnis)
  {
    perror("fkt:");
    exit(fehlercode);
  }
else...
```

Die Funktion perror ist dabei eine hilfreiche Unterstützung. In den Beispielen der einzelnen Funktionsbeschreibungen wurden diese Abfragen häufig weggelassen, um die Beispiele nicht unnötig aufzublähen.

1.5.3 Ergebnistyp Zeiger

<typ> *funkt(...)

Funktionen, die einen Zeiger zurückliefern, schreiben i.a. ihr Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird. Weil diese Tatsache eine häufige Fehlerquelle ist, wird bei solchen Funktionen darauf hingewiesen (Abschnitt **Achtung**).

1.5.4 Ergebnisparameter Zeiger

<typ1> funkt(...,parv,...**)**
<typ> *parv;

Wenn eine Funktion einen Zeiger als Ergebnisparameter hat, müssen Sie vor Aufruf der Funktion den Speicherplatz für das Ergebnis explizit bereitstellen, wie z.B.

```
struct stat dinf;  /* Speicherplatz bereitstellen */
fstat(dk,&dinf);   /* Funktionsaufruf */
```

Weil man dies häufig vergißt, erinnern wir daran in der jeweiligen Funktionsbeschreibung (Abschnitt **Achtung**).

1.5.5 Konstante oder symbolische Konstante?

Sie sollten, um Ihr Programm lesbarer zu machen, wenn es angebracht ist, symbolische Konstanten verwenden.
Aus Portabilitätsgründen sollen außerdem die include-Dateien berücksichtigt werden, die in einer Funktionsbeschreibung angegeben sind. Zunächst müssen Sie sich vergewissern, daß die include-Datei auf Ihrem Rechner vorhanden ist und anschließend ihren Inhalt überprüfen.
Bei einigen Dateien aus /usr/include/sys ist Vorsicht geboten, da sie äußerst maschinenabhängig sind.

2 Nachschlageteil

Sie finden hier in alphabetischer Reihenfolge die Beschreibungen aller Systemaufrufe, C-Funktionen und Makros, die Ihnen als Benutzer im C-Entwicklungssystem zur Verfügung stehen.

Prozeß abbrechen

int abort()

abort schickt das Signal SIGIOT an den aufrufenden Prozeß. Das Signal bewirkt, daß der Prozeß abgebrochen und ein Kernspeicherabzug abgelegt wird.

Typ

C-Funktion

Parameter

keine

Meldung

'IOT trap - Speicherabzug (core) auf Platte geschrieben'.

> > > > signal, exit, kill, adb(Kommando)

Absolutbetrag einer ganzen Zahl

int abs(i)
int i;

abs berechnet den Absolutbetrag einer ganzen Zahl.

Typ

C-Funktion

Parameter

int i	Ganzzahlige Variable, deren Absolutbetrag berechnet werden soll.

Ergebnis

|i|	für eine ganzzahlige Variable *i*.

Beispiel

Gib zu einem eingelesenen Wert den entsprechenden Absolutbetrag:

```
#include <stdio.h>

main()
{
   int i;
   if(scanf("%d",&i) == 1)
      printf("i=%d : |i|=%d\n",i,abs(i));
}
```

>>>> cabs, fabs

Zugriffsrechte für Dateien oder Dateiverzeichnisse überprüfen

int access(name,modus)
char *name;
int modus;

access prüft, ob auf eine Datei oder ein Dateiverzeichnis auf eine bestimmte Art zugegriffen werden darf. access überprüft das Zugriffsrecht in Bezug auf die **reale** Benutzer- und Gruppennummer des Prozesses.

Typ

Systemaufruf

Parameter

char *name	Dateiname der Datei oder des Dateiverzeichnisses
int modus	ganze Zahl, die angibt, was überprüft werden soll:
0	überprüfen, ob die Datei *name* existiert und ob die Dateiverzeichnisse, die auf dem Pfad zu *name* liegen, durchsucht werden dürfen.
	Ansonsten können nach folgendem Code die Zugriffsrechte überprüft werden:
1	ausführen bzw. durchsuchen
2	schreiben
4	lesen
5	lesen und ausführen/durchsuchen
6	schreiben und lesen
7	schreiben, lesen und ausführen/durchsuchen

Ergebnis

0 das in *modus* angegebene Zugriffsrecht besteht.

-1 Fehler, wenn

- die gewünschte Zugriffsart nicht gestattet ist oder ein Dateiverzeichnis auf dem Pfad zu *name* nicht durchsucht werden darf oder
- *name* nicht existiert oder
- eine Komponente auf dem Pfad zu *name* kein Dateiverzeichnis ist

Fehlermeldung

Liefert access das Ergebnis -1, so wird in errno zusätzlich ein Fehlercode abgelegt und zwar:

EACCES : Zugriff nicht gestattet
ENOENT : Datei oder Dateiverzeichnis unbekannt
ENOTDIR : Kein Dateiverzeichnis

Hinweis

- Die Überprüfung der Zugriffsrechte richtet sich nach den Eintragungen im Indexeintrag der Datei:
  beim Dateieigentümer werden die Eigentümerbits überprüft, bei einem Gruppenmitglied werden die Gruppenbits überprüft, bei allen Anderen werden die Zugriffsbits für Andere überprüft.
- Die Tatsache, daß access bzgl. realer Benutzernummer überprüft, können Sie in Programmen ausnutzen, bei denen das s-Bit gesetzt ist.
  Der Aufruf von access in so einem Programm ermöglicht es dem Programm-Aufrufer (auch wenn er nicht der Eigentümer der Programmdatei ist), seine Zugriffsrechte zu ermitteln.
- Beachten Sie, daß access lediglich die Schutzbits des Indexeintrages der Datei oder des Dateiverzeichnisses überprüft.
  Es kann daher vorkommen, daß Sie Operationen auf eine Datei oder ein Dateiverzeichnis nicht ausführen könnnen, obwohl access die erforderlichen Zugriffsrechte bestätigt, wie in folgendem Beispiel:

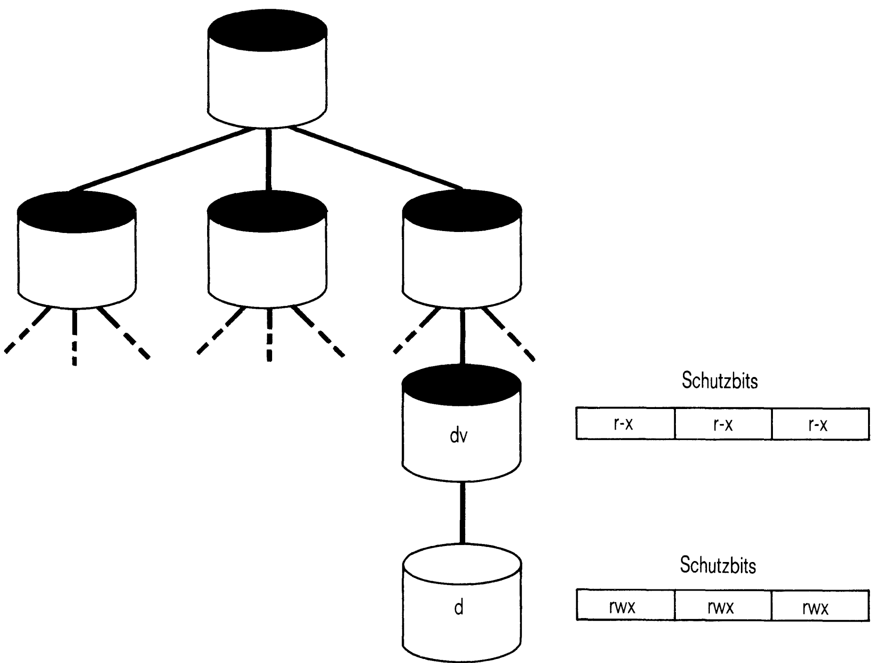


Bild 2-1 Situation im Dateibaum:

Der Aufruf access(d,7) liefert das Ergebnis 0, d.h. gemäß access haben Sie volles Zugriffsrecht auf die Datei *d.* Trotzdem können Sie die Datei z.B. nicht löschen, da Sie im übergeordneten Dateiverzeichnis *dv* keine Schreiberlaubnis haben.

Beispiel

Mit dem folgenden Programm wird überprüft, ob eine Datei ausführbar ist:

```
#include <errno.h>
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
  if((access(argv[1],1)) == 0)
    printf("Sie können die Datei ausführen\n");
  else perror("Fehler");
}
```

>>>> chmod, stat, fstat

arcus cosinus

#include <math.h>

double acos(x)
double x;

acos ist die Umkehrfunktion zu cos und berechnet zu einer Zahl aus dem Intervall [-1.0, +1.0] den entsprechenden Winkel im Bogenmaß.

Typ

C-Funktion

Parameter

double x	Zahl, deren Arcus Cosinus berechnet werden soll. Bei ganzzahligen Werten kann es zu Rundungsfehlern kommen.

Ergebnis

arcus cosinus(x), eine Gleitkommazahl vom Typ double aus [0,TT]	
	für Werte *x* aus dem Intervall [-1.0, +1.0]
0	für Werte außerhalb [-1.0, +1.0]

Fehlermeldung

Bei Argumenten außerhalb [-1.0, +1.0] besetzt acos die Variable errno mit dem Fehlercode :

EDOM : Argument zu groß

Hinweis

Wenn Sie in Ihrem Programm acos verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Folgendes Programm druckt für Werte aus dem Intervall [0.0,1.0] die entsprechenden Arcus Cosinus-Werte:

```
#include <math.h>
#include <stdio.h>

main( )
{
  double x;
  for(x = 0.0; x < 1.1; x = x+0.1)
  printf("x=%g : acos(%g)=%g\n",x,x,acos(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > cos, sin, tan, asin, atan

Alarmuhr stellen

unsigned alarm(sek)
unsigned sek;

alarm löst nach einer als Parameter angegebenen Zeitspanne das Signal SIGALRM beim aufrufenden Prozeß aus. Danach wird der Prozeß abgebrochen, falls das Signal nicht abgefangen wird (was i.a. geschieht, siehe Beispiel). alarm Aufrufe werden nicht gekellert. Nacheinander folgende Aufrufe setzen die Alarmuhr neu. Da die Uhr einen 1-Sekunden-Takt hat, kann es beim Auslösen des Signals zu Verschiebungen bis zu einer Sekunde kommen.

Typ

Systemaufruf

Parameter

unsigned sek

Zeitspanne in Sekunden, nach deren Ablauf alarm das Signal an den Prozeß schicken soll.

0

Der Aufruf alarm(0) löst keinen Alarm aus, setzt die Alarmuhr auf 0 und löscht noch nicht erledigte Alarmanfragen.

Ergebnis

Restzeit in der Alarmuhr vor Ausführung des Alarmaufrufes.

Hinweis

- fork setzt die Alarmuhr des neuen Prozesses auf 0.
- Bei einem exec Aufruf übernimmt das aufgerufene Programm die Restzeit in der Alarmuhr vom aufrufenden Programm.

- Wird das Signal abgefangen, kann sich das Wiederaufsetzen des unterbrochenen Prozesses aus Prioritätsgründen verzögern. Zur Illustration probieren Sie das Beispiel unten mit folgender Änderung:

```
for(;;)
   printf("*");
```

- Mit der Zuweisung: i = alarm(0) stellen Sie die Alarmuhr ab und können zudem feststellen, wieviel Zeit seit der letzten Alarmanforderung noch übrig gewesen wäre.

Beispiel

Circa alle zwei Sekunden ertönt am Bildschirm der Piepston:

```
#define PIEP                    '\7'
               /* Piepston ([CTRL] g) */
#include <signal.h>
#include <stdio.h>

catch()
              /* Signalbehandlung für SIGALRM */
{
    putchar(PIEP);
               /* Signalroutinen neu aufsetzen,
                  damit der Prozeß nicht abgebrochen wird */
    signal(SIGALRM,catch);
               /* Signal neu aufsetzen */
    alarm(2);
}

main()
{
  signal(SIGALRM,catch);
  alarm(2);
  for(;;)
      ;
}
```

> > > > pause, signal, sleep

Datum mit Uhrzeit in Englisch

#include <time.h>

char *asctime(tm)
struct tm *tm;

asctime wandelt eine gemäß der Struktur tm aufgeschlüsselte Zeitangabe in eine ASCII-Zeichenreihe um.
Die Ergebniszeichenreihe hat die Länge 26 und das Format einer Datums-mit-Uhrzeit-Angabe in Englisch:

```
                Wochentag Monat Tag Std:Min:Sek Jahr
zum Beispiel:         MON   DEC  10  15: 20: 54 1984\n\0
```

Typ

C-Funktion

Parameter

struct tm *tm

Struktur gemäß der include-Datei <time.h>:

```
struct tm{
      int   tm_sec;          Sekunden
      int   tm_min;          Minuten
      int   tm_hour;         Stunden (24 Stunden Zeit)
      int   tm_mday;         Monatstag (1-31)
      int   tm_mon;          Monat (0-11)
      int.  tm_year;         Jahr (minus 1900)
      int   tm_wday;         Wochentag (0-6, Sonntag=0)
      int   tm_yday;         Jahrestag (0-365)
      int   tm_isdst;        Sommerzeitanzeige,
                             nicht unterstützt
}
```

Ergebnis

Zeiger auf die erzeugte ASCII-Zeichenreihe

Die Ergebniszeichenreihe hat die Länge 26 und das gleiche Format wie bei ctime.

Achtung

- asctime schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Außerdem verwenden asctime und meztime denselben Datenbereich. Wenn sie also hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben.

Hinweis

Die Aufrufe asctime(localtime(sek_zg)) und ctime(sek_zg) sind äquivalent.

Beispiel

```
#include <time.h>

long time();
char *asctime();
struct tm *gmtime();
struct tm *zeit;
char *daten;
long clock;

main()
{
   clock = time(0L);
   zeit = gmtime(&clock);
   printf("Jahr: 19%d\n",zeit->tm_year);
   printf("Uhrzeit in Stunden: %d\n",zeit->tm_hour);
   printf("Jahrestag: %d\n",zeit->tm_yday);

   daten = asctime(zeit);
   printf("%s",daten);
}
```

Dateien

/usr/include/time.h
Definition der Struktur tm

> > > > ctime, ftime, gctime, gmtime, localtime, meztime, time

arcus sinus

#include <math.h>

double asin(x)
double x;

asin ist die Umkehrfunktion zu sin und berechnet zu einer Zahl aus dem Intervall [-1.0, +1.0] den entsprechenden Winkel im Bogenmaß.

Typ

C-Funktion

Parameter

double x — Zahl, deren Arcus Sinus berechnet werden soll. Bei ganzzahligen Werten kann es zu Rundungsfehlern kommen.

Ergebnis

arcus sinus(x), ein Gleitkommawert vom Typ double aus [-TT/2, +TT/2]
für Werte *x* aus dem Intervall [-1.0, +1.0]

0 — für Werte außerhalb [-1.0, +1.0]

Fehlermeldung

Bei Werten außerhalb [-1.0, +1.0] besetzt asin die Variable errno mit dem Fehlercode:

EDOM : Argument zu groß

Hinweis

Wenn Sie in Ihrem Programm asin verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Drucke für die Werte 0.0, 0.1,..., 1.0 die entsprechenden Arcus Sinus-Werte:

```
#include <math.h>
#include <stdio.h>

main()
{
   double x;
   for(x = 0.0; x < 1.1; x = x+0.1)
   printf("x=%g : asin(%g)=%g\n",x,x,asin(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > sin, cos, acos, tan, atan

arcus tangens

#include <math.h>

double atan(x)
double x;

atan ist die Umkehrfunktion zu tan und berechnet zu einer Gleitkommazahl den entsprechenden Winkel im Bogenmaß.

Typ

C-Funktion

Parameter

double x   Zahl, deren Arcus Tangens berechnet werden soll.

Ergebnis

arcus tangens(x), ein Gleitkommawert vom Typ double aus
]-TT/2, + TT/2[

Hinweis

Wenn Sie in Ihrem Programm atan verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Folgendes Programm druckt zu einem eingelesenen Wert den entsprechenden Arcus Tangens:

```
#include <math.h>
#include <stdio.h>

main()
{
  double x;
  if( scanf("%lf",&x) == 1)
     printf("x=%g : atan(%g)=%g",x,x,atan(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > tan, sin, asin, cos, acos

Umwandlung einer Zeichenreihe in eine Gleitkommazahl

double atof(zg)
char *zg;

atof wandelt eine ASCII-Zeichenreihe in eine Gleitkommazahl um. Die umzuwandelnde Zeichen-Darstellung der Zahl kann dabei wie folgt aufgebaut sein:

```
          ⎧tab⎫      ⎧+⎫                              ⎧E⎫ ⎧+⎫
(*)      [⎨   ⎬...][⎨ ⎬][Ziffer...][.][Ziffer...][⎨ ⎬[⎨ ⎬]Ziffer...]
          ⎩ ␣ ⎭      ⎩-⎭                              ⎩e⎭ ⎩-⎭
```

Typ

C-Funktion

Parameter

char *zg	Zeiger auf die umzuwandelnde ASCII-Zeichenreihe

Ergebnis

Gleitkommazahl vom Typ double
für Zeichenreihen, die eine wie oben in (*) beschriebene Struktur haben und einen Zahlenwert darstellen, der im Intervall [-{FLOAT}, +{FLOAT}] liegt

0 für Zeichenreihen, die nicht mit Ziffern beginnen.

Fehlermeldung

bei Zeichenreihen, deren Zahlenwert außerhalb dem zulässigen Gleitkommabereich liegt, wird das Programm abgebrochen (Signal SIGFPE) mit der Meldung:
'Gleitkomma Ausnahme-Speicherabzug (core) auf Platte geschrieben'

Hinweis

- Für die Bereiche [-10^{53}, -{FLOAT}] und [{FLOAT}, 10^{54}] gibt es keine Überlaufbehandlung in der Version 1.0B und 1.0C.
- atof erkennt auch Zeichenreihen, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. atof schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Beispiel

Folgendes Programm wandelt eine beim Aufruf übergebene Zeichenreihe in die entsprechende Gleitkommazahl um.

```
#include <stdio.h>

double atof();

main(argc,argv)
int argc;
char **argv;
               /* Zahlen werden als Zeichenreihen!! übergeben.
                  Eine Umwandlung ist erforderlich,
                  falls der Zahlenwert benötigt wird */
{
   ++argv;
               /* Programmname überlesen */
   printf("floating : %f\n",atof(*argv));
}
```

> > > > atoi, atol, fscanf, scanf,sscanf

Umwandlung einer Zeichenreihe in eine ganze Zahl

int atoi(zg)
char *zg;

atoi wandelt eine ASCII-Zeichenreihe in eine ganze Zahl um. atoi erkennt Zeichenreihen, die wie folgt aufgebaut sind:

```
        ⎡tab⎤      ⎡+⎤
(*)    [⎨   ⎬...][⎨ ⎬]Ziffer...
        ⎣ ␣ ⎦      ⎣-⎦
```

Typ

C-Funktion

Parameter

char *zg   Zeiger auf die umzuwandelnde Zeichenreihe

Ergebnis

ganzzahliger Wert vom Typ int
für Zeichenreihen, die eine wie oben in (*) beschriebene Struktur haben und einen Zahlenwert darstellen, der im Intervall [-{INT}, +{INT}-1] liegt.

ganzzahlig abgerundeter Wert
für Zeichenreihen, deren Zahlenwert eine Gleitkommazahl in [-{INT}, +{INT}-1] ist

Überlauf   für Zeichenreihen, die eine Zahl außerhalb [-{INT}, +{INT}-1] darstellen

0   sonst, d.h. für Zeichenreihen, die nicht mit Ziffern beginnen.

Hinweis

– Es gibt keine Überlaufbehandlung.

– atoi wandelt auch Zeichenreihen um, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. atoi schneidet den Ziffernteil ab, wandelt ihn wie oben beschrieben um und ignoriert den Rest.

Beispiel

Folgendes Programm wandelt eine beim Aufruf übergebene Zeichenreihe in den entsprechenden ganzzahligen Wert um.

```
#include <stdio.h>

main(argc,argv)
int argc;
char **argv;
                /* Zahlen werden als Zeichenreihe!! übergeben.
                   Eine Umwandlung ist erforderlich,
                   falls der Zahlenwert benötigt wird. */
{
  ++argv;
                /* Programmname überlesen */
  printf("integer : %d\n",atoi(*argv));
}
```

>>> atof, atol, fscanf, scanf, sscanf

Umwandlung einer Zeichenreihe in eine ganze Zahl (Typ long)

long atol(zg)
char *zg;

atol wandelt eine ASCII-Zeichenreihe in eine ganze Zahl vom Typ long um. atol erkennt die gleichen Zeichenreihen wie atoi, nämlich:

```
          ⎧tab⎫      ⎧+⎫
(*)      [⎨   ⎬...][⎨ ⎬]Ziffer...
          ⎩ ␣ ⎭      ⎩-⎭
```

Typ

C-Funktion

Parameter

char *zg	Zeiger auf die umzuwandelnde ASCII-Zeichenreihe

Ergebnis

ganzzahliger Wert vom Typ long
: für Zeichenreihen, die eine wie in (*) beschriebene Struktur haben und einen Zahlenwert darstellen, der im Intervall [-{LONGINT}, +{LONGINT}-1] liegt

ganzzahlig abgerundeter Wert
: für Zeichenreihen, deren Zahlenwert eine Gleitkommazahl in [-{LONGINT}, +{LONGINT}-1] ist

Überlauf
: für Zeichenreihen, die eine Zahl außerhalb [-{LONGINT}, +{LONGINT}-1] darstellen

0
: sonst, d.h. für Zeichenreihen, die nicht mit einer Ziffer beginnen

Hinweis

- Es gibt keine Überlaufbehandlung.
- atol wandelt auch Zeichenreihen um, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. atol schneidet den Ziffernteil ab, wandelt ihn wie oben beschrieben um und ignoriert den Rest.

Beispiel

Folgendes Programm wandelt eine beim Aufruf übergebene Zeichenreihe in den entsprechenden ganzzahligen Wert um.

```
#include <stdio.h>
long atol();

main(argc,argv)
int argc;
char **argv;
               /* Zahlen werden als Zeichenreihe !! übergeben.
                  Eine Umwandlung ist erforderlich,
                  falls der Zahlenwert benötigt wird */
{
  ++argv;
               /* Programmname überlesen */
  printf("long integer : %ld\n",atol(*argv));
}
```

> > > > atof, atoi, fscanf, scanf, sscanf

Größe des Datensegmentes verändern

int brk(adr**)**
unsigned adr;

brk und sbrk werden dazu verwendet, den Speicherplatz für das Datensegment eines Prozesses dynamisch zu verändern. brk sorgt dafür, daß der adressierbare Speicher (Ende des Datensegmentes) bis zur Adresse *adr* zur Verfügung steht.

Schematisch sieht die Speicherverteilung wie folgt aus:

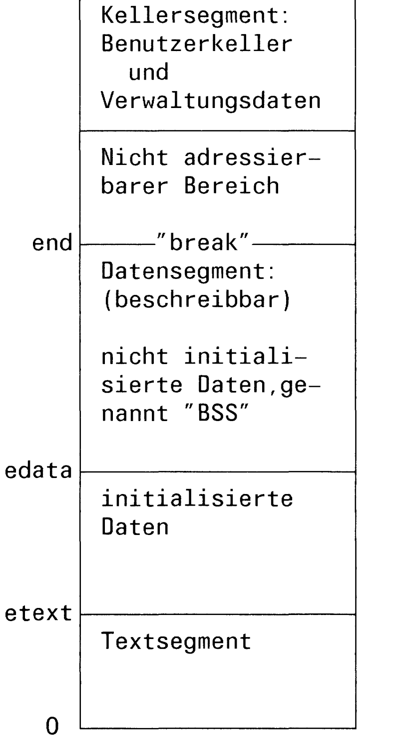


Bild 2-2 logische Speicherverteilung

Die Wirkung eines brk Aufrufes, der das Datensegment verkleinern soll, läßt sich dann wie folgt darstellen:

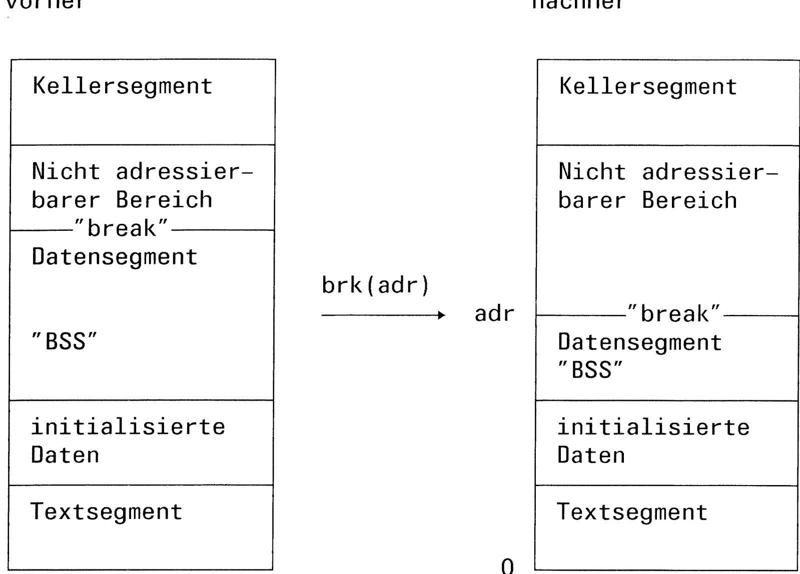


Bild 2-3 Wirkung eines brk Aufrufes

Der "break" kennzeichnet das Ende des für den Benutzer zugänglichen Datensegmentes und ist die erste nicht belegte Adresse.
Der daran anschließende Bereich gehört nicht mehr zum Adreßraum des Prozesses. Der Versuch, in diesen Bereich zu adressieren, führt zu einem Speicherfehler.
brk setzt den "break" neu und zwar auf die Adresse, die Sie in *adr* angeben, aufgerundet auf das nächste Vielfache von {BRK_ZAHL} Bytes.

Typ

Systemaufruf

Parameter

unsigned adr

Adresse in Bytes für den "break"

Ergebnis

0	der "break" wurde neu gesetzt
-1	der "break" wurde nicht neu gesetzt, weil das Programm mehr Speicher fordert, als das System zuläßt (siehe auch ulimit).

Fehlermeldung

Ende mit Fehler liefert in errno den Fehlercode:

ENOSPC : Speicherkapazität erschöpft

Hinweis

- Wenn die Ausführung eines Programms beginnt, bestimmmt das System automatisch die durch das Programm (statisch) festgelegten Bereiche für Text- und Datensegmente und setzt damit auch den "break" und zwar auf die erste freie Adresse über den nicht initialisierten Daten ("BSS"). Daher müssen nur die Programme, deren Datensegmente während der Ausführung (dynamisch) wachsen, selbstständig die Speicherverwaltung organisieren.
- brk kann mit jedem Wert aufgerufen werden, der im Bereich der Adressen liegt, die von einem sbrk Aufruf geliefert wurden. Falls Sie brk benutzen, um außerhalb dieses Bereiches Speicher zuzuteilen oder freizugeben, kann es zu unerwünschten Effekten kommen!
  Die einzige sinnvolle Anwendung von brk ist daher die Freigabe eines großen Speicherbereiches, der zuvor mittels sbrk zugewiesen wurde. Wenn dabei ein Bereich freigegeben und anschließend wieder zugewiesen wird, bleibt der alte Inhalt nicht notwendig erhalten!
- Wenn möglich, verwenden Sie malloc und calloc, um dynamisch Speicherplatz zu besorgen. Auf keinen Fall sollte in einem Programm sowohl malloc (calloc) als auch brk (sbrk) vorkommen!

Beispiel

Folgendes Programm zeigt die Wirkung von brk und sbrk und die Veränderung des Kellers beim Ablauf einer rekrursiven Funktion:

Wenn Sie das Programm übersetzt haben, können Sie es in der Form:

a.out [-**b**adresse] [-**s**inkr] [-**r**n]

aufrufen. Dabei bedeutet:

-badresse
: setze den "break" auf *adresse*

-sinkr
: verändere den "break" um *inkr*

-rn
: berechne n-Fakultät

```
#include <stdio.h>

extern etext();
extern edata;
extern end;

char gebrauch[]= "gebrauch: a.out -badr -sinkr -rn\n"

anzeige()
{
               /* Variable, mit der die Veränderung
                  des Kellers gezeigt wird */
  int mark = 0;
  printf("%u: %u: %u: %u: %u\n",&etext,&edata,&end,sbrk(0),&mark);
}
```

```
int fak(n)
               /* Fakultätsfunktion */
int n;
{
  printf("%d!\n",n);
  anzeige();
  if(n)
    return(n*fak(n-1));
  return 1;
}

main(argc,argv)
int argc;
char **argv;
{
  register int num;
  printf("etext: edata: end: break: Keller\n");
  anzeige();

  while(--argc && **++argv == '-')
      {
        num = atoi(*argv + 2);
        switch((*argv[1]) {
              case 'b'  :  printf("brk(%d) = %d\n",num,brk(num));
                           break;
              case 's'  :  printf("sbrk(%d) = %d\n",num,sbrk(num));
                           break;
              case 'r'  :  printf("%d! = %d\n",num,fak(num));
                           break;
              default   :  fputs(gebrauch, stderr);
                           exit(1);
                          }
        anzeige();
      }
}
```

Externe Größen

extern end
: erste Adresse oberhalb des Datensegmentes
Wenn die Programmausführung beginnt, ist der "break" automatisch auf end gesetzt (siehe oben).

extern etext
: erste Adresse oberhalb des Textsegmentes

extern edata
: erste Adresse oberhalb der initialisierten Daten

> > > > sbrk, exec, calloc, malloc, ulimit

Absolutbetrag einer komplexen Zahl

#include <math.h>

double cabs(z**)**
struct {double x,y;**}** z;

cabs berechnet den Betrag einer komplexen Zahl.

Typ

C-Funktion

Parameter

struct {double x,y;} z
Komplexe Zahl *z* mit Realteil *x* und Imaginärteil *y*.

Ergebnis

sqrt(x*x + y*y)
Absolutbetrag der komplexen Zahl *z*.

Fehlermeldung

Bei Überlauf bricht das Programm ab (Signal SIGFPE) und bringt die Meldung:
'Gleitkomma Ausnahme - Speicherabzug(core) auf Platte geschrieben'

Hinweis

Verwenden Sie in Ihrem Programm cabs, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Folgendes Programm berechnet den Absolutbetrag einer komplexen Zahl:

```
#include <stdio.h>

double cabs( );

main()
{
 struct {double x,y;}z;
 if ( scanf("%lf%lf",&z.x,&z.y) == 2)
     printf("%f : Absolutbetrag\n",cabs(z));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > abs, fabs, hypot, sqrt

Speicherplatz reservieren

char *calloc(n,elgröße**)**
unsigned n,elgröße;

calloc beschafft zur Ausführungszeit zusammenhängenden Speicherplatz für ein Feld mit *n* Elementen, wobei jedes Element *elgröße* Bytes beansprucht. calloc initialisiert jedes Element des neuen Feldes zu Null.

Typ

C-Funktion

Parameter

unsigned n — Anzahl der Elemente des Feldes

unsigned elgröße
Größe eines Feldelementes in Bytes

Ergebnis

Zeiger auf die Anfangsadresse des zugewiesenen Speicherplatzes falls genügend Speicherplatz vorhanden ist

Nullzeiger — falls der Speicherplatz für die Anforderungen nicht ausreicht

Beispiel

calloc können Sie einsetzen, um Felder zu bearbeiten, deren Größe im voraus nicht bekannt ist.
In folgendem Programmstück reserviert calloc Speicherplatz für ein Feld, dessen Größe und Elemente von der Standardeingabe eingelesen werden:

```
main()
{
  int i;
  char *feld;
  unsigned anz,groesse;
  if ( scanf("%d %d",&groesse,&anz) == 2)
    {
      feld = calloc(anz,groesse);
      for(i = 0;i < anz;i++)
         scanf("%d",&feld[i]);
    }
}
```

> > > > malloc, free, realloc

Aufrunden

```
#include <math.h>

double ceil(x)
double x;
```

ceil rundet eine Gleitkommazahl nach oben (ganzzahlig) auf.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl, die aufgerundet werden soll

Ergebnis

kleinste ganze Zahl, die größer oder gleich *x* ist.

Hinweis

Wenn Sie in Ihrem Programm ceil verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

```
#include <math.h>
#include <stdio.h>

main()
{
  double x;
  if ( scanf("%lf",&x) == 1)
      printf("Die Zahl %g wird aufgerundet zu %f\n",x,ceil(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > abs, fabs, floor

Aktuelles Dateiverzeichnis wechseln

int chdir(dvname**)**
char *dvname;

chdir macht *dvname* zum aktuellen Dateiverzeichnis.

Typ

Systemaufruf

Parameter

char *dvname
: Name des Dateiverzeichnisses, das aktuelles Dateiverzeichnis werden soll

Ergebnis

0
: das aktuelle Dateiverzeichnis wurde geändert.

–1
: chdir hat das aktuelle Dateiverzeichnis nicht geändert, da

- kein Dateiverzeichnis *dvname* existiert oder
- ein Dateiverzeichnis auf dem Pfad zu *dvname* nicht durchsucht werden darf oder
- eine Komponente des Pfadnamens kein Dateiverzeichnis ist.

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT : Datei oder Dateiverzeichnis unbekannt
EACCES : Zugriff untersagt
ENOTDIR : Kein Dateiverzeichnis

Hinweis

Das aktuelle Dateiverzeichnis ist Ausgangspunkt für alle Pfadnamen, die nicht mit '/' beginnen.

Beispiel

Sie möchten das bei Programmaufruf übergebene Argument zum aktuellen Dateiverzeichnis machen und ein Inhaltsverzeichnis davon ausdrucken:

```
main(argc,argv)
int argc;
char **argv;
{
  chdir(*++argv);
  system("ls -l");
}
```

> > > > cd(Kommando), chroot

Zugriffsrechte ändern

int chmod(name,modus)
char *name;
int modus;

Nur für den Dateieigentümer oder Systemverwalter!
chmod ändert die Zugriffsrechte für die Datei *name* entsprechend *modus*.

Typ

Systemaufruf

Parameter

char *name — Name der Datei, deren Zugriffsrechte geändert werden sollen.

int modus — mit *modus* geben Sie an, wie Sie die Zugriffsrechte ändern wollen. Sie können eine (beliebige) vierstellige Oktalzahl angeben, wobei die Bedeutung der Oktalziffern 1, 2 und 4 durch folgende Tabelle festgelegt ist. Die Bedeutung der restlichen Ziffern ergibt sich aus entsprechenden Kombinationen (00006 bedeutet z.B. Lese- und Schreibrecht für Andere).

Oktalzahl		Bedeutung
04000		s-Bit für Eigentümer
02000		s-Bit für Gruppe
01000		Sticky-Bit
00400	Eigentümer:	lesen
00200		schreiben
00100		ausführen bzw. durchsuchen
00040	Gruppe:	lesen
00020		schreiben
00010		ausführen bzw. durchsuchen
00004	Andere:	lesen
00002		schreiben
00001		ausführen bzw durchsuchen

Ergebnis

0 die Zugriffsrechte wurden entsprechend *modus* geändert

–1 chmod kann die Zugriffsrechte nicht ändern, da

- die Datei *name* nicht vorhanden ist oder
- das Programm weder unter der Kennung des Dateieigentümers noch des Systemverwalters läuft oder
- eine Komponente des Pfadnamens kein Dateiverzeichnis ist oder
- ein Dateiverzeichnis auf dem Pfad zu *name* nicht durchsucht werden darf.

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT : Datei oder Dateiverzeichnis unbekannt
EPERM : Hat anderen Eigentümer
ENOTDIR : Kein Dateiverzeichnis
EACCES : Zugriff untersagt.

Hinweis

Wurde beim Übersetzen einer Programmdatei der -n oder -i Schalters im Kommando ld gesetzt, dann verhindert das sticky-Bit (modus = = 01000), daß das System das Programmtextsegment nach Ausführung auslagert und den zugeteilten Swapbereich freigibt. Soll dann das Programm wieder ausgeführt werden, so muß der Programmtext nicht erneut blockweise von der Platte in den Kernspeicher eingelesen werden, sondern kann in einem Stück aus dem Swapbereich geladen und ausgeführt werden.
Nur der Systemverwalter darf modus = 01000 angeben.
Die Angabe ist nur sinnvoll bei Kommandos, die sehr oft benutzt werden (wie z.B. ed, cat, ls).

Beispiel

Ändere das Zugriffsrecht der als Argument übergebenen Datei so, daß Eigentümer und Gruppe lesen, schreiben und ausführen dürfen:

```
main(argc,argv)
int argc;
char **argv;
{
  chmod(*++argv,00770);
}
```

> > > > fstat, stat, chmod(Kommando)

Eigentümer und Gruppe einer Datei ändern

int chown(name,ben_nr,gr_nr**)**
char *name;
int ben_nr;
int gr_nr;

Nur für den Systemverwalter!
chown ändert die Eigentümer- und Gruppenkennung der Datei *name*.

Typ

Systemaufruf

Parameter

char *name	Name der Datei, deren Kennungen geändert werden sollen.
int ben_nr	neue Eigentümernummer
int gr_nr	neue Gruppennummer

Ergebnis

0 — chown hat die neuen Nummern als Eigentümer- und Gruppenkennung eingetragen.

−1 — keine Änderung, falls

- eine Komponente des Pfadnamens kein Dateiverzeichnis ist oder
- keine Datei *name* existiert oder
- das Programm nicht unter der Kennung des Systemverwalters läuft.

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehler-code:

ENOTDIR : Kein Dateiverzeichnis
ENOENT : Datei oder Dateiverzeichnis unbekannt
EPERM : Hat anderen Eigentümer

Dateien

/etc/passwd Liste aller Systembenutzer

> > > > fstat, stat, chown(Kommando)

Root-Dateiverzeichnis ändern

int chroot(rname**)**
char *rname;

Nur für den Systemverwalter!
chroot setzt das Root-Dateiverzeichnis neu.

Typ

Systemaufruf

Parameter

char *rname
Name des neuen Root-Dateiverzeichnisses

Ergebnis

0 das Root-Dateiverzeichnis wurde geändert.

–1 chroot hat das Root-Dateiverzeichnis nicht geändert, da

- eine Komponente des Pfadnamens kein Dateiverzeichnis ist oder
- das Dateiverzeichnis *rname* nicht existiert oder
- das Programm nicht unter der Kennung des Systemverwalters läuft.

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOTDIR : Kein Dateiverzeichnis
ENOENT : Datei oder Dateiverzeichnis unbekannt
EPERM : Hat anderen Eigentümer

Hinweis

Das Root-Dateiverzeichnis ist der Ausgangspunkt für alle Pfadnamen, die mit '/' beginnen.

Lese- oder Schreib-Fehleranzeige löschen

#include <stdio.h>

void clearerr(dz**)**
FILE *dz;

clearerr löscht die Lese- oder Schreib-Fehleranzeige auf einer Datei.

Typ

C-Funktion (s)

Parameter

FILE *dz	Zeiger auf die File-Struktur, in der die Fehleranzeige steht.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

Beispiel

siehe Beispiel bei ferror

> > > > ferror

Datei schließen

int close(dk)
int dk;

close schließt die Datei, der mittels open, creat, dup, fcntl oder pipe die Dateikennzahl *dk* zugewiesen wurde.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl der geöffneten Datei

Ergebnis

0	close hat die Datei mit der Kennzahl *dk* geschlossen.
-1	die Dateikennzahl ist unbekannt

Fehlermeldung

Bei Ergebnis -1 steht in errno der Fehlercode:

EBADF : Unzulässige Dateinummer

Hinweis

Bei Beendigung eines Prozesses (normal und mit exit) werden automatisch alle offenen Dateien des Prozesses geschlossen.
Pro Prozeß dürfen maximal {PDAT_MAX} Dateien gleichzeitig geöffnet sein. Daher müssen Programme, die mehr Dateien benötigen, zwischenzeitlich nicht gebrauchte Dateien schließen.

Beispiel

siehe Beispiel bei lseek

> > > > creat, dup, dup2, exec, fcntl, fclose, open, pipe, pclose

Cosinus

```
#include <math.h>

double cos(x)
double x;
```

cos berechnet für Gleitkommazahlen die trigometrische Funktion Cosinus.

Typ

C-Funktion

Parameter

double x	Winkel im Bogenmaß

Ergebnis

cos(x)	eine Gleitkommazahl im Intervall [-1.0, + 1.0]

Hinweis

Wenn Sie in Ihrem Programm cos verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Folgende Funktion gibt für Werte aus [-1.0, +1.0] die entsprechenden cosinus-Werte aus:

```
#include <math.h>

main()
{
   double x;
   for (x = -1.0; x<1.1; x = x+0.1)
       printf("cos(%lf) = %lf\n",x,cos(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

>>> acos, cosh, sin, asin, sinh, tan, atan, tanh

Cosinus Hyperbolicus

#include <math.h>

double cosh(x)
double x;

cosh berechnet den Cosinus Hyperbolicus für Gleitkommazahlen.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl

Ergebnis

cosh(x)	für eine Gleitkommazahl *x*.
+HUGE	bei Überlauf

Hinweis

Verwenden Sie cosh, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > acos, cos, sin, asin, sinh, tan, atan, tanh

Datei neu anlegen

int creat(name,modus**)**;
char *name;
int modus;

creat legt eine neue Datei an oder verkürzt eine Datei, die es bereits gibt, auf die Länge 0. Die Datei ist nach dem creat Aufruf zum Schreiben geöffnet.

- Existiert die Datei noch nicht, so errechnen sich die Zugriffsrechte der neuen Datei aus der Angabe in *modus* und der Schutzbitmaske des Prozesses (kurz: Prozeßmaske, siehe umask).
  Die effektive Benutzer- und Gruppennummer des Prozesses werden zur Eigentümer- und Gruppenkennung der neuen Datei.
- Gibt es die Datei bereits, bleiben Eigentümer und Schutzbits unverändert.

Typ

Systemaufruf

Parameter

char *name — Dateiname der Datei, die neu angelegt werden soll.

int modus — Sie können hier eine vierstellige Oktalzahl angeben.
Die tatsächliche Schutzbiteinstellung der neuen Datei ist das bitweise UND aus der Binärdarstellung von *modus* und dem Komplement der Prozeßmaske.
Beachten Sie, daß die Prozeßmaske nur drei relevante Oktalstellen hat, so daß die höchste Stelle von *modus* (s- und sticky-Bit) unverändert übernommen wird.

Beispiel:

	oktal	binär
Prozeßmaske	002	000 000 010
Komplement		111 111 101
modus	1772	001 111 111 010
Schutzbiteinstellung	1770	001 111 111 000

Sie sehen also, daß das sticky-Bit aus *modus* übernommen wird, die restlichen Schutzbits aber erst errechnet werden. So erhält man trotz Angabe 2 keine Schreiberlaubnis für Andere, weil die Prozeßmaske dies verbietet.

Das Ergebnis (als Oktalzahl) wird wie bei chmod nach folgender Tabelle gedeutet:

Oktalzahl		Bedeutung
04000		s-Bit für Eigentümer
02000		s-Bit für Gruppe
01000		Sticky-Bit
00400	Eigentümer:	lesen
00200		schreiben
00100		ausführen bzw. durchsuchen
00040	Gruppe:	lesen
00020		schreiben
00010		ausführen bzw. durchsuchen
00004	Andere:	lesen
00002		schreiben
00001		ausführen bzw durchsuchen

Ergebnis

Dateikennzahl

falls creat eine neue Datei angelegt oder eine bereits bestehende Datei zum Wieder-Beschreiben eröffnet hat.

-1

falls creat aus einem der folgenden Gründe nicht erfolgreich war:

- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder
- ein Dateiverzeichnis auf dem Pfad zu *name* darf nicht durchsucht werden oder in dem Dateiverzeichnis, in das die neue Datei eingetragen werden soll, ist Schreiben nicht erlaubt oder *name* existiert bereits ohne Schreiberlaubnis oder
- *name* ist ein Dateiverzeichnis oder
- es sind bereits {PDAT_MAX} Dateien geöffnet oder
- die Systemtabelle aller offenen Dateien im System ist voll ({SDAT_MAX}).

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOTDIR : Kein Dateiverzeichnis
EACCES : Zugriff untersagt
EISDIR : Ist ein Dateiverzeichnis
EMFILE : Zu viele offene Dateien im System
ENFILE : Überlauf der Dateientabelle im System

Hinweis

- Das sbe-Bit (siehe fcntl) der Datei wird standardmäßig auf 0 gesetzt, d.h. die Datei bleibt bei einem exec Aufruf geöffnet.
- Der Lese/Schreibzeiger steht auf Dateianfang.
- Wenn Sie eine Datei neu anlegen, können Sie die Zugriffsrechte beliebig definieren, auch so ,daß die Schreiberlaubnis fehlt. Die Datei ist jedoch nach erfolgreicher Ausführung von creat immer zum Schreiben geöffnet. Dies wird z.B. bei Programmen eingesetzt, die einen festen Satz von Temporärdateien verwenden:
  Eine Temporärdatei wird im ersten Programmlauf mittels creat ohne Schreiberlaubnis angelegt.
  Versucht dann ein zweiter Programmlauf dieselbe Datei erneut anzulegen, liefert creat das Ergebnis -1 und das Programm weiß, daß der Dateiname momentan nicht frei ist.

Beispiel

Anlegen der Datei *neu* mit der Schutzbitbelegung: rws r-- r-

```
#include <stdio.h>
#define MODE  04744

main()
{
   int dk;
               /* Prozeßmaske auf 0 setzen,
                  d.h. keine Einschränkungen */
   umask(000);
   dk = creat("neu",MODE);
   printf("%d\n",dk);
}
```

> > > > chmod, close, dup, dup2, fcntl, open, read, umask, write, perror

Verschlüsselung

char *crypt(wort,schlüssel**)**
char *wort, *schlüssel;

crypt dient zum Verschlüsseln von Passwörtern. Grundlage der Funktion ist der NBS Data Encryption Standard (DES).

Typ

C-Funktion

Parameter

char *wort — Passwort des Benutzers, das verschlüsselt werden soll

char *schlüssel
Zeichenreihe aus zwei Zeichen über dem Alphabet [a-zA-Z0-9./].
schlüssel dient zur Steuerung des DES Algorithmus.

Ergebnis

Zeiger auf das verschlüsselte Passwort, das wie *schlüssel* aus dem Alphabet [a-zA-Z0-9./] ist. Die beiden ersten Buchstaben sind identisch zu *schlüssel.*

Achtung

crypt schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Dateien

/etc/passwd — Liste aller Systembenutzer

> > > >setkey, encrypt, getpass, login(Kommando), passwd(Kommando)

Datum mit Uhrzeit (MEZ) in Englisch

char *ctime(sek _ zg**)**
long *sek _ zg;

ctime interpretiert den Wert, auf den *sek _ zg* zeigt, als Zeit in Sekunden seit dem 1. Januar 1970 00:00:00 (GMT) (siehe time).
Es berechnet daraus Ortszeit (MEZ) und wandelt das Ergebnis in eine ASCII-Zeichenreihe um. Die Ergebniszeichenreihe hat die Länge 26 und das Format einer Datums-mit-Uhrzeit-Angabe in Englisch:

```
                  Wochentag  Monat  Tag  Std:Min:Sek  Jahr
zum Beispiel:           Mon    Jan   28   12: 34: 00  1985\n\0
```

Typ

C-Funktion

Parameter

long *sek _ zg
Zeiger auf die Zeitangabe in Sekunden

Ergebnis

Zeiger auf die erzeugte ASCII-Zeichenreihe der Länge 26.

Achtung

- ctime schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Außerdem verwenden ctime und gctime denselben Datenbereich d.h., wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben!
- Für die Ausgabe gilt die 24-Stunden-Uhr.

Beispiel

Wandele einen Wert in Ortszeit um und gib das Ergebnis in Form einer englischen Datums-mit-Uhrzeit-Angabe aus:

```
main()
{
   long sek, time();
   sek = time(0L);
   printf("%s",ctime(&sek));
}
```

>>>>asctime, localtime, gmtime, time, gctime, meztime, date(Kommando)

Zusätzliche Dateikennzahl einrichten

int dup(dk)
int dk;

dup ordnet einer bereits offenen Datei eine weitere Dateikennzahl zu: Die Systemaufrufe creat, dup, dup2, fcntl, open und pipe liefern eine Dateikennzahl *dk*, die zur Identifikation der geöffneten Datei dient. dup richtet ein Synonym für *dk* ein, so daß gilt:

- die neue und alte Dateikennzahl bezeichnen dieselbe Datei und haben den gleichen Dateizeiger
- die Zugriffsberechtigung (lesen, schreiben oder beides) bleibt unverändert
- das sbe-Bit (siehe fcntl) wird auf 0 gesetzt d.h., die Datei bleibt nach einem exec Aufruf geöffnet.

Typ

Systemaufruf

Parameter

int dk — Dateikennzahl, die mittels creat, dup, dup2, fcntl open oder pipe zugewiesen wurde.

Ergebnis

neue Dateikennzahl
dup ordnet die kleinste Dateikennzahl zu, die gerade frei ist (siehe Beispiel unten).

-1 — dup hat keine neue Dateikennzahl zugeordnet, da

- *dk* eine ungültige Dateikennzahl ist oder
- bereits {PDAT_MAX} Dateien geöffnet sind

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EBDAF : Unzulässige Dateinummer
EMFILE : Zu viele offene Dateien im System

Beispiel

Dateikennzahl 0 für Standardeingabe wird mit *datei* verbunden:

```
#include <stdio.h>

main()
{
  int dk, c;
               /* datei zum Lesen öffnen */
  if((dk = open("datei",0)) == -1)
     printf("Datei kann nicht geöffnet werden\n");
               /* Dateikennzahl 0 für Standardeingabe
                  freigeben */
  close(0);
               /* dk verdoppeln; niedrigste freie
                  Dateikennzahl (jetzt 0) wird zugewiesen */
  dup(dk);
               /* Dateikennzahl 0 weist jetzt auf datei */
               /* Dateikennzahl dk wird nicht mehr
                  benötigt */
  close(dk);
               /* einlesen von Standardeingabe,
                  d.h jetzt direkt von datei */
  while((c=getchar()) != EOF)
       putchar(c);
}
```

> > > > dup2, creat, open, close, pipe, fcntl

Zusätzliche Dateikennzahl einrichten

int dup2(dk,n**)**
int dk,n;

Wie dup ordnet dup2 einer bereits offenen Datei eine weitere Dateikennzahl zu. Zusätzlich geben Sie bei dup2 mit dem Parameter *n* an, welche Zahl zugewiesen werden soll.
Wenn *n* bereits auf eine offene Datei verweist, schließt dup2 diese Datei, bevor *n* neu zugewiesen wird.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl, die mittels creat, fcntl, dup, open oder pipe zugewiesen wurde
int n	eine gültige Dateikennzahl, d.h. eine nicht negative Zahl aus [0, {PDAT_MAX}].

Ergebnis

n	dup2 hat *n* als Synonym für *dk* eingerichtet
–1	dup2 hat kein Synonym eingerichtet, da – *dk* oder *n* keine gültige Dateikennzahl ist oder – bereits {PDAT_MAX} Dateien geöffnet sind.

Beispiel

Dateikennzahl 1 für Standardausgabe wird mit *datei* verbunden:

```
#include <stdio.h>

main()
{
  int dk, c;
              /* datei zum Schreiben anlegen */
  if((dk = creat("datei",0600)) == -1)
    printf("Fehler beim Anlegen von  datei\n");
              /* Standardausgabe mit datei verbinden */
  dup2(dk,1);
              /* Dateikennzahl 1 für Standardausgabe
                 zeigt jetzt auf datei
                 dk wird nicht mehr benötigt */
  close(dk);
              /* Ausgabe mit putchar auf Standardausgabe,
                 d.h. jetzt auf datei */
  while((c=getchar()) != EOF)
       putchar(c);
}
```

> > > > dup, creat, open, close, pipe, fcntl

Umwandlung in ASCII für die Ausgabe

char *ecvt(wert,anz,dez_pkt,vorzeichen**)**
double wert;
int anz, *dez_pkt, *vorzeichen;

ecvt wandelt einen internen Gleitkommawert in eine Zeichenreihe aus ASCII-Ziffern um und liefert als Ergebnis einen Zeiger auf diese Zeichenreihe.

Typ

C-Funktion

Parameter

double wert
Gleitkommawert, der für die Ausgabe aufbereitet werden soll.

int anz — Anzahl der Ziffern in der Ergebniszeichenreihe

← int *dez_pkt
Zeiger auf eine ganze Zahl, die die Position des Dezimalpunktes in der Ergebniszeichenreihe angibt:

positive Zahl
Position relativ zum Anfang der Zeichenreihe

negative Zahl
Dezimalpunkt steht vor der ersten Ziffer der Ergebniszeichenreihe.

← int *vorzeichen
Zeiger auf eine ganze Zahl, die das Vorzeichen der Ergebniszeichenreihe angibt:

0 — das Vorzeichen ist positiv

ungleich 0
das Vorzeichen ist negativ.

Ergebnis

Zeiger auf die umgewandelte ASCII-Zeichenreihe
bei Erfolg

Fehlermeldung

Falsche Parameter, etwa ein integer statt double Wert, führen zum Programmabbruch mit der Meldung: 'Speicherfehler - Speicherabzug(core) auf Platte geschrieben.'

Achtung

- ecvt schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Sie müssen dafür sorgen, daß die Ergebniszeiger *dez_pkt* und *vorzeichen* auf integer-Speicherplätze zeigen!

Hinweis

- Bei der Umwandlung wird die niedrigste Stelle gerundet.
- Die Standardausgabefunktion printf benützt ecvt, fcvt und gcvt.

Beispiel

Das Programm liest einen Gleitkommawert ein, wandelt ihn nach der Angabe in *anz* um und gibt ihn als ASCII-Zeichenreihe wieder aus. Zusätzlich wird das berechnete Vorzeichen ausgegeben.

```
#include <stdio.h>
char *ecvt();

main()
{
   double wert;
   int anz,dez_pkt,vorzeichen;
   printf("Bitte Gleitkommazahl eingeben:  ");
   if ( scanf("%lf",&wert) == 1)
      {
        printf("Wieviel signifikante Stellen :  ");
        if ( scanf("%d",&anz) == 1)
           {
             printf("Die Zahl lautet umgewandelt :  %s \n",
                     ecvt(wert,anz,&dez_pkt,&vorzeichen));
             printf("Das Vorzeichen ist %s \n",
                    (vorzeichen == 0 ? "positiv" | "negativ"));
           }
      }
}
```

>>>> fcvt, gcvt, printf, fprintf, sprintf

Ver- oder Entschlüsseln

void encrypt(feld, modus**)**
char *feld;
int modus;

encrypt ver- oder entschlüsselt das Binärfeld *feld*, je nachdem, ob *modus* 0 oder 1 ist.

Typ

C-Funktion

Parameter

← char *feld — Wie bei setkey geben Sie hier einen Vektor der Länge 64 an, der nur die Werte 0 und 1 enthält. Zum Ver- oder Entschlüsseln wird dann der mit setkey initialisierte DES Algorithmus verwendet.

int modus — Angabe, ob ver- oder entschlüsselt werden soll

0 — verschlüsseln des Vektors *feld*
1 — entschlüsseln des Vektors *feld*

Zum Entschlüsseln müssen Sie den DES Algorithmus mit dem Schlüssel initialisieren, der beim Verschlüsseln verwendet wurde.

Beispiel

Initialisierung des DES Algorithmus und anschließend Ver- und Entschlüsselung eines Binärfeldes:

```
#include <stdio.h>

char schl[] = {
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1  };

char feld[] = {
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,
     0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1  };

main()
{
   setkey(schl);
   pr();
   encrypt(feld,0);
   pr();
   encrypt(feld,1);
   pr();
}

pr()
{
   register int i;
   for (i=1; i<sizeof(feld); i++)
       printf("%d",feld[i]);
   putchar('\n');
}
```

> > > > setkey, crypt

"Gruppendatei" /etc/group schließen

int endgrent()

endgrent schließt die "Gruppendatei" /etc/group. Sie können endgrent im Anschluß an die Funktionen getgrent, getgrid und getgrnam verwenden.

Typ

C-Funktion

Parameter

keine

Beispiel

siehe Beispiel bei getgrent

Dateien

/etc/group	Liste der Benutzergruppen im System

> > > > getgrent, getgrgid, getgrnam, setgrent

"Passwortdatei" /etc/passwd schließen

int endpwent()

endpwent schließt die "Passwortdatei" /etc/passwd. Sie können endpwent im Anschluß an die Funktionen getpwent, getpwuid und getpwnam verwenden.

Typ

C-Funktion

Parameter

keine

Beispiel

siehe Beispiel bei getpwent

Dateien

/etc/passwd Liste aller Systembenutzer

>>>> getpwent, getpwuid, getpwnam, setpwent, getlogin

Die exec Systemaufrufe dienen alle dazu, ein neues Programm aufzurufen. Das neue Programm überlagert das aufrufende Programm.
Die einzelnen Möglichkeiten sind nachfolgend ausführlich beschrieben. Hier geben wir einen Überblick, der für alle exec Aufrufe zutrifft.

Das neue Programm steht in einer Programmdatei, die drei Teile hat:

- Dateikopf (Organisationsdaten)
- Textsegment (ausführbarer Code)
- Datensegment

Die Ausführung eines C Programms beginnt mit dem Aufruf von:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

Die drei Parameter, die der main-Funktion zur Verfügung stehen, werden entweder automatisch bei Programmaufruf oder explizit vom aufrufenden Programm mit aktuellen Werten versorgt.

Dabei bedeutet:

int argc — Anzahl der Argumente aus der Aufrufzeile
argc hat mindestens den Wert 1, weil der Programmname als erstes Argument zählt.
argc wird automatisch berechnet.

char **argv — Vektor von Zeigern auf die einzelnen Argumente, mit denen das Programm aufgerufen wurde
Ein Argument ist eine C-Zeichenreihe d.h. ein Vektor von Zeichen, dessen letztes Element das Nullbyte (\0) ist.

char **envp

Vektor von Zeigern auf die Umgebungsvariablen
Bei execl, execlp, execv und execvp wird vom C-Laufzeitsystem bei Programmstart automatisch ein Zeiger auf den globalen Vektor

extern char ** environ

übergeben.

Ein erfolgreicher exec Aufruf kehrt nicht zurück.
Wir heben nochmals hervor, daß mit exec kein neuer Prozeß erzeugt wird, sondern innerhalb eines fortbestehenden Prozesses lediglich ein Programm durch ein anderes überlagert wird! Das hat zur Folge, daß Text- und Datensegmente ausgetauscht werden, die restliche Prozeßumgebung aber fast vollständig erhalten bleibt. Wie diese Umgebung im einzelnen aussieht, zeigt die Tabelle auf der nächsten Seite. Eine Größe aus der Prozeßumgebung nennen wir "Prozeßkenndatum".

Die folgende Tabelle zeigt, wie die Prozeßkenndaten zustandekommen, die die aktuelle Prozeßumgebung des aufgerufenen Programms bilden:

Prozeßkenndaten:	Wert bei exec Aufruf
Prozeßnummer	ü
Prozeßnummer des Vaters	ü
Prozeßgruppennummer	ü
Datensichtstationsnummer	ü
Prozeßzeiten	ü (siehe times)
Prozeßpriorität	ü (siehe nice)
reale Benutzernummer	ü
reale Gruppennummer	ü
effektive Benutzernummer	wird neu gesetzt, falls bei der neuen Programmdatei das s-Bit für Eigentümer gesetzt ist
effektive Gruppennummer	wird neu gesetzt, falls bei der neuen Programmdatei das s-Bit für Gruppe gesetzt ist
Restzeit in der Alarmuhr	ü (siehe alarm, sleep)
Aktuelles Dateiverzeichnis	ü
Root Dateiverzeichnis	ü
Prozeßmaske	ü (siehe umask)
Maximale Dateigröße	ü (siehe ulimit)
offene Dateien	bleiben offen, falls ihr sbe-Bit auf Standard (0) gesetzt ist (siehe fcntl)
Trace Flag	ü (siehe ptrace, *anfrage* 0)
Zeit-Auswertung	wird abgeschaltet (siehe profil)
ignorierte Signale	ü
abgefangene Signale	werden auf Standardbehandlung (SIG_DFL) zurückgesetzt (siehe signal)

ü heißt: das entsprechende Prozeßkenndatum wird übernommen

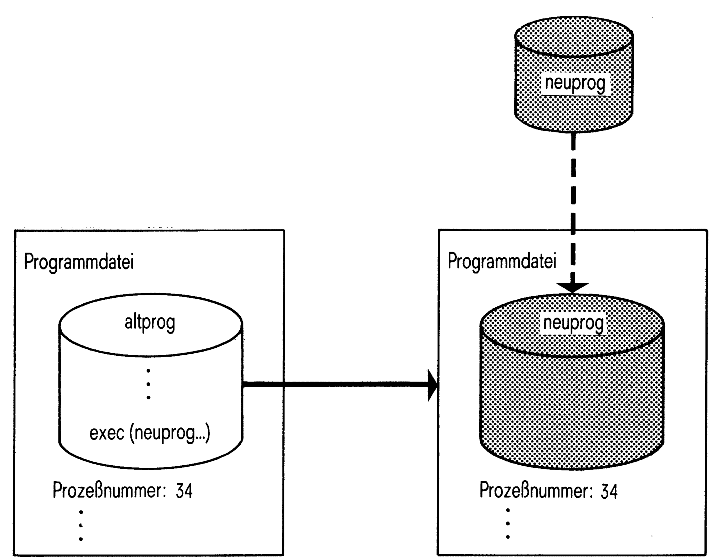


Bild 2-4 Wirkung eines exec Aufrufes

Es gibt sechs exec Systemaufrufe, deren Unterscheidungsmerkmale wieder in einer Tabelle zusammengestellt sind:

Art der Unterscheidung	Möglichkeiten	bei Systemaufruf
Anzahl der Argumente, mit denen das neue Programm aufgerufen wird	fest	execl, execle, execlp
	variabel	execv, execve, execvp
Umgebungsvariablen	werden übernommen	execl, execlp, execv, execvp
	können beim Aufruf übergeben werden	execle, execve
Suche nach der Programmdatei	findet nicht statt	execl, execle, execv, execve
	wird durchgeführt wie bei der Shell	execlp, execvp

In den Einzelbeschreibungen stehen am Anfang die Möglichkeiten des jeweiligen exec Aufrufes.

Programmaufruf:

- Anzahl der Argumente fest
- Umgebungsvariablen werden übernommen
- Programmdatei wird nicht gesucht

int execl(pfad, arg0, arg1, ..., argn, **NULL)**;
char *pfad, *arg0, *arg1, ..., *argn;

execl ruft das Programm in der Datei mit Pfadnamen *pfad* auf.
Das aufrufende Programm wird überlagert.
Nur im Fehlerfall ist eine Rückkehr ins aufrufende Programm möglich.

Typ

Systemaufruf

Parameter

char *pfad — voller Pfadname der Programmdatei, die ausgeführt werden soll. Der Pfadname muß stimmen, da die Datei nicht in einem Dateiverzeichnis gesucht wird.

char *arg0 — hier können Sie eine genauere Programmangabe machen, z.B. bewirkt der Aufruf
execl("/bin/date","datum",NULL), daß das Datum in Deutsch statt in Englisch ausgegeben wird.
Bei Programmen, die nicht näher spezifiziert werden können, ist dieser Parameter ohne Bedeutung. Üblicherweise gibt man nochmals die letzte Komponente des Pfadnamens an, wie etwa:
execl("/bin/cat","cat",...,NULL)

char *arg1,...,*argn
Argumente für das aufgerufene Programm

Ergebnis

Nur im Fehlerfall liefert execl ein Ergebnis:

–1

- eine Komponente des Pfadnamens existiert nicht oder
- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder
- ein Dateiverzeichnis auf dem Pfad zur Programmdatei darf nicht durchsucht werden oder die Programmdatei hat keine Ausführberechtigung oder die Programmdatei hat nicht das richtige Format (gültige magic number) oder
- es ist nicht genügend Speicher vorhanden oder
- die Argumentliste ist größer als die zulässige Systemkonstante {ARG_MAX}

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT :	Datei oder Dateiverzeichnis unbekannt
ENOTDIR :	Kein Dateiverzeichnis
EACCES :	Zugriff untersagt
ENOMEM :	Arbeitsspeicher unzureichend
E2BIG :	Liste der Argumente zu lang

Hinweis

Wenn das durch execl aufgerufene Programm bei main(argc,argv) startet, entsprechen sich die Argumente im exec Aufruf und in *argv* wie folgt:

arg0 <--> argv[0]
(Programmname)

arg1 <--> argv[1]
(1.Argument)

usw.

Beispiel

Der Aufruf execl("/bin/ls","ll","dv1","dv2",NULL) bewirkt, daß die Dateiverzeichnisse *dv1* und *dv2* wie bei ls -l aufgelistet werden:

```
#include <stdio.h>

main()
{
  execl("/bin/ls","ll","dv1","dv2",NULL);
}
```

Externe Größen

extern char **environ
Feld, in dem die Umgebungsvariablen definiert sind

> > > > execle, execlp, execv, execve, execvp, fork, alarm, exit, nice, profil, ptrace, signal, times, ulimit, umask

Programmaufruf:

- Anzahl der Argumente fest
- Umgebungsvariablen neu festlegen
- Programmdatei wird nicht gesucht

int execle(pfad, arg0, ..., argn, **NULL**, u_zg**)**
char *pfad, *arg0, *arg1, ..., *argn, *u_zg[];

execle ruft das Programm in der Datei mit Pfadnamen *pfad* auf.
Das aufrufende Programm wird überlagert.
Nur im Fehlerfall ist eine Rückkehr ins aufrufende Programm möglich.
Im Unterschied zu execl haben Sie hier die Möglichkeit, über den Parameter *u_zg* die Umgebungsvariablen neu zu definieren.

Typ

Systemaufruf

Parameter

char *pfad
: Voller Pfadname der Programmdatei, die ausgeführt werden soll. Der Pfadname muß stimmen, da die Datei nicht in einem Verzeichnis gesucht wird.

char *arg0
: Hier können Sie eine genauere Programmangabe machen; z.B. bewirkt der Aufruf execle("/bin/date","datum",...), daß das Datum in Deutsch statt in Englisch ausgegeben wird.
Bei Programmen, die nicht näher spezifiziert werden können, hat dieser Parameter keine Bedeutung. Üblicherweise gibt man die letzte Komponente des Pfadnamens nochmals an.

char *arg1,...,*argn
: Argumente für das Programm

char *u_zg[]
: Zeiger auf ein Feld von Zeichenreihen, die die Umgebungsvariablen des Prozesses definieren (siehe unter Hinweis)

Ergebnis

Nur im Fehlerfall liefert execle ein Ergebnis:

-1
- eine Komponente des Pfadnamens existiert nicht oder
- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder
- ein Dateiverzeichnis auf dem Pfad zur Programmdatei darf nicht durchsucht werden oder die Programmdatei hat keine Ausführberechtigung oder die Programmdatei hat nicht das richtige Format (gültige magic number) oder
- es ist nicht genügend Speicher vorhanden oder
- die Argumentliste ist größer als die zulässige Systemkonstante {ARG_MAX}

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT :	Datei oder Dateiverzeichnis unbekannt
ENOTDIR :	Kein Dateiverzeichnis
EACCES :	Zugriff untersagt
ENOMEM :	Arbeitsspeicher unzureichend
E2BIG :	Liste der Argumente zu lang

Hinweis

Im Unterschied zu execl können Sie bei execle Umgebungsvariablen neu definieren. *u_zg* ist ein Zeiger auf ein Feld von Zeichenreihen der Form:

name=wert,

dabei gelten dieselben Konventionen wie bei der Definition von Shell-Variablen, nämlich:

- *name* muß mit einem Buchstaben beginnen
- *name* und *wert* werden ohne Leerzeichen mit '=' verbunden
- *wert* wird mit \0 abgeschlossen.

Vorsicht mit häufig exportierten Shell-Variablen wie MAIL, PS1, PS2, IFS!

Beispiel

```
#include <stdio.h>

extern char **environ;
               /* Bei Programmausführung wird eine Kopie
                  der Umgebungsvariablen in environ abgelegt */

main()
{
               /* PATH neu definieren */
  strcpy(environ[19], "PATH=:/usr/sissi");
  execle("/usr/xantype/bsp","bsp",NULL,environ);
}
```

Und hier ist das aufgerufene Beispielprogramm bsp in /usr/xantype:

```
#include <stdio.h>

main()
{
  system("date");
}
```

Ergebnis: Im aufgerufenen Beispielprogramm *bsp* kann date nicht mehr ausgeführt werden, da nach dem Umdefinieren von PATH die Shell das Programm date nicht mehr findet.

Externe Größen

extern char **environ
    Feld, in dem die Umgebungsvariablen definiert sind

> > > > execl, exclp, execv, execve, execvp, fork, alarm, exit, nice, profil, ptrace, signal, times, ulimit, umask

Programmaufruf:

- Anzahl der Argumente fest
- Umgebungsvariablen werden übernommen
- Programmdatei wird gesucht

int execlp(name, arg0, arg1, ..., argn, **NULL)**
char *name, *arg0, *arg1, ..., *argn;

execlp ruft das Programm in der Programmdatei *name* auf.
execlp ist identisch zu execl außer, daß wie bei der Shell die auszuführende Programmdatei gesucht wird und zwar in den Dateiverzeichnissen, die in der Shell-Variablen PATH definiert sind.

Typ

Systemaufruf

Parameter

char *name — Name der Programmdatei, die ausgeführt werden soll. execlp sucht die Datei *name* wie die Shell in den Dateiverzeichnissen, die in der Umgebungsvariablen PATH angegeben sind.

char *arg0,...,*argn
Argumente für das aufgerufene Programm

Ergebnis

Nur im Fehlerfall liefert execlp ein Ergebnis:

–1
- eine Komponente des Pfadnamens existiert nicht oder
- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder

- ein Dateiverzeichnis auf dem Pfad zur Programmdatei darf nicht durchsucht werden oder die Programmdatei hat keine Ausführberechtigung oder die Programmdatei hat nicht das richtige Format (gültige magic number) oder
- es ist nicht genügend Speicher vorhanden oder
- die Argumentliste ist größer als die zulässige Systemkonstante {ARG_MAX}

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT :	Datei oder Dateiverzeichnis unbekannt
ENOTDIR :	Kein Dateiverzeichnis
EACCES :	Zugriff untersagt
ENOMEM :	Arbeitsspeicher unzureichend
E2BIG :	Liste der Argumente zu lang

Beispiel

Dateiinhalte von *datei1* und *datei2* ausgeben:
Das Kommando cat wird in den Dateiverzeichnissen gesucht, die in der Umgebungsvariablen PATH angegeben sind. Ist das Kommando cat vorhanden, werden *datei1* und *datei2* ausgegeben, vorausgesetzt die Dateien existieren im aktuellen Dateiverzeichnis.

```
#include <stdio.h>

main()
{
  execlp("cat","cat","datei1","datei2",NULL);
}
```

Externe Größen

extern char **environ
Feld, in dem die Umgebungsvariablen definiert sind

>>>> execl, execle, execv, execve, execvp, fork, alarm, exit, nice, profil, ptrace, signal, times, ulimit, umask

Programmaufruf:

- Anzahl der Argumente variabel
- Umgebungsvariablen werden übernommen
- Programmdatei wird nicht gesucht

int execv(pfad,argv**)**
char *pfad, *argv[];

execv ruft das Programm in der Datei mit Pfadname *pfad* auf.
Das aufrufende Programm wird überlagert.
Nur im Fehlerfall ist eine Rückkehr ins aufrufende Programm möglich.
Im Unterschied zu execl kann bei execv die Anzahl der Argumente variieren.

Typ

Systemaufruf

Parameter

char *pfad

Voller Pfadname der Programmdatei, die ausgeführt werden soll. Der Pfadname muß stimmen, da die Datei nicht gesucht wird.

char *argv[]

Zeiger auf den Argumentenvektor; *argv[0]* kann wie *arg0* bei execl eine nähere Programmspezifikation sein.
Das letzte Argument muß der Nullzeiger sein.

Ergebnis

Nur im Fehlerfall liefert execv ein Ergebnis:

–1
- eine Komponente des Pfadnamens existiert nicht oder
- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder
- ein Dateiverzeichnis auf dem Pfad zur Programmdatei darf nicht durchsucht werden oder die Programmdatei hat keine Ausführberechtigung oder die Programmdatei hat nicht das richtige Format (gültige magic number) oder
- es ist nicht genügend Speicher vorhanden oder
- die Argumentliste ist größer als die zulässige Systemkonstante {ARG_MAX}

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT :	Datei oder Dateiverzeichnis unbekannt
ENOTDIR :	Kein Dateiverzeichnis
EACCES :	Zugriff untersagt
ENOMEM :	Arbeitsspeicher unzureichend
E2BIG :	Liste der Argumente zu lang

Hinweis

Die Verwendung von execv ist dann sinnvoll, wenn die Anzahl der Argumente bei verschiedenen Aufrufen wechseln kann oder im voraus nicht bekannt ist.

Beispiel

Dieses Programm ruft das Kommando lpr aus der Position des Systemverwalters auf. Jeder Benutzer kann über diese Funktion die lpr-Funktionen des Systemverwalters (Schalter -dk, -du) ausführen. Das übersetzte Programm muß dem Systemverwalter gehören und es muß das s-Bit gesetzt sein.

```
main(argc,argv)
int argc;
char **argv;
{
  setuid(0);    /* reale Benutzernummer auf die des System-
                   verwalters setzen *
  execv("/bin/lpr",argv);

                   Meldung bei Mißerfolg */
  printf("Aufruf nicht erfolgreich\n");
}
```

Externe Größen

extern char **environ
: Feld, in dem die Umgebungsvariablen definiert sind

> > > > execl, execle, execlp, execve, execvp, fork, alarm, exit, nice, profil, ptrace, signal, times, ulimit, umask

Programmaufruf:

- Anzahl der Argumente variabel
- Umgebungsvariablen neu festlegen
- Programmdatei wird nicht gesucht

int execve(pfad,argv,u_zg**)**
char *pfad, *argv[], *u_zg[];

execve ruft das Programm in der Datei mit Pfadname *pfad* auf.
Das aufrufende Programm wird überlagert.
Nur im Fehlerfall ist eine Rückkehr ins aufrufende Programm möglich.
execve macht dasselbe wie execle für eine variable Anzahl von Argumenten. Im Unterschied zu execv haben Sie hier die Möglichkeit, über den Parameter *u_zg* Umgebungsvariablen neu zu definieren.

Typ

Systemaufruf

Parameter

char *pfad
: Voller Pfadname der Programmdatei, die ausgeführt werden soll. Der Pfadname muß stimmen, da die Datei nicht gesucht wird.

char *argv[]
: Zeiger auf den Argumentenvektor; *argv[0]* kann wie *arg0* bei execl eine nähere Programmspezifikation sein.
Das letzte Argument muß der Nullzeiger sein.

char *u_zg[]
: Zeiger auf ein Feld, das die Umgebungsvariablen enthält (siehe Hinweis bei execle).

Ergebnis

Nur im Fehlerfall liefert execve ein Ergebnis:

-1
- eine Komponente des Pfadnamens existiert nicht oder
- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder
- ein Dateiverzeichnis auf dem Pfad zur Programmdatei darf nicht durchsucht werden oder die Programmdatei hat keine Ausführberechtigung oder die Programmdatei hat nicht das richtige Format (gültige magic number) oder
- es ist nicht genügend Speicher vorhanden oder
- die Argumentliste ist größer als die zulässige Systemkonstante {ARG_MAX}

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT :	Datei oder Dateiverzeichnis unbekannt
ENOTDIR :	Kein Dateiverzeichnis
EACCES :	Zugriff untersagt
ENOMEM :	Arbeitsspeicher unzureichend
E2BIG :	Liste der Argumente zu lang

Externe Größen

extern char **environ

Feld, in dem die Umgebungsvariablen definiert sind

> > > > execl, execle, execlp, execv, execvp, fork, alarm, exit, nice, profil, ptrace, signal, times, ulimit, umask

Programmaufruf:

- Anzahl der Parameter variabel
- Umgebungsvariablen werden übernommen
- Programmdatei wird gesucht

int execvp(name,argv**)**
char *name, *argv**[]**;

execvp ruft das Programm in der Programmdatei *name* auf.
execvp ist identisch zu execv außer, daß wie bei execlp die auszuführende Programmdatei in den Dateiverzeichnissen gesucht wird, die in der Umgebungsvariablen PATH angegeben sind.

Typ

Systemaufruf

Parameter

char *name
: Name der Programmdatei
Hier müssen Sie nicht den vollen Pfadnamen angeben; execvp sucht wie die Shell in den Dateiverzeichnissen, die in PATH angegeben sind, nach der Datei *name*.

char *argv[]
: Zeiger auf den Argumentenvektor

Ergebnis

Nur im Fehlerfall liefert execvp ein Ergebnis:

-1
- eine Komponente des Pfadnamens existiert nicht oder
- eine Komponente des Pfadnamens ist kein Dateiverzeichnis oder
- ein Dateiverzeichnis auf dem Pfad zur Programmdatei darf nicht durchsucht werden die Programmdatei hat keine Ausführberechtigung oder die Programmdatei hat nicht das richtige Format (gültige magic number) oder
- es ist nicht genügend Speicher vorhanden oder
- die Argumentliste ist größer als die zulässige Systemkonstante {ARG_MAX}

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOENT : Datei oder Dateiverzeichnis unbekannt
ENOTDIR : Kein Dateiverzeichnis
EACCES : Zugriff untersagt
ENOMEM : Arbeitsspeicher unzureichend
E2BIG : Liste der Argumente zu lang

Externe Größen

extern char **environ
Feld, das die Umgebungsvariablen enthält

> > > > execl, execle, execlp, execv, execve, fork, alarm, exit, nice, profil, ptrace, signal, times, ulimit, umask

Prozeßbeendigung

void _exit(status)
int status;

_exit beendet das aufrufende Programm sofort, ohne die Ausgabepuffer zu leeren. Ansonsten funktioniert _exit wie exit.

Typ

Systemaufruf

Parameter

int status Endestatus; ein Vaterprozeß, der auf die Beendigung des Sohnprozesses wartet (wait), erhält nach dem exit Aufruf die niedrigeren 8 Bits von *status* und kann diese als Endestatus auswerten.

status:

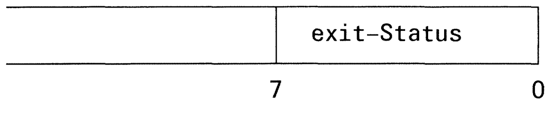


Bild 2-5 exit-Status

Hinweis

Nur bei exit werden angebrochene Ausgabepuffer noch korrekt und vollständig ausgegeben. Der Systemaufruf _exit macht keinerlei Bereinigung.

>>>> exit, signal, wait

Prozeßbeendigung

void exit(status)
int status;

exit beendet den aufrufenden Prozeß und führt dabei folgende Tätigkeiten durch:

- Daten, die noch in Ausgabepuffern stehen, werden ausgegeben.
- Alle geöffneten Dateien werden geschlossen.
- Wenn der Vaterprozeß auf die Beendigung wartet (auch wenn er wait erst zu einem späteren Zeitpunkt aufruft), erfährt er vom Ende des Sohnes und erhält die niedrigeren 8 Bits (also die Bits 0377) von *status*.
- Wenn der Vaterprozeß nicht wartet, endet der Sohn als (inaktiver) Zombieprozeß und wird irgendwann zu einem späteren Zeitpunkt entfernt.
- Alle Sohn- und Zombieprozesse des aufrufenden Prozesses werden von einem speziellen Systemprozeß übernommen (die Prozeßnummer des speziellen Systemprozesses wird Vaterprozeßnummer).
- Wenn der aufrufende Prozeß Bereiche gesperrt hält, werden diese wieder freigegeben.
- Der erste Prozeß, der einen Bildschirm eröffnet hat (i.a. die Login Shell), ist der Prozeßgruppenchef dieses Bildschirms. Wenn sich dieser Prozeß beendet, werden alle Prozesse aus der gleichen Prozeßgruppe mit dem Signal SIGHUP benachrichtigt.

Typ

C-Funktion

Parameter

int status   Endestatus; ein Vaterprozeß, der auf die Beendigung des Sohnprozesses wartet (wait), erhält nach dem exit Aufruf die niedrigeren 8 Bits von *status* und kann diese als Endestatus auswerten.

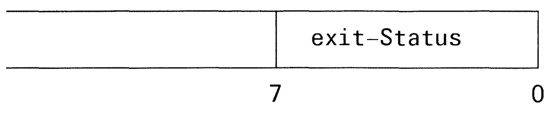


Bild 2-6   exit-Status

Hinweis

- Nur bei exit werden angebrochene Ausgabepuffer noch korrekt und vollständig ausgegeben. Dazu gibt es in der Bibliothek libc.a eine Funktion _cleanup, die Sie bei Bedarf auch selbst schreiben können. Der Systemaufruf _exit macht keinerlei Bereinigung.
- Immer, wenn ein Programm normal endet, wird automatisch exit(0) ausgeführt.

Beispiel

siehe u.a. Beispiel bei fdopen, fgets, fopen, fork

> > > > wait, signal, _exit

Exponentialfunktion

#include <math.h>

double exp(x)
double x;

exp berechnet die Exponentialfunktion für zulässige Gleitkommazahlen.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl

Ergebnis

e**x	falls *x* und das Ergebnis im zulässigen Gleitkommaintervall liegen
HUGE	Überlauf

Fehlermeldung

Bei Überlauf wird errno besetzt mit dem Fehlercode:

ERANGE : Argument zu groß

Hinweis

Wenn Sie in Ihrem Programm exp verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Berechne e**x für einen eingelesenen Wert *x*:

```
#include <math.h>
#include <stdio.h>

main()
{
  double x;
  if ( scanf("%lf",&x) == 1)
      printf("exp(%g) = %g\n",x,exp(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > log, log10, pow

Absolutbetrag einer Gleitkommazahl

```
#include <math.h>

double fabs(x)
double x;
```

fabs berechnet den Absolutbetrag einer Gleitkommazahl.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl, deren Absolutbetrag berechnet werden soll

Ergebnis

$\|x\|$	falls *x* und das Ergebnis im zulässigen Gleitkommaintervall liegen

Hinweis

Wenn Sie fabs verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Absolutbetrag einer eingelesenen Gleitkommazahl berechnen:

```
#include <math.h>
#include <stdio.h>

main()
{
   double x;
   if ( scanf("%lf",&x) == 1)
       printf("|%g| = %g\n",x,fabs(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

>>>> floor, ceil, abs, cabs

Datei schließen

#include <stdio.h>

int fclose(dz)
FILE *dz;

fclose schließt die Datei, auf deren FILE-Struktur der Dateizeiger *dz* zeigt und gibt *dz* frei. Speicherplatz, der für diese FILE-Struktur dynamisch (bei fopen) angelegt wurde, wird freigegeben. fclose ruft fflush auf bevor die Datei geschlossen wird.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger für die Datei, die geschlossen werden soll

Ergebnis

0	Die Datei wurde geschlossen
EOF	fclose war nicht erfolgreich, weil – *dz* keiner Datei zugeordnet ist oder – beim Leeren des Puffers ein Fehler auftrat

Fehlermeldung

Wenn der Dateizeiger *dz* nicht auf eine Struktur FILE zeigt, bricht das Programm mit der Meldung ab:
'Speicherfehler - Speicherabzug(core) auf Platte geschrieben'

Hinweis

Immer wenn ein Programm normal endet, wird automatisch für jede offene Datei fclose ausgeführt. Sie brauchen fclose also nur dann explizit aufzurufen, wenn Sie vor Programmbeendigung eine Datei schließen wollen, damit z.B. nicht das Dateilimit überschritten wird.

Beispiel

Programmstück zum Schließen der Datei mit Dateizeiger *dz* bei Erreichen des Dateiendes:

```
FILE *dz;

if(feof(dz))
  fclose(dz);
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fflush, close, fopen, setbuf, exit

Kontrollfunktionen auf eine geöffnete Datei

#include <sys/fcntl.h>

int fcntl(dk,akt,arg**)**
int dk, akt, arg;

fcntl gibt Ihnen die Möglichkeit, mit Hilfe der durch *akt* ausgewählten Aktion den Status der (geöffneten) Datei mit Dateikennzahl *dk* abzufragen oder zu verändern.

Typ

Systemaufruf

Parameter

int dk — Dateikennzahl

int akt
int arg — Sie geben hier eine Konstante an, die die gewünschte Aktion (abfragen oder verändern) auswählt. *arg* ist in einigen Fällen aktueller Parameter für die Aktion. In <sys/fcntl.h> sind folgende Möglichkeiten definiert:

F_DUPFD
der Datei wird eine zusätzliche Dateikennzahl zugeordnet (vgl. dup, dup2) und zwar folgendermaßen:

- es wird die kleinste freie Dateikennzahl zugewiesen, die größer oder gleich *arg* ist
- die neue und alte Dateikennzahl bezeichnen dieselbe Datei und haben den gleichen Dateizeiger
- die Zugriffsart (lesen, schreiben oder beides) bleibt unverändert
- der Dateistatus bleibt unverändert
- das sbe-Bit wird auf Standard (0) gesetzt, d.h., die Datei bleibt nach einem exec Aufruf geöffnet.

Die restlichen Aktionen betreffen das Prozeß-Dateistatus-Byte (F_GETFD, F_SETFD) bzw. das System-Dateistatus-Byte (F_GETFL, F_SETFL) (siehe Hinweis).

F_GETFD

das Prozeß-Dateistatus-Byte abfragen
In diesem Byte ist lediglich das niedrigste Bit, das sbe-Bit definiert.

F_SETFD

das Prozeß-Dateistatus-Byte (d.h. das sbe-Bit) wird neu gesetzt und zwar auf das niedrigere Byte von *arg*

0 die Datei bleibt nach einem exec Aufruf offen
1 die Datei wird bei einem exec Aufruf geschlossen

F_GETFL

das System-Dateistatus-Byte abfragen (siehe Ergebnis)

F_SETFL

die Bits (O_NDELAY, O_APPEND, O_SYNCW) im System-Dateistatus-Byte werden auf die entsprechenden Bits im niedrigeren Byte von *arg* gesetzt.

Ergebnis

Bei Erfolg hängt das Ergebnis von der ausgewählten Aktion ab:

neue Dateikennzahl

falls die Aktion F_DUPFD gewählt wurde

Prozeß-Dateistatus-Byte

falls die Aktion F_GETFD gewählt wurde. Nur das niedrigere Byte ist definiert und darin lediglich das sbe-Bit.

ungleich -1

falls die Aktion F_SETFD gewählt wurde

System-Dateistatus-Byte

falls die Aktion F_GETFL gewählt wurde
Nur das niedrigere Byte ist definiert und darin die Bits O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY, O_APPEND und O_SYNCW.

ungleich -1

falls die Aktion F_SETFL gewählt wurde

-1	Fehler, wenn

- *dk* keine gültige Dateikennzahl ist oder
- bereits {PDAT_MAX} Dateien geöffnet sind (F_DUPFD) oder
- eine neue Dateikennzahl zugewiesen werden sollte, *arg* aber negativ oder größer als {PDAT_MAX} ist

Fehlermeldung

Bei Rückkehr mit Fehler steht in errno ein entsprechender Fehlercode:

EBDAF :	Unzulässige Dateikennzahl
EMFILE :	Zu viele offene Dateien im System
EINVAL :	Unzulässiges Argument

Hinweis

Für jede offene Datei gibt es zwei Bytes, die Statusinformation in Form von Bits enthalten:

- das Prozeß-Dateistatus-Byte (file descriptor flags) gehört dem Prozeß und ist für jede seiner offenen Dateien definiert.
  Derzeit gibt es darin lediglich ein gültiges Bit, das sbe-Bit (close-on-exec). Dieses Bit wird nur im exec Aufruf betrachtet und gibt dort an, ob die Datei geöffnet bleibt oder geschlossen wird.

Standardeinstellung	0 :	die Datei bleibt geöffnet
	1 :	die Datei wird bei exec geschlossen

  Sie können nur mit fcntl das sbe-Bit abfragen oder neu setzen.

- das System-Dateistatus-Byte (file flags) gehört dem System und wird für jede (logisch) geöffnete Datei im System neu angelegt, auch wenn mehrere Benutzer die gleiche Datei geöffnet haben oder ein Benutzer die gleiche Datei unter verschiedenen Dateikennzahlen. Bei jedem Zugriff (open, read, write) auf die Datei wird dieses Byte ausgewertet. Die darin enthaltenen Bits O_NDELAY und O_APPEND und O_SYNCW können mit fcntl verändert werden.
  Die Bits sind in <sys/fcntl.h> definiert und bedeuten:

  O_APPEND : beim Schreiben (write) werden die Daten an das Dateiende angehängt.

  O_NDELAY: Lesen (read) und Schreiben (write) ohne Blockierungsgefahr auf eine Pipe

  O_SYNCW : bei jedem write wird sofort physikalisch auf Platte geschrieben ohne System-interne Pufferung.

Dateien

/usr/include/sys/fcntl.h
: Definition der Aktionskonstanten und der Bits, die den Dateistatus angeben.

>>>> dup, dup2, exec, open, close

Umwandlung in ASCII Ziffern entsprechend Fortran F-Format

char *fcvt(wert,anz,dez_pkt,vorzeichen**)**
double wert;
int anz, *dez_pkt, *vorzeichen;

Wie ecvt konvertiert fcvt einen internen Gleitkommawert in eine Zeichenreihe aus ASCII-Ziffern. Dabei sorgt fcvt zusätzlich dafür, daß das Ausgabeformat dem Fortran F-Format entspricht.

Typ

C-Funktion

Parameter

double wert
: Gleitkommawert, der für die Ausgabe aufbereitet werden soll.

int anz
: Anzahl der Ziffern in der Ergebniszeichenreihe, fcvt rundet *wert* passend für das Fortran F-Format.

← int *dez_pkt
: Zeiger auf eine ganze Zahl, die die Position des Dezimalpunktes in der Ergebniszeichenreihe angibt:

positive Zahl
: Position relativ zum Anfang der Zeichenreihe

negative Zahl
: Dezimalpunkt steht vor der ersten Ziffer der Ergebniszeichenreihe

← int *vorzeichen
: Zeiger auf eine ganze Zahl, die das Vorzeichen der Ergebniszeichenreihe angibt:

0
: das Vorzeichen ist positiv

ungleich 0
: das Vorzeichen ist negativ

Ergebnis

Zeiger auf die umgewandelte ASCII-Zeichenreihe
bei Erfolg

Fehlermeldung

Falsche Parameter, etwa ein integer statt double Wert, führen zu Programmabbruch mit der Meldung:
'Speicherfehler - Speicherabzug(core) auf Platte geschrieben'

Achtung

- fcvt schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Sie müssen dafür sorgen, daß die Ergebniszeiger *dez_pkt* und *vorzeichen* auf einen integer-Speicherplatz verweisen!

Beispiel

siehe Beispiel bei ecvt

> > > > ecvt, gcvt, printf, fprintf, sprintf

Dateizeiger zuweisen

 # include <stdio.h>

FILE *fdopen(dk,art**)**
int dk;
char *art;

fdopen weist der bereits geöffnete Datei (mit Dateikennzahl *dk*) eine FILE-Struktur und einen Dateizeiger zu.

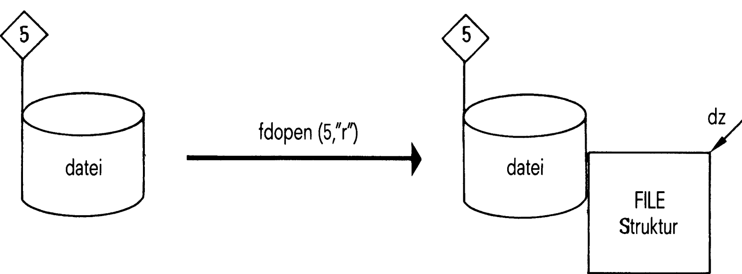


Bild 2-5 Wirkung eines fdopen Aufrufs

Die Dateikennzahl *dk* wurde von einem creat, dup, dup2, fcntl, open oder pipe Aufruf geliefert und wird von elementaren Zugriffsoperationen benützt. Nach einem fdopen Aufruf können Sie dann die Datei auch mit den Funktionen aus der Standardein/ausgabe Bibliothek bearbeiten.

Typ

C-Funktion (s)

Parameter

int dk	Dateikennzahl, die durch einen creat, dup, dup2, fcntl, open oder pipe Aufruf zugewiesen wurde
char *art	*art* gibt die gewünschte Zugriffsart an und kann eine der folgenden Zeichenreihen sein:
"r"	Datei öffnen zum Lesen
"w"	Datei öffnen zum Schreiben; neuer Text wird da eingefügt, wo der Lese/Schreibzeiger gerade steht (s. Beispiel unten).
"r+"	Datei öffnen zum Lesen und Schreiben
"a"	Datei öffnen, um Text ans Ende der Datei anzufügen

Ergebnis

Dateizeiger, zeigt auf die zugewiesene FILE-Struktur
bei Erfolg

undefiniert
falls ein Fehler, wie z.B. eine ungültige Dateikennzahl, vorliegt.
Treten Fehler auf, liefert fdopen weder ein wohldefiniertes Ergebnis noch eine Fehlermeldung und das Programm bricht auch nicht ab.

Hinweis

- Die Angabe in *art* muß mit dem bisherigen Zugriffsmodus für die geöffnete Datei übereinstimmen.
- Für die Dateikennzahlen 0, 1 und 2 liefert fdopen nicht notwendig die entsprechenden Dateizeiger stdin, stdout bzw. stderr!
  Diese Zuordnung liefert sicher die Funktion freopen.

Beispiel

Die Datei *dat* für elementare und Standard-Ein/Ausgabe öffnen:

```
#include <stdio.h>

FILE *fp, *fdopen();
int fd;
char buf[10];
int c;

main()
{
   int i;
   int zähler;

               /* zuerst mit Dateikennzahl arbeiten */
   if((fd = open("dat",2)) < 0);
     {
        perror("open");
        exit(1);
     }

   if((zähler = read(fd,buf,10)) > 0)
      write(1,buf,zähler);

               /* Dateizeiger mit Dateikennzahl verbinden */
   fp = fdopen(fd,"w");
   while((c = getchar()) != EOF)
        putc(c,fp);
   fclose(fp);
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > creat, dup, dup2, fclose, fseek, fcntl, fopen, freopen, open, pipe

Test auf Dateiende

#include <stdio.h>

int feof(dz)
FILE *dz;

feof erkennt das Ende der Datei mit Dateizeiger *dz*.

Typ

Makro (s)

Parameter

FILE *dz	Dateizeiger der Datei, die auf Dateiende abgeprüft werden soll

Ergebnis

ungleich 0	Dateiende
0	sonst

Hinweis

feof wird üblicherweise nach Zugriffsfunktionen angewendet, die keine Dateiende-Meldung machen (fread, fwrite).

Beispiel

Programmstück für zeichenweises Einlesen aus der Datei mit Dateizeiger *dz* bis Dateiende erreicht ist:

```
char name[10];
FILE *dz;
.
.
.
do
   fread(name, sizeof(name),1,dz);
while(!feof(dz));
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > clearerr, ferror, fileno, fopen, open

Test auf Dateifehler

#include <stdio.h>

int ferror(dz)
FILE * dz;

ferror überprüft, ob in der FILE-Struktur, auf die *dz* zeigt, das Fehlerflag gesetzt ist.

Typ

Makro (s)

Parameter

FILE *dz	Zeiger auf die FILE-Struktur, in der das Fehlerflag untersucht werden soll

Ergebnis

ungleich 0	das Fehlerflag ist gesetzt, weil während vorangegangener Lese- oder Schreibzugriffe ein Fehler auftrat
0	sonst

Hinweis

Wird eine Fehleranzeige nicht explizit mittels clearerr gelöscht, so bleibt sie bestehen bis der zugehörige Dateizeiger freigegeben wird (durch fclose oder Programmbeendigung).

Beispiel

Die Funktion ferror sollten Sie immer verwenden, bevor Sie eine Datei lesen oder beschreiben wollen.
In folgendem Beispiel wird vor jedem fread Aufruf abgeprüft, ob ein Fehler angezeigt ist auf der FILE-Struktur, auf die *dz* zeigt:

```
FILE *dz;
char buf[10];
char x[5];

while( !ferror(dz))
     fread(buf,sizeof(x),10,dz);

clearerr(dz);
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > clearerr, feof, fileno, fopen, open

Puffer leeren

#include <stdio.h>

int fflush(dz)
FILE *dz;

fflush leert den Puffer der Datei mit Dateizeiger *dz* und schreibt alles, was noch im Puffer steht, in die Datei.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger der Datei, deren Puffer geleert werden soll

Ergebnis

0	fflush hat den Puffer geleert.
EOF	fflush hat den Puffer nicht geleert, weil – der Zeiger *dz* keiner Datei zugeordnet ist oder – die gepufferten Daten nicht übertragen werden konnten.

Hinweis

- fclose ruft fflush auf, bevor die Datei geschlossen wird.
- Wenn ein Programm normal oder durch exit beendet wird, wird automatisch fflush aufgerufen.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

>>>> close, exit, fclose, fopen, setbuf

Zeichen einlesen

```
#include <stdio.h>

int fgetc(dz)
FILE *dz;
```

fgetc liest ein Zeichen aus der Datei, der der Dateizeiger *dz* zugeordnet ist.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger für Eingabedatei

Ergebnis

eingelesenes Zeichen als int
fgetc liefert bei Erfolg das eingelesene Zeichen als positiven integer-Wert

EOF — Fehler oder Dateiende

Hinweis

- fgetc verhält sich wie getc, ist aber eine Funktion und kein Makro.
- Wenn Sie in Ihrem Programm einen Vergleich wie etwa in
  while((c = fgetc(dz)) != EOF)
  verwenden, sollten Sie die Variable *c* immer als integer Größe vereinbaren. Wenn Sie *c* als char definieren, kann es sein, daß die Bedingung nie erfüllt ist, weil die Vorzeichenpropagierung bei der Umwandlung von char in int maschinenabhängig ist.

Beispiel

Folgendes Programm liest aus einer Reihe (maximal 10) beim Aufruf übergebener Dateien nacheinander jeweils ein Zeichen und gibt es auf Standardausgabe aus.

```
#include <stdio.h>

FILE *dz[10], **app;

main(argc, argv)
int argc;
char *argv[];
{
  int c,i;
  for (i=1; i<argc; i++)
       dz[i-1] = fopen(argv[i],"r");
  app = dz;
  while(*app != NULL)
       {
         c = fgetc(*app++);
         putchar(c);
       }
  putchar('\n');
}
```

Wir haben dieses (etwas gekünstelte) Beispiel gewählt, um eine Anwendung von fgetc zu zeigen, in der ausgenutzt wird, daß fgetc eine Funktion ist. Mit dem Makro getc würde der Aufruf getc(*app + +) nicht funktionieren.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > getc, getchar, getw, gets, fopen, fscanf, fread, ungetc

Zeichenreihe aus einer Datei einlesen

#include <stdio.h>

char *fgets(s,n,dz**)**
char *s;
int n;
FILE *dz;

fgets liest aus der Datei mit Dateizeiger *dz* entweder n-1 Zeichen oder bis einschließlich zum nächsten Zeilenende oder bis Dateiende, falls dies vorher eintrifft. Die eingelesenen Zeichen trägt fgets in den Bereich ein, auf den *s* zeigt.

Typ

C-Funktion (s)

Parameter

← char *s	Zeiger für die Ergebniszeichenreihe
int n	maximale Länge der Ergebniszeichenreihe
FILE *dz	Dateizeiger der Eingabedatei

Ergebnis

eingelesene Zeichenreihe (Zeiger auf das erste gelesene Zeichen)
fgets schließt die eingelesene Zeichenreihe mit '\0' ab

Nullzeiger — fgets hat nichts eingelesen, weil

- Dateiende erreicht ist oder
- ein Fehler beim Lesen auftrat

Achtung

Den Bereich, in den fgets die gelesene Zeichenreihe abspeichern soll, müssen Sie explizit bereitstellen! (siehe Beispiel)

Hinweis

Im Unterschied zu gets trägt fgets auch ein gelesenes Zeilenende-Zeichen in die Ergebniszeichenreihe ein.

Beispiel

fgets wird verwendet, um aus einer Datei zeilenweise einzulesen.
Dieses Beispiel zeigt die unterschiedliche Behandlung von Zeilenwechsel bei fgets und gets:

```
              /* Inhalt von datei:
                 Reden ist Silber
                 Schreien ist Gold */

#include <stdio.h>

char *fgets();
char *gets();

main()
{
   FILE *dz;
   int i = 17;
   char t[20], r[20];
   char s[BUFSIZ];

   if((dz = fopen("datei","r")) == NULL)
     {
        perror("fopen");
        exit(10);
     }

   fgets(t,i,dz);
   printf("%s",t);
   fgets(r,i,dz);
   printf("%s",r);
              /* Eingabe von Standardeingabe */
   while(gets(s)) != NULL)
        printf("%s",s);
 }
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > getc, gets, ,feof, fopen, fscanf, fgetc, scanf, sscanf

Dateikennzahl abfragen

#include <stdio.h>

int fileno(dz)
FILE *dz;

fileno gibt die Dateikennzahl der Datei mit Dateizeiger *dz* aus.

Typ

Makro (s)

Parameter

File *dz	Dateizeiger für die Datei, deren Dateikennzahl ausgegeben werden soll.

Ergebnis

gültige Dateikennzahl
falls der Dateizeiger *dz* auf eine Datei zeigt, auf die das Programm Zugriff hat.

undefiniert
sonst

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

>>>> open, creat, fopen

Abrunden

#include <math.h>

double floor(x**)**
double x;

floor rundet eine Gleitkommazahl nach unten ganzzahlig ab.

Typ

C-Funktion

Parameter

double x — Gleitkommazahl, die abgerundet werden soll.

Ergebnis

größte ganze Zahl, die kleiner oder gleich *x* ist.

Hinweis

Wenn Sie in Ihrem Programm floor verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

siehe Beispiel bei ceil

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > ceil

Datei öffnen

#include <stdio.h>

FILE *fopen(d_name,art)
char *d_name, *art;

fopen öffnet die Datei *d_name* und weist ihr eine FILE-Struktur und einen Dateizeiger zu. Der Dateizeiger zeigt auf die zugewiesene FILE-Struktur.
Die FILE-Struktur ist in der Datei <stdio.h> definiert. Sie enthält notwendige Information für die meisten Funktionen aus der Standardein/ausgabe Bibliothek.

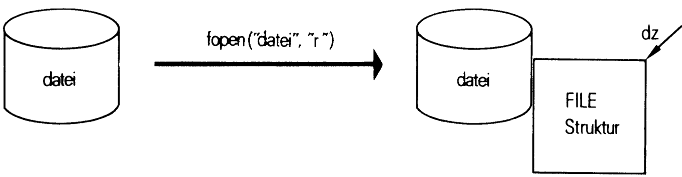


Bild 2-6 Wirkung eines fopen Aufrufs

Typ

C-Funktion (s)

Parameter

char *d_name
Name der Datei, die geöffnet werden soll

char *art — art gibt die gewünschte Zugriffsart an und kann eine der folgenden Zeichenreihen sein:

"r" — Datei öffnen zum Lesen
"w" — Datei öffnen zum Neubeschreiben
"a" — Datei öffnen, um Text ans Ende der Datei anzufügen.

Ergebnis

Zeiger auf die zugewiesene FILE-Struktur
bei Erfolg

Nullzeiger — fopen kann *d_name* nicht öffnen, weil

- das Programm nicht die nötige Zugriffsberechtigung hat oder
- ein Fehler aufgetreten ist (siehe creat)

Hinweis

- Wenn ein Programm startet, werden ihm automatisch drei Dateizeiger für Standardeingabe, -ausgabe und -fehlerausgabe zugeordnet und zwar:

  stdin : Dateizeiger für Standardeingabe (i.a. Datensichtstation)
  stdout : Dateizeiger für Standardausgabe (i.a. Datensichtstation)
  stderr : Dateizeiger für Standardfehlerausgabe (i.a. Datensichtstation)

- Wenn Sie eine Datei zum Schreiben öffnen, wird der alte Inhalt der Datei gelöscht; wenn Sie die Datei dagegen mit Modus "a" öffnen, bleibt der alte Inhalt erhalten und der neue Text wird ans Ende der Datei angehängt.

- Der Versuch, eine Datei, die es noch nicht gibt, zum Lesen zu öffnen, endet mit Fehler; ebenso der Versuch, eine Datei zu eröffnen, für die das Programm nicht die entsprechende Zugriffsberechtigung hat. Wenn Sie eine Datei, die es noch nicht gibt, zum Schreiben öffnen, wird die Datei neu erstellt.

- Sie können eine Datei gleichzeitig für verschiedene Zugriffsmodi eröffnen. Das folgende Beispiel zeigt, wie die Datei "/usr/pups" gleichzeitg zum Lesen und Schreiben geöffnet wird:

```
FILE *lp,*sp;
lp = fopen("/usr/pups","r");
sp = fopen("/usr/pups","a");
```

  Vorsicht mit dem Zugriffsmodus "w"!!

Beispiel

```
              /* Programm zum Kopieren von
                 datei1 und datei2 auf datei3 */

#include <stdio.h>

FILE *dp_l, *dp_s;

main()
{
   if((dp_l = fopen("datei1","r")) == NULL || (dp_s = fopen("datei3","w")) ==NULL)

      {
              /* Programmabbruch bei Fehler mit
                 Rückgabewert 1 */
        perror("fopen");
        exit(1);
      }

   copy();
              /* Umhängen des Dateizeigers von datei1
                 auf datei2 */

   if((freopen("datei2","r",dp_l)) == NULL)

              /* Programmabbruch bei Fehler mit
                 Rückgabewert 2 */
     exit(2);

   copy();
   fclose(dp_l);
   fclose(dp_s);
}

copy()
{
   int c;
   while((c = getc(dp_l)) != EOF)
        putc(c,dp_s);
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > creat, fdopen, freopen, ferror, open, fclose

Neuen Prozeß erzeugen

int fork();

fork verdoppelt den aufrufenden Prozeß und erzeugt damit zwei unabhängige Prozesse, genannt Vater- und Sohnprozeß.

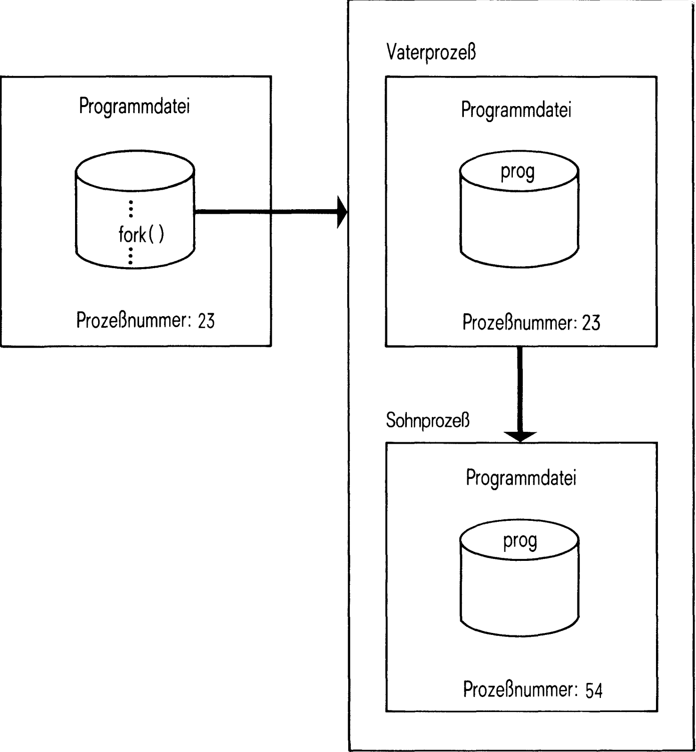


Bild 2-7 Wirkung eines fork Aufrufs

Der (neue) Sohnprozeß ist bis auf wenige Ausnahmen eine exakte Kopie des (ursprünglichen) Vaterprozesses. Wie die Prozeßkenndaten des Sohnes im Einzelnen zustandekommen, zeigt folgende Übersicht:

Der Sohn übernimmt vom Vater:

- die Umgebungsvariablen (siehe exec)
- das sbe-Bit (siehe fcntl)
- die vom Vater festgelegten Signalbehandlungen (SIG_DFL, SIG_IGN oder eine Funktion, siehe signal)
- s-Bit für Eigentümer
- s-Bit für Gruppe
- reale Benutzernummer
- reale Gruppennummer
- effektive Benutzernummer
- effektive Gruppennummer
- Prozeßpriorität (siehe nice)
- Datensichtstationsnummer (siehe signal)
- Prozeßgruppennummer (siehe setpgrp)
- Trace Flag (siehe ptrace)
- Zeitauswertung (ja oder nein, siehe profil)
- Aktuelles Dateiverzeichnis
- Root Dateiverzeichnis
- Prozeßmaske (siehe umask)
- Maximale Dateigröße (siehe ulimit)
- den Lese/Schreibzeiger für jede Datei, die zur Zeit des fork Aufrufs geöffnet ist

Der Sohn unterscheidet sich in folgenden Prozeßkenndaten:

- Prozeßnummer
  der Sohn erhält eine neue eindeutige Prozeßnummer

- Prozeßnummer des Vaters
  der Prozeß, der fork aufruft, wird zum Vaterprozeß

- Alarmuhr
  die Restzeit in der Alarmuhr wird auf 0 gesetzt

- Prozeßzeiten
  Benutzer- und Systemzeiten des Sohnprozesses werden auf 0 gesetzt (siehe times)

Typ

Systemaufruf

Parameter

keine

Ergebnis

Prozeßnummer des Sohnes
bei Erfolg im Vaterprozeß

0 bei Erfolg im Sohnprozeß

–1 bei Fehler im Vaterprozeß, fork hat keinen neuen Prozeß erzeugt, da

- die Prozeßquote des Benutzers {BPROZ_MAX} aufgebraucht ist oder
- die Prozeßtabelle des Systems voll ist {SPROZ_MAX} oder
- der Swap-Bereich schon voll ist und keine neuen Prozesse zugelassen werden

Fehlermeldung

Im Fehlerfall kann fork einen der beiden Fehlercodes in errno ablegen:

EAGAIN : Keine weiteren Prozesse
ENOMEM : Arbeitsspeicher unzureichend

Hinweis

- Nach dem fork Aufruf läuft der Vaterprozeß weiter wie nach einem gewöhnlichen Funktionsaufruf. Der Sohnprozeß startet direkt nach dem fork Aufruf.

- Vater und Sohn haben einen gemeinsamen Lese/Schreibzeiger für jede Datei, die zur Zeit des fork Aufrufes geöffnet ist.
  Insbesondere können über diesen Mechanismus Standardein- und ausgabedateien an den Sohnprozeß weitergegeben werden.

Beispiel

Das folgende Beispiel zeigt eine typische Aufruffolge von fork und exec: zuerst wird fork aufgerufen, um einen neuen Prozeß zu erzeugen, dann ruft der neue Sohnprozeß mit exec ein Programm auf, das den aufrufenden Sohnprozeß überlagert:

```
char *kommando;
int pnr;

if((pnr = fork()) == 0)
               /* Sohn */
   execl("/bin/sh","sh","-c",kommando,NULL);
else if (pnr < 0)
     {         /* Fehler */
        perror("fork");
        exit(1);
     }
     else     /* Vater */
        .
        .
        .
```

> > > > exec, nice, ptrace, profil, signal, times, ulimit, umask, wait, popen

Formatierte Ausgabe in eine Datei

#include <stdio.h>

int fprintf(dz,format [,arg]...**)**
FILE *dz;
char *format;

fprintf gibt die Argumente gemäß den Formatanweisungen in *format* auf die Datei mit Dateizeiger *dz* aus. fprintf arbeitet wie printf, außer daß die Ausgabe in eine Datei und nicht auf Standardausgabe geschrieben wird.

Typ

C-Funktion (s)

Parameter

← FILE *dz — Dateizeiger für die Ausgabedatei

char *format

Die Formatzeichenreihe enthält drei Klassen von Zeichen:

a) Formatsteuerzeichen

- Neue Zeile ('\n')
- Rücksetzen ('\b)
- Tabulator ('\t')
- Seitenwechsel ('\f')
- Wagenrücklauf ('\r')

b) Zeichen, die 1:1 ohne Umwandlung ausgegeben werden

c) Formatangaben für die Ausgabe der Argumente von der Form:

```
                        [ [l]{d|o|x|u}     ]
  [+]      [n] [.m]     [                  ]
% [ ] [0]  [ ] [  ]     {    {D|O|X|U}     }
  [-]      [*] [.*]     [                  ]
                        [ {c|e|f|g|s|%}    ]
```

dabei bedeutet:

+	das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben
-	linksbündig ausrichten
0	mit Nullen auffüllen Standard: Leerzeichen werden links angefügt
n	minimale Gesamtfeldbreite (inklusive Dezimalpunkt). Falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung
*	die Gesamtfeldbreite ist variabel; der aktuelle Wert (Typ: unsigned) steht vor dem dazugehörenden Argument in der Argumentenliste
.m	*m* gibt die Stellen nach dem Komma für eine e- oder f-Konvertierung an oder die maximale Anzahl von Zeichen, die von einer Zeichenreihe ausgegeben werden sollen.
.*	Möglichkeit, die Stellen nach dem Komma oder die Länge einer Zeichenreihe variabel anzugeben; der aktuelle Wert (Typ:unsigned) steht vor dem dazugehörenden Argument in der Argumentenliste
l vor d, o, u, x	Format für ein Argument vom Typ long int.

Folgende Parameter legen die eigentliche Umwandlung fest:

c	einzelnes Zeichen (char); das Zeichen '\0' wird ignoriert
d	Dezimaldarstellung einer ganzen Zahl (int)
D	Dezimaldarstellung einer ganzen Zahl (long int)
e	Gleitpunktzahl (float oder double) im Format [-]d.ddd**e**{+ | -}dd Die Anzahl der Stellen nach dem Komma hängt von der Genauigkeitsangabe *.m* ab. Standard (keine Angabe): 5 Stellen
f	Gleitkommazahl (float oder double) im Format [-]ddd.ddd Die Anzahl der Stellen nach dem Komma hängt von der Genauigkeitsangabe in *.m* ab: Standard (keine Angabe) : 6 Stellen 0 : ganzahlige Ausgabe ohne Dezimalpunkt
g	Gleitkommazahl (float oder double) im d, f oder e Format, je nachdem, welche Darstellung unter Wahrung der Genauigkeit am wenigsten Platz beansprucht
o	Oktale Darstellung einer ganzen Zahl (int)
O	Oktale Darstellung einer ganzen Zahl (long int)
s	Format für Zeichenreihen Die Zeichenreihe sollte mit '\0' abgeschlossen sein. fprintf schreibt so viele Zeichen der Zeichenreihe, wie in *.m* angegeben ist: Standard (keine Angabe) : fprintf schreibt alle oder 0 : Zeichen bis '\0'
u	Vorzeichenlose Dezimalzahl (unsigned) Format zur Ausgabe von Zeigerwerten
x	hexadezimale Darstellung einer ganzen Zahl (int)
X	hexadezimale Darstellung einer ganzen Zahl (long int)
%	Ausdruck von %, keine Umwandlung

Ergebnis

0	bei Erfolg
negativ	bei Fehler

Beispiel

Folgendes Programmschema gibt in jedem Schleifendurchlauf die Werte der Variablen *var1*, *var2* und *var3* in eine Datei *protokoll* aus.

```
#include <stdio.h>

main()
{
   int i,var1;
   char var2[BUFSIZ];
   double var3;
   File *dz;
   .
   .
   dz = fopen("protokoll","w");
               /* Datei protokoll zum Schreiben öffnen */
   .
   .
   for (i=1;i<11;i++)
      {
        .
        .
        fprintf(dz,"Werte beim %2d-ten Durchlauf : \n \n",i);
        fprintf(dz,"1.Variable : %d \n",var1);
        fprintf(dz,"2.Variable : %s \n",var2);
        fprintf(dz,"3.Variable : %f \n",var3);
        .
        .
      }
}
```

Dateien

/usr/include/stdio.h

Definitionen für Standardein/ausgabe

> > > > printf, sprintf, putc, putchar, puts, scanf, fscanf

Zeichen ausgeben auf Datei

#include <stdio.h>

int fputc(c,dz)
char c;
FILE *dz;

fputc schreibt das Zeichen *c* in die Datei mit Dateizeiger *dz* an die aktuelle Lese/Schreibposition.

Typ

C-Funktion (s)

Parameter

char c	Zeichen, das auf die Datei geschrieben werden soll
← FILE *dz	Dateizeiger für die Ausgabedatei

Ergebnis

c	bei Erfolg liefert fputc das geschriebene Zeichen
EOF	sonst

Hinweis

Im Unterschied zu putc ist fputc eine Funktion und kein Makro.

Beispiel

siehe putc

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fopen, ferror, fread, fprintf, putc, putchar, putw, setbuf

Zeichenreihe auf Datei ausgeben

#include <stdio.h>

int fputs(s,dz**)**
char *s;
FILE *dz;

fputs schreibt die Zeichenreihe *s* in die Datei mit Dateizeiger *dz*.

Typ

C-Funktion (s)

Parameter

char *s	Zeichenreihe, die ausgegeben werden soll *s* muß mit '\0' abgeschlossen sein!
← FILE *dz	Dateizeiger für Ausgabedatei

Ergebnis

0	bei Erfolg
EOF	sonst

Hinweis

- Im Gegensatz zu puts schließt fputs seine Ausgabe nicht mit Neue Zeile ab.
- Das abschließende Nullbyte ('\0') von *s* wird nicht mitausgegeben.

Beispiel

Zeichenreihen von Standardeingabe einlesen und auf *datei* ausgeben:

```
#include <stdio.h>

main()
{
   FILE *fp;
   char s[BUFSIZ];

   if((fp=fopen("datei","w")) == NULL)
     {
       perror("fopen");
       exit(10);
     }

   while(gets(s) != NULL)
         fputs(s,fp);
 }
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > ferror, fopen, fread, fprintf, putc, puts, setbuf

Blockweise einlesen

#include <stdio.h>

int fread(zg,anz,n,dz)
char *zg;
int anz;
int n;
FILE *dz;

fread liest *n* Elemente, die jeweils *anz* Bytes beanspruchen, aus der Datei mit Dateizeiger *dz*. Die gelesenen Datenelemente speichert fread in den Bereich, auf dessen Anfang *zg* zeigt. Nach erfolgreichem Lesen steht der Lese/Schreibzeiger hinter dem letzten gelesenen Byte.

Typ

C-Funktion (s)

Parameter

← char *zg	Zeiger auf den Anfang des Bereichs, in den die eingelesenen Elemente abgespeichert werden.
int anz	Anzahl der Bytes, die jedes einzulesende Element beansprucht
int n	Anzahl der einzulesenden Elemente
FILE *dz	Dateizeiger für Eingabedatei, aus der fread liest

Ergebnis

Anzahl der eingelesenen Elemente
bei Erfolg

0 Dateiende oder Fehler

Achtung

Sie müssen dafür sorgen, daß der Bereich, auf den *zg* zeigt, zum Abspeichern der eingelesenen Datenelemente ausreicht.

Hinweis

- Um sicherzugehen, daß *anz* die richtige Größe für ein Datenelement angibt, sollten Sie die Funktion sizeof verwenden.
- fread unterscheidet nicht zwischen Dateiende und Fehler. Sie sollten daher vor (bzw. nach) jedem fread Aufruf mit den Funktionen feof und ferror überprüfen, ob ein korrekter Lesezugriff möglich ist.
- fread eignet sich dazu, Binärdaten aus einer Datei zu lesen.

Beispiel

```
                /* Einträge im aktuellen Dateiverzeichnis
                   interaktiv löschen */?

#include <stdio.h>

struct direntry {
       short inumber;
       char name[14];
                } dn;      /* Länge 16 */

FILE *dir;
int ch;

main()
{
                /* aktuelles Dateiverzeichnis zum Lesen öffnen */
   dir = fopen(".","r");
                /* zwei Einträge (=32 Bytes) überspringen:
                   aktuelles und Vater- Dateiverzeichnis */
   fseek(dir,32L,01);

   while(fread((char *)&dn,16,1,dir) != 0)
        if(dn.inumber)
                /* falls == 0, ==> Eintrag unbenutzt */
                {
                 printf("%s j? ",dn.name);
                 if((ch=getchar()) == 'j')
                    unlink(dn.name);
                /* restliche Zeichen überlesen */
                 while(ch != '\n' && ch != EOF)
                       ch = getchar();
                }
}
```

Dateien

/usr/include/stdio.h

Definitionen für Standardein/ausgabe

> > > > fwrite, feof, ferror, read, fopen, fgetc, fgets, fscanf

Speicherplatz freigeben

void free(zg)
char *zg;

free gibt den Speicherplatz frei, der zuvor mittels malloc, calloc oder realloc reserviert wurde.

Typ

C-Funktion

Parameter

char *zg	Zeiger auf den freizugebenden Speicherbereich *zg* muß das Ergebnis eines vorangegangenen malloc, calloc oder realloc Aufrufs sein!

Beispiel

siehe Beispiel bei malloc

> > > > malloc, calloc, realloc

Dateizeiger neu zuweisen

#include <stdio.h>

FILE *freopen(d _ name,art,dz**)**
char *d _ name, *art;
FILE *dz;

freopen dient dazu, einen bereits definierten Dateizeiger einer neuen Datei zuzuordnen.
freopen schließt die Datei mit Dateizeiger *dz*, öffnet die Datei *d _ name* und ordnet ihr die FILE-Struktur mit Dateizeiger *dz* zu.

Typ

C-Funktion (s)

Parameter

char *d _ name
Dateiname der neuen Datei

char *art — art gibt die gewünschte Zugriffsart an und kann eine der folgenden Zeichenreihen sein:

"r"	Datei öffnen zum Lesen
"w"	Datei öffnen zum Schreiben
"a"	Datei öffnen, um Text ans Ende der Datei anzuhängen.

FILE *dz — Dateizeiger, der neu zugewiesen werden soll

Ergebnis

dz — bei Erfolg liefert freopen den ursprünglichen Dateizeiger

Nullzeiger — freopen konnte die neue Datei nicht öffnen, da

- das Programm auf die Datei *d _ name* nicht zugreifen kann oder
- eine Datei zum Lesen geöffnet werden soll, die nicht existiert.

Hinweis

- Die Datei, der *dz* ursprünglich zugeordnet war, wird auf alle Fälle geschlossen, auch wenn die neue Datei nicht geöffnet werden kann.
- Wenn Sie eine Datei zum Schreiben öffnen, wird der alte Inhalt gelöscht; wenn Sie die Datei dagegen mit Modus "a" öffnen, bleibt der alte Inhalt erhalten und der neue Text wird ans Ende der Datei angehängt.
- Der Versuch, eine Datei, die es noch nicht gibt, zum Lesen zu öffnen, endet mit Fehlerabbruch. Wenn Sie eine Datei, die es noch nicht gibt, zum Schreiben öffnen, wird die Datei neu erstellt.

Beispiel

freopen wird hauptsächlich dazu verwendet, die vordefinierten Dateizeiger stdin, stdout und stderr umzudefinieren, d.h. anderen Dateien als i.a. der angeschlossenen Datensichtstation zuzuordnen. Diese Anwendung entspricht dem Umlenkmechanismus der Shell.

Folgendes Programmstück macht die Datei *neu* zur Standardausgabedatei:

```
FILE *dz;

dz = freopen("neu","w",stdout)
```

dz und stdout sind danach beide Dateizeiger für die Datei *neu.*

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

>>>> fopen, fdopen

normierte Darstellung zur Basis 2 einer Gleitkommazahl

double frexp(wert,e_zg**)**
double wert;
int *e_zg;

frexp liefert zu einem Gleitkommawert seine normierte Darstellung zur Basis 2, die von der Form ist:

wert = x * 2**n, 0.5 <= $|x|$ < 1.0, n ganzzahlig

Dabei ist die Mantisse x das Ergebnis von frexp und der Exponent n kann indirekt über *e_zg* angesprochen werden.

Typ

C-Funktion

Parameter

double wert	Gleitkommawert
← int *e_zg	Zeiger auf den Exponent der Basis-2-Darstellung

Ergebnis

mantisse x	*wert* = x * 2**n x ist ein Wert kleiner 1

Achtung

Sie müssen auch hier den Speicherplatz für den Exponenten explizit bereitstellen!

Beispiel

Normierte Darstellung zur Basis 2 der Zahl 5:

```
#include <stdio.h>

double frexp();
int exp;

main()
{
   double z;

   z = frexp((double)5,&exp);
   printf("5 = %g * 2 ** %d\n",z,exp);
}
```

> > > > ldexp, modf

Formatiertes Einlesen von einer Datei

#include <stdio.h>

int fscanf(dz,format [,arg_zg]...)
FILE *dz;
char *format;
<typ> *arg_zg, ...;

fscanf liest von der Datei mit Dateizeiger *dz*. fscanf wandelt ein Eingabefeld gemäß den Formatanweisungen um und speichert das Ergebnis an die Stelle, die der entsprechende Ergebniszeiger angibt.

Typ

C-Funktion (s)

Parameter

FILE *dz   Dateizeiger für die Eingabedatei

char *format

Die Formatzeichenreihe kann drei Klassen von Zeichen enthalten:

a) Zwischenraum

- Tabulator
- Leerzeichen
- Zeilenwechsel

Zwischenraumzeichen in der Eingabe werden außer bei %c ignoriert.

b) beliebiges Zeichen außer '%'

das Zeichen muß mit dem nächsten Zeichen aus der Eingabe übereinstimmen, ansonsten wird die Eingabebearbeitung abgebrochen.

c) Formatangaben für die Umwandlung der eingelesenen Zeichen. Eine Formatangabe ist von der Form:

```
              ⎡ [l] {d|e|f|o|x}    ⎤
              ⎢     {D|E|F|O|X}    ⎥
% [*] [n]    ⎨     {c|s|%}        ⎬
              ⎣     {[...]|[^...]} ⎦
```

Zu einer Formatangabe gehört ein Eingabefeld d.h. eine Zeichenreihe ohne Leerzeichen. Führende Tabulator- und Leerzeichen werden übersprungen (beachte jedoch Format %c!). Die Länge der Zeichenreihe ist entweder explizit durch die Feldbreite *n* festgelegt oder ergibt sich implizit, sobald das erste Zeichen gelesen wird, das nicht zu der Formatangabe paßt.
Das erste '%' kennzeichnet die Formatangabe, die restlichen Zeichen werden wie folgt interpretiert:

* — überspringe eine Zuweisung:
das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber nicht abgespeichert.

n — maximale Länge der umzuwandelnden Eingabezeichenreihe; tritt vorher ein Zeichen auf, das nicht zur Formatangabe paßt, wird die Länge entsprechend gekürzt.

l — Angabe für doppelte Länge:

vor d, o, x

Umwandlung einer Dezimal-, Oktal- oder Hexadezimalzahl in long int

← long *arg_zg

vor e, f

Gleitkommazahl
Umwandlung in double

← double *arg_zg

Folgende Formatelemente legen die eigentliche Umwandlung fest:

Formatelement/ Ergebniszeiger	erwartete Eingabe
c ← char *arg_zg	einzelnes (abdruckbares) Zeichen; Achtung ein einzelnes Zwischenraumzeichen wird auch gelesen und nicht wie sonst übersprungen!
nc ← char arg_zg[]	Feld von *n* (abdruckbaren) Zeichen Achtung führende Tabulator- und Leerzeichen werden übersprungen, aber Zwischenraumzeichen innerhalb des Eingabefeldes nicht (sind keine Trenner mehr)!
d ← int *arg_zg	ganze Dezimalzahl (mit oder ohne führende Null)
D ← long *arg_zg	ganze Dezimalzahl (mit oder ohne führende Null) Umwandlung wie bei ld
e ← float *arg_zg	Gleitkommazahl im Format: [+|-]dd[.dd{e|E}[+|-]dd]
E ← double *arg_zg	Gleitkommazahl wie bei e Umwandlung wie bei le
f ← float *arg_zg	Gleitkommazahl im Format: dd[.ddd]
F ← double *arg_zg	Gleitkommazahl wie bei f Umwandlung wie bei lf

Formatelement/ Ergebniszeiger	erwartete Eingabe
o	ganze Oktalzahl (mit oder ohne führende Null)
← int *arg_zg	
O	ganze Oktalzahl (mit oder ohne führende Null) Umwandlung wie bei lo
← long *arg_zg	
s	Zeichenreihe; Das zugehörige Eingabefeld wird durch ein Zwischenraumzeichen abgeschlossen, falls eine Feldbreite *n* angegeben ist, werden *n* Zeichen oder bis zum nächsten Zwischenraum gelesen, je nachdem was vorher kommt (Zwischenraumzeichen sind hier also Trenner, vgl %nc!) Die eingelesene Zeichenreihe wird mit '\0' abgeschlossen. Führende Tabulator- und Leerzeichen werden übersprungen, also
1s	liest das erste Nicht-Leerzeichen (vgl %c!).
← char arg_zg[]	
x	ganze Hexadezimalzahl (mit oder ohne führende Null) (aber nicht 0xddd!)
← int *arg_zg	
X	ganze Hexadezimalzahl (mit oder ohne führende Null) Umwandlung wie bei lx
← long *arg_zg	
[...]	Zeichenreihe, die nur aus Zeichen besteht, die in [...] vorkommen, z.B: [a] : Zeichenreihe, die nur aus 'a' besteht. Es wird eingelesen bis zum ersten Zeichen, das nicht in [...] vorkommt.
← char arg_zg[]	
[^...]	Zeichenreihe, die nur aus Zeichen besteht, die nicht in [^...] vorkommen, z.B: [^a]: Zeichenreihe, in der kein 'a' vorkommt. Es wird eingelesen bis das erste Zeichen auftritt, das in [^...] vorkommt.
← char arg_zg[]	
%	das Zeichen '%' selbst.

Ergebnis

Anzahl der eingelesenen und erfolgreich umgewandelten Datenelemente

	Mit dem Ergebnis eines erfolgreichen fscanf Aufrufes können Sie feststellen, wieviele Datenelemente tatsächlich eingelesen wurden.
EOF	Dateiende
0	Konvertierungsfehler

Hinweis

- Das Ergebnis eines fscanf Aufrufes sollten Sie immer abfragen, um sicher zu sein, daß kein Fehler passiert ist!
- Der nächste fscanf Aufruf startet mit dem Lesen unmittelbar nach dem Zeichen, das als letztes vom vorherigen Aufruf verarbeitet wurde.
- Wenn ein Eingabezeichen nicht dem angegebenen Format entspricht, wird es in den Eingabepuffer zurückgeschrieben. Es muß dort mit getc abgeholt werden, sonst erhält der nächste fscanf Aufruf das gleiche Zeichen wieder.

Beispiel

Der Aufruf:

```
int i;
float x;
char name[20];
scanf("%2d %f %*d %6s",&i,&x,name);

weist die Eingabedaten

234567 678 Zipfelxy

wie folgt zu:
i    : 23
x    : 4567.0
name : Zipfel\0
```

Die Zuweisung von 678 wird wegen der %*d Angabe nicht ausgeführt. Der nächste Aufruf einer Einlesefunktion beginnt mit dem Zeichen 'x'.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fgetc, fgets, fread, fopen, setbuf,scanf, sscanf

Lese/Schreibzeiger positionieren

#include <stdio.h>

int fseek(dz,distanz,ort**)**
FILE *dz;
long distanz;
int ort;

fseek verschiebt den Lese/Schreibzeiger für die Datei mit Dateizeiger *dz* gemäß den Angaben in *distanz* und *ort*.
Sie haben also damit die Möglichkeit, eine Datei nicht-sequentiell zu bearbeiten.

Typ

C-Funktion (s)

Parameter

FILE *dz — Dateizeiger für die Datei, bei der der Lese/Schreibzeiger positioniert werden soll.

long distanz
Anzahl der Bytes, um die der aktuelle Lese/Schreibzeiger verschoben werden soll:

positive Zahl
Verschiebung Richtung Dateiende
negative Zahl
Verschiebung Richtung Dateianfang

int ort — gibt an, wo die Verschiebung stattfinden soll:

0 Dateianfang
1 aktuelle Position
2 Dateiende

Ergebnis

0	bei Erfolg
-1	Fehler

Hinweis

- Der Aufruf fseek(dz,0L,0) ist äquivalent zu dem Aufruf rewind(dz).
- Die Wirkung von ungetc wird durch einen fseek Aufruf zerstört, d.h. ein zurückgestelltes Zeichen ist nicht mehr verfügbar.
- Auf einer Pipe kann nicht positioniert werden!

Beispiel

datei ab dem elften Zeichen bis Dateiende lesen:

```
#include <stdio.h>

main()
{
  FILE *fp;
  int c;

  if((fp = fopen("datei","r"))!= NULL);
{
               /* die ersten 10 Zeichen überspringen */
  fseek(fp,10L,0);
  while((c=getc(fp)) != EOF)
        putc(c,stdout);
  fclose(fp);
}
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > ftell, lseek, ungetc

Dateiinformationen ausgeben

#include <sys/types.h>
#include <sys/stat.h>

int fstat(dk,dinf**)**
int dk;
struct stat *dinf;

fstat schreibt ausführliche Information über die geöffnete Datei mit Dateikennzahl *dk* in die Struktur, auf die *dinf* zeigt.

Typ

Systemaufruf

Parameter

int dk — Dateikennzahl, die durch einen open, creat, dup, fcntl, oder pipe Aufruf zugewiesen wurde.

← struct stat *dinf

Zeiger auf die Ergebnisstruktur. Die Struktur ist in <sys/stat.h> wie folgt definiert:

```
struct stat
{
        dev_t   st_dev;          /* major- und minor-Nummer */
                                 /* des Geräts, das einen   */
                                 /* Dateiverzeichniseintrag für */
                                 /* diese Datei enthält */
        ino_t   st_ino;          /* Indexnummer (i-node)*/
        unsigned short st_mode;  /* Dateityp, Zugriffsrechte */
        short   st_nlink;        /* Anzahl der Verweise */
        short   st_uid;          /* Benutzernr. des Eigentümers */
        short   st_gid;          /* Gruppennummer */
        dev_t   st_rdev;         /* major- und minor-Nummer */
                                 /* nur für zeichen- oder block-*/
                                 /* orientierte Gerätedateien */
        off_t   st_size;         /* Dateigröße in Bytes */
        time_t  st_atime;        /* letzter Zugriff  */
        time_t  st_mtime;        /* letzte Änderung  */
        time_t  st_ctime;        /* letzte Änderung des Datei- */
                                 /* status  */
};
```

<sys/stat.h> enthält außerdem Defintionen von symbolischen Konstanten, mit denen die Bitbelegungen der Komponente st_mode gedeutet werden, u.a.:

```
#define S_IFMT   0170000      /* Dateityp Maske */
#define S_IFDIR  0040000      /* Dateiverzeichnis */
#define S_IFCHR  0020000      /* Zeichen-orientiert */
#define S_IFBLK  0060000      /* Block-orientiert */
#define S_IFREG  0100000      /* normale Datei */

#define S_ISUID  0004000      /* s-Bit für Eigentümer */
#define S_ISGID  0002000      /* s-Bit für Gruppe */
#define S_ISVTX  0001000      /* sticky Bit */
#define S_IREAD  0000400      /* Leseerlaubnis, Eigentümer */
#define S_IWRITE 0000200      /* Schreiberlaubnis, Eigentümer */
#define S_IEXEC  0000100      /* Ausführerlaubnis, Eigentümer */
```

Die Bitbelegungen 0000010 bis 0000070 geben die Zugriffsrechte für Gruppe an, die Bitbelegungen 0000001 bis 0000007 die Zugriffsrechte für Andere.
Dabei bedeutet wie bei chmod:

1 : ausführen
2 : schreiben
4 : lesen

Ergebnis

0 fstat hat die Dateiinformationen in *dinf* abgelegt.

-1 Fehler, wenn

- *dk* keine gültige Dateikennzahl ist oder
- *dinf* eine unzulässige Adresse ist

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EBDAF : Unzulässige Dateinummer
EFAULT : Unzulässige Adresse

Achtung

Sie müssen den Speicherplatz für die Ergebnisstruktur stat explizit bereitstellen!

Hinweis

- die Datentypen in stat sind verschiedene integer Typen, die in <sys/types.h> definiert sind. Sie haben folgende Bedeutung:

ino_t	Datentyp für eine Indexnummer
time_t	Datentyp für eine Zeitangabe in Sekunden
dev_t	Datentyp für eine major- und minor-Nummer
off_t	Datentyp für die Position des Lese/Schreibzeigers

- Die drei Zeitangaben werden bei folgenden Systemaufrufen verändert:

st_atime :	creat, locking, mknod, pipe, utime, read
st_mtime :	creat, mknod, pipe, utime, write
st_ctime :	chmod, chown, creat, link, mknod, pipe, unlink, utime, write

- Die Komponente st_atime, die den letzten Zugriff angibt, wird nicht verändert, wenn ein Dateiverzeichnis durchsucht wird!
- Die Komponente st_mtime, die die letzte Änderung angibt, ist von Änderungen des Indexeintrages nicht betroffen!
- Wenn die Dateikennzahl eine Pipe bezeichnet, liefert fstat als Dateityp normale Datei, die Zugriffsrechte sind eingeschränkt und die Größe gibt an, wieviele Bytes in der Pipe aktuell gespeichert sind.

Beispiel

siehe Beispiel bei pipe

Dateien

/usr/include/sys/filsys.h
Definitionen für die Implementierung des Dateisystems

/usr/include/types.h
Typdefinitionen

/usr/include/stat.h
Definition der Struktur stat

> > > > stat, ls(Kommando)

Aktuelle Lese/Schreibposition abfragen

#include <stdio.h>

long ftell(dz)
FILE *dz;

ftell bestimmt die aktuelle Position des Lese/Schreibzeigers für die Datei mit Dateizeiger *dz.*

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger für die Datei, deren Lese/Schreibposition bestimmt werden soll

Ergebnis

Anzahl der Bytes, die der Lese/Schreibzeiger vom Dateianfang entfernt ist

Beispiel

Ab dem elften Zeichen wird jedes Zeichen von *datei* mit der Position des Lese/Schreibzeigers ausgegeben:

```
#include <stdio.h>

main()
{
   FILE *fp;
   int c;
   if((fp = fopen("datei","r")) != NULL)
     {
              /* die ersten 10 Zeichen werden übersprungen */
       fseek(fp,10L,0);
       while((c=getc(fp)) != EOF)
            printf("%Position : %ld, Zeichen : %c\n",ftell(fp),c);
       fclose(fp);
     }
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fseek, tell

Aktuelle Zeit

#include <sys/types.h>
#include <sys/timeb.h>

int ftime(zg**)**
struct timeb *zg;

ftime liefert dieselbe Zeit wie time, bloß in einem anderen Format. ftime füllt die Struktur auf, auf die *zg* zeigt.

Typ

Systemaufruf

Parameter

← struct timeb *zg

Zeiger auf eine Struktur, die wie folgt in <sys/timeb.h> definiert ist:

```
struct timeb {
        time_t  time;
        unsigned short millitm;
        short   timezone;
        short   dstflag;
 };
```

Ergebnis

0	immer

Achtung

Wie immer in solchen Fällen, müssen Sie den Speicherplatz für die Ergebnisstruktur explizit bereitstellen!

Hinweis

- der Typ time_t ist in <sys/types.h> definiert.
- Die einzelnen Komponenten einer von ftime aufgefüllten Struktur enthalten:

time :	Zeit in Sekunden seit dem 1.Januar 1970 00:00:00 (GMT)
mtime :	Angabe in Millisekunden (0 bis 1000) zur Erhöhung der Genauignkeit von time
timezone:	lokale Zeitzone, gemessen in Minuten westlich von Greenwich
dstflag :	Flag für Sommerzeit (wird nicht unterstützt, immer 0)

Beispiel

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>

struct timeb tp;

main()
{

   ftime(&tp);
   printf("%d\n",tp.time);
   printf("%d\n",tp.millitm);
   printf("%d\n",tp.timezone);
   printf("%d\n",tp.dstflag);
 }
```

Dateien

/usr/include/sys/types.h
Typdefinitionen

/usr/include/sys/timeb.h
Definition der Struktur timeb

> > > > time, stime, ctime, date(Kommano)

Blockweise ausgeben

#include <stdio.h>

int fwrite(zg,anz,n,dz)
char *zg;
unsigned anz;
int n;
FILE *dz;

fwrite schreibt *n* Elemente, die jeweils *anz* Bytes beanspruchen auf die Datei mit Dateizeiger *dz.*
Die Position des Lese/Schreibzeigers ist anschließend um die Anzahl der geschriebenen Bytes erhöht.

Typ

C-Funktion (s)

Parameter

char *zg	Zeiger auf das erste Element, das übertragen werden soll
int anz	Größe in Bytes von einem Element
int n	Anzahl der Elemente, die geschrieben werden sollen
← FILE *dz	Dateizeiger für die Ausgabedatei

Ergebnis

Anzahl der tatsächlich geschriebenen Elemente
bei Erfolg

0	Dateiende oder Fehler

Hinweis

- Um sicherzugehen, daß *anz* die richtige Größe für ein Datenelement angibt, sollten Sie die Funktion sizeof verwenden.
- fwrite zeigt Dateiende oder Fehler nicht explizit an. Sie sollten daher vor (bzw. nach) jedem fwrite Aufruf mit den Funktionen feof und ferror überprüfen, ob ein korrekter Schreibzugriff möglich ist.
- fwrite eignet sich dazu, Binärdaten auf eine Datei zu schreiben

Beispiel

Folgendes Programmstück überträgt zwei Personeneinträge auf die Datei mit Dateizeiger *p_liste*.

```
FILE *p_liste;
struct eintrag{
         char name[20];
         int alter;
} person[10];

fwrite(person,sizeof(struct eintrag),2,p_liste);
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fread, feof, ferror

Datum mit Uhrzeit (MEZ) in Deutsch

char *gctime(sek _ zg**)**
long *sek _ zg;

gctime ist das deutsche Gegenstück zu ctime.
gctime interpretiert den Wert, auf den *sek _ zg* zeigt, als Zeit in Sekunden seit dem 1.Januar 1970 00:00:00 (GMT). Es berechnet daraus MEZ und wandelt das Ergebnis in eine ASCII-Zeichenreihe um. Die Ergebniszeichenreihe hat die Länge 26 und das Format eines deutschen Datums mit Uhrzeit-Angabe:

```
                        Wochentag.Tag.Monat.Jahr, Std:Min:Sek
zum Beispiel:                 Sa. 22.  Jun.1985,   12: 34: 24 \n\0
```

Typ

C-Funktion

Parameter

long *sek _ zg
: Zeiger auf eine Zeitangabe in Sekunden

Ergebnis

Zeiger auf die erzeugte ASCII-Zeichenreihe
: Die Ergebniszeichenreihe hat die Länge 26 und ist mit Neue Zeile, Nullbyte (\n\0) abgeschlossen.

Achtung

- gctime liefert sein Ergebnis in einem statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Außerdem verwenden gctime und ctime **denselben** Datenbereich, d.h. wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben!

Hinweis

- Die Aufrufe gctime(sek_zg) und meztime(localtime(sek_zg)) sind äquivalent.
- Der Aufruf clock = time(0L); gctime(&clock); liefert das aktuelle Datum mit Uhrzeit in Deutsch.
- Für die Ausgabe gilt die 24-Stunden-Uhr.

Beispiel

Aktuelles Datum mit Uhrzeit in Englisch und Deutsch ausgeben:

```
long time();
char *ctime();
char *gctime();
long clock;

main()
{
   clock = time(0L);
   printf("%s\n",ctime(&clock));
   printf("%s\n",gctime(&clock));
}
```

>>>> ctime, meztime, localtime, asctime, gmtime, time,
datum(Kommando)

Umwandlung in ASCII für die Ausgabe

char *gcvt(wert,anz,puf**)**
double wert;
int anz;
char *puf;

gcvt wandelt einen internen Gleitkommawert in eine Zeichenreihe aus ASCII-Ziffern um, speichert die Zeichenreihe in den Bereich, auf den *puf* zeigt und liefert als Ergebnis einen Zeiger auf diesen Bereich. Wenn möglich, werden *anz* signifikante Stellen für Fortran F-Format erstellt, wenn nicht, ist die Ausgabe in Fortran E-Format, wobei Nullen am Ende eventuell unterdrückt werden.

Typ

C-Funktion

Parameter

double wert
Gleitkommawert, der für die Ausgabe aufbereitet werden soll.

int anz — Anzahl der Ziffern in der Ergebniszeichenreihe

← char *puf — Zeiger auf die umgewandelte Zeichenreihe

Ergebnis

Zeiger auf die umgewandelte ASCII-Zeichenreihe
bei Erfolg

Fehlermeldung

Falsche Parameter, etwa ein integer statt double Wert, führen zum Programmabbruch mit der Meldung:
'Speicherfehler - Speicherabzug(core) auf Platte geschrieben.'

Achtung

Der Zeiger *puf* muß auf einen Speicherbereich von mindestens (*anz* + 4) Bytes zeigen.

Hinweis

- Bei der Umwandlung wird die niedrigste Stelle gerundet.
- Die Standardausgabefunktion printf benützt ecvt, fcvt und gcvt.

Beispiel

siehe Beispiel bei ecvt

> > > > fcvt, gcvt, printf, fprintf, sprintf

Zeichen von Datei einlesen

#include <stdio.h>

int getc(dz**)**
FILE *dz;

getc liest ein Zeichen aus der Datei mit Dateizeiger *dz*.

Typ

Makro (s)

Parameter

FILE *dz	Dateizeiger für die Eingabedatei

Ergebnis

eingelesenes Zeichen als int
getc liefert bei Erfolg das eingelesene Zeichen als positiven integer Wert

EOF	Fehler oder Dateiende

Hinweis

- Im Unterschied zu fgetc ist getc ein Makro und keine Funktion.
- Wenn Sie in Ihrem Programm einen Vergleich wie etwa in
  while((c = getc(dz)) != EOF)
  verwenden, sollten Sie die Variable *c* immer als integer Größe vereinbaren. Wenn Sie *c* als char definieren, kann es sein, daß die Bedingung nie erfüllt ist, weil die Vorzeichenpropagierung bei der Umwandlung von char in int maschinenabhängig ist.

Beispiel

Programmstück für zeichenweises Einlesen aus der Datei mit Dateizeiger *eingabe* bis Dateiende erreicht ist:

```
int c, i=0;
char buf[MAX];
FILE *eingabe;
  .
  .
  .

while((c = getc(eingabe)) != EOF)
      buf[i++] = c;
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fgetc, getchar, getw, gets, fscanf, fread, ungetc

Ein Zeichen von Standardeingabe lesen

#include <stdio.h>

int getchar()

getchar liest ein Zeichen von Standardeingabe (Dateizeiger stdin).
getchar ist ein Makro und definiert als:

getc(stdin)

Typ

Makro (s)

Parameter

keine

Ergebnis

eingelesenes Zeichen als int
getchar liefert bei Erfolg das eingelesene Zeichen als positiven integer Wert.

EOF Fehler oder Dateiende

Hinweis

Möchten Sie in Ihrem Programm mit einer Schleife der Art
int c;
while((c = getchar()) != EOF)
bis Eingabeende lesen, so sollten Sie *c* als int und nicht als char Variable vereinbaren, da Sie sonst durch die Typangleichung in der Zuweisung eine Endlosschleife bekommen können (maschinenabhängig).

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > getc, gets, scanf, putc, putc, printf

Effektive Gruppennummer abfragen

int getegid()

getegid liefert die **effektive** Gruppennummer des Prozesses. Mit dieser Gruppennummer werden die Zugriffsrechte (bzgl. Gruppe) des Prozesses überprüft.

Typ

Systemaufruf

Parameter

keine

Ergebnis

effektive Gruppennummer

Hinweis

getegid ist nützlich bei Programmen, die das s-Bit für Gruppe gesetzt haben, um die Gruppennummer desjenigen festzustellen, der die Programmdatei erstellt hat.

Beispiel

Schreiben Sie die Objektdatei a.out zu diesem Programm unter Ihrer Kennung (nicht root) und setzen Sie mit dem chmod Kommando
(z.B. chmod 2777 a.out) das s-Bit für Gruppe.
Rufen Sie dann das lauffähige Programm a.out unter einer Kennung
(z.B. root) mit anderer Gruppennummer auf:

```
main()
{
  printf("reale Gruppennr:%d\neffektive Gruppennr: %d\n",getgid(),getegid());
}
```

> > > > geteuid, getgid, getuid, setuid

Umgebungsvariable abfragen

char *getenv(name**)**
char *name;

getenv sucht unter den Umgebungsvariablen (extern char **environ) nach einer Variablendefinition:

name = wert.

und liefert bei Erfolg einen Zeiger auf *wert* zurück.

Typ

C-Funktion

Parameter

char *name	Zeiger auf den Variablenname, der gesucht werden soll.

Ergebnis

Zeiger auf das erste Zeichen von wert	falls die Programmumgebung eine Definition *name = wert* enthält
Nullzeiger	sonst

Beispiel

Wenn Sie folgendes Programm z.B. mit a.out USER aufrufen, liefert es Ihnen den aktuellen Benutzernamen.

```
#include <stdio.h>

main(argc,argv)
int argc;
char **argv;
{
  if((getenv(argv[1])) != NULL)
      printf("%s = %s\n",argv[1],getenv(argv[1]));
  else
      printf("%s : nicht vorhanden\n",argv[1]);
}
```

Externe Größen

char **environ
: Feld, das die Umgebungsvariablen enthält

> > > > exec, printenv(Kommando)

Effektive Benutzernummer abfragen

int geteuid()

geteuid liefert die **effektive** Benutzernummer des Prozesses. Mit dieser Nummer werden die Zugriffsrechte (bzgl. Eigentümer) des Prozesses überprüft.

Typ

Systemaufruf

Parameter

keine

Ergebnis

effektive Benutzernummer

Hinweis

geteuid ist nützlich bei Programmen, die das s-Bit gesetzt haben, um die Benutzernummer desjenigen festzustellen, der die Programmdatei erstellt hat.

Beispiel

Schreiben Sie die Objektdatei a.out zu diesem Programm unter Ihrer Kennung (nicht root) und setzen Sie mit dem chmod Kommando
(z.B. chmod 4777 a.out) das s-Bit für Eigentümer.
Rufen Sie dann das lauffähige Programm a.out unter einer anderen Kennung (z.B. root) auf:

```
main()
{
  printf("reale Benutzernr:%d\neffektive Benutzernr:%d\n",getuid(),geteuid());
}
```

>>>> getegid, getgid, getuid, setuid

Reale Gruppennummer abfragen

int getgid()

getgid liefert die **reale** Gruppennummer des Prozesses. Diese Nummer ist die Gruppennummer des Aufrufers der Programmdatei.

Typ

Systemaufruf

Parameter

keine

Ergebnis

reale Gruppennummer

Hinweis

getgid ist nützlich bei Programmen, die das s-Bit für Gruppe gesetzt haben, um die Gruppennummer des Programmaufrufers zu ermitteln.

Beispiel

Schreiben Sie die Objektdatei a.out zu diesem Programm unter Ihrer Kennung (nicht root) und setzen Sie mit dem chmod Kommando (z.B. chmod 2777 a.out) das s-Bit für Gruppe.
Rufen Sie dann das lauffähige Programm a.out unter einer Kennung (z.B. root) mit anderer Gruppennummer auf:

```
main()
{
  printf("reale Gruppennr: %d\neffektive Gruppennr: %d\n",getgid(),getegid());
}
```

> > > > getegid, geteuid, getuid

Nächste Zeile von /etc/group ausgeben

#include <grp.h>

struct group *getgrent()

getgrent liefert die nächste Zeile aus der "Gruppendatei" /etc/group.

Typ

C-Funktion

Parameter

keine

Ergebnis

Zeiger auf eine group Struktur

getgrent liefert die nächste Zeile von /etc/group in einer Struktur, die wie folgt in <grp.h> definiert ist:

```
struct group {
        char *gr_name;      /* Gruppenname */
        char *gr_passwd;    /* Passwort für Gruppe*/
        int gr_gid;         /* Gruppennummer */
        char **gr_mem;      /* Vektor von Zeigern */
                            /* auf die einzelnen  */
                            /* Gruppenmitglieder  */
}
```

Nullzeiger   Dateiende oder Fehler

Achtung

getgrent schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

getgrent öffnet die "Gruppendatei" automatisch.

Beispiel

Folgendes Beispiel druckt die Einträge von /etc/group mit entsprechenden Kommentaren:

```
#include <stdio.h>
#include <grp.h>

struct group *getgrent();
struct group *k;

main()
{
  int i;
  while((k = getgrent()) != NULL)
        {
          printf(" Name : %s\n",k->gr_name);
          printf(" Passwort : %s\n",k->gr_passwd);
          printf(" Gruppennummer : %d\n",k->gr_gid);
          i=0;
          while(k->gr_mem[i] != NULL)
                printf(" Mitglied : %s\n",k->gr_mem[i++]);
        }
  endgrent();
}
```

Dateien

/etc/group — Gruppeneinträge; Eingabedatei für getgrent

/usr/include/grp.h — Definition der Struktur group

> > > > getgrgid, getgrnam, setgrent, endgrent

Gruppennummer in /etc/group suchen

#include <grp.h>

struct group *getgrgid(nr)
int nr;

getgrgid sucht in der "Gruppendatei" /etc/group die Gruppennummer *nr* und liefert bei Erfolg die zugehörige Zeile zurück.

Typ

C-Funktion

Parameter

int nr Gruppennummer, die in /etc/group gesucht werden soll.

Ergebnis

Zeiger auf group Struktur
Falls getgrgid die Gruppennummer *nr* in /etc/group findet, liefert es den Inhalt der zugehörigen Zeile in einer Struktur, die wie folgt in <grp.h> definiert ist:

```
struct group {
        char *gr_name;      /* Gruppenname */
        char *gr_passwd;    /* Passwort für Gruppe*/
        int gr_gid;         /* Gruppennummer */
        char **gr_mem;      /* Vektor von Zeigern */
                            /* auf die einzelnen  */
                            /* Gruppenmitglieder  */
}
```

Nullzeiger Dateiende oder Fehler

Achtung

getgrgid schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird.

Hinweis

getgrgid öffnet die "Gruppendatei" automatisch.

Beispiel

Folgendes Beispiel druckt die Einträge in /etc/group zur Gruppennummer 1 mit entsprechenden Kommentaren:

```
#include <stdio.h>
#include <grp.h>

struct group *getgrgid();
struct group *k;

main()
{
  int i;
  if((k = getgrgid(1)) != NULL)
       {
         printf(" Name : %s\n",k->gr_name);
         printf(" Passwort : %s\n",k->gr_passwd);
         printf(" Gruppennummer : %d\n",k->gr_gid);
         i=0;
         while(k->gr_mem[i] != NULL)
               printf(" Mitglied : %s\n",k->gr_mem[i++]);
       }
}
```

Dateien

/etc/group Gruppeneinträge; Eingabedatei für getgrgid

/usr/include/grp.h
Definition der Struktur group

> > > > getgrent, getgrnam, setgrent, endgrent

Gruppennamen in /etc/group suchen

#include <grp.h>

struct group *getgrnam(name**)**
char *name;

getgrnam sucht in der "Gruppendatei" /etc/group den Gruppennamen *name* und liefert bei Erfolg die zugehörige Zeile zurück.

Typ

C-Funktion

Parameter

char *name   Gruppenname, der in /etc/group gesucht werden soll.

Ergebnis

Zeiger auf group Struktur
Falls getgrnam den Gruppennamen *name* in /etc/group findet, liefert es den Inhalt der zugehörigen Zeile in einer Struktur, die wie folgt in <grp.h> definiert ist:

```
struct group {
        char *gr_name;      /* Gruppenname */
        char *gr_passwd;    /* Passwort für Gruppe*/
        int gr_gid;         /* Gruppennummer */
        char **gr_mem;      /* Vektor von Zeigern */
                            /* auf die einzelnen  */
                            /* Gruppenmitglieder  */
}
```

Nullzeiger   Dateiende oder Fehler

Achtung

getgrnam schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird.

Hinweis

getgrnam öffnet die "Gruppendatei" automatisch.

Beispiel

Folgendes Beispiel druckt die Einträge in /etc/group zu einem beim Aufruf übergebenen Namen mit entsprechenden Kommentaren:

```
#include <stdio.h>
#include <grp.h>

struct group *getgrnam();
struct group *k;

main(argc,argv)
int argc;
char **argv;
{
  int i;
  if((k = getgrnam(argv[1])) != NULL)
       {
         printf(" Name : %s\n",k->gr_name);
         printf(" Passwort : %s\n",k->gr_passwd);
         printf(" Gruppennummer : %d\n",k->gr_gid);
         i=0;
         while(k->gr_mem[i] != NULL)
               printf(" Mitglied : %s\n",k->gr_mem[i++]);
       }
}
```

Dateien

/etc/group Gruppeneinträge; Eingabedatei für getgrnam

/usr/include/grp.h
Definition der Struktur group

> > > > getgrent, getgrgid, setgrent, endgrent

Login-Namen abfragen

char *getlogin()

getlogin liefert den login-Namen des Aufrufers, der in /etc/utmp eingetragen ist.

Typ

C-Funktion

Parameter

keine

Ergebnis

Zeiger auf den login-Namen	bei Erfolg
Nullzeiger	Fehler, wenn – kein login-Name gefunden wurde oder – der Prozeß nicht an eine Datensichtstation angeschlossen ist.

Achtung

getlogin schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

In der Kombination getpwnam(getlogin()) kann getlogin dazu benutzt werden, um in /etc/passwd den richtigen Eintrag zu finden, falls mehrere login-Namen dieselbe Benutzernummer haben.

Beispiel

login-Name des Aufrufers:

```
char *getlogin();
main()
{
  printf("%s\n",getlogin());
}
```

Dateien

/etc/utmp Einträge der login-Namen; Eingabedatei für getlogin

>>>> getgrent, getpwent

Passwort einlesen

char *getpass(meldung**)**
char *meldung;

getpass gibt *meldung* auf dem Bildschirm aus, stellt das Echo ab und liest dann, falls möglich, von /dev/tty, sonst von der Standardeingabe, ein Passwort ein.

Typ

C-Funktion

Parameter

char *meldung
: Zeichenreihe, die vor Eingabe des Passwortes auf dem Bildschirm ausgegeben werden soll, z.B:
Kennwort:

Ergebnis

Zeiger auf das eingelesene Passwort
: Zeichenreihe, die mit '\0' abgeschlossen ist und aus höchstens 8 Zeichen besteht

Achtung

Der Ergebniszeiger zeigt auf einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird.

Beispiel

```
main()
{
  char *getpass();
  char *z;

  z = getpass("Geheimcode 007:");
  printf("%s\n",z);
}
```

Dateien

/dev/tty Kontrolldatei für zeichenorientierte Geräte

> > > > crypt

Prozeßgruppe abfragen

int getpgrp()

getpgrp liefert die Prozeßgruppennummer des aufrufenden Prozesses.

Typ

Systemaufruf

Parameter

keine

Ergebnis

Prozeßgruppennummer

> > > > getpid, getppid, setpgrp, exec, fork

Prozeßnummer abfragen

int getpid()

getpid liefert die Prozeßnummer des aktuellen Prozesses.

Typ

Systemaufruf

Parameter

keine

Ergebnis

Prozeßnummer

Hinweis

getpid wird häufig dazu verwendet, um einmalige Dateinamen für Temporärdateien zu erzeugen (siehe mktemp).

Beispiel

Prozeßnummer ausgeben:

```
main()
{
  printf("Prozessnummer: %d\n",getpid());
}
```

> > > > mktemp, getppid, getpgrp, setpgrp, exec, fork

Prozeßnummer des Vaters abfragen

int getppid()

getppid liefert die Prozeßnummer des Vaterprozesses.

Typ

Systemaufruf

Parameter

keine

Ergebnis

Prozeßnummer des Vaterprozesses

> > > > getpid, getpgrp, setpgrp, exec, fork

Eintrag in /etc/passwd suchen

int getpw(ben _ nr,puf**)**
int ben _ nr;
char *puf;

getpw sucht in der "Passwortdatei" /etc/passwd nach der Benutzernummer *ben _ nr* und speichert bei Erfolg die zugehörige Zeile in den Bereich, auf den *puf* zeigt.

Typ

C-Funktion

Parameter

int ben _ nr	ganze Zahl, die eine Benutzernummer darstellt
← char *puf	Zeiger für die Ergebniszeichenreihe; zeigt auf das erste Zeichen einer Zeile aus /etc/passwd

Ergebnis

0	getpw hat die Benutzernummer *ben _ nr* in /etc/passwd gefunden
ungleich 0	getpw hat die Benutzernummer nicht gefunden

Achtung

Den Speicherplatz für die Zeile aus /etc/passwd müssen Sie wie üblich explizit bereitstellen.

Hinweis

- getpw öffnet die "Passwortdatei" automatisch.
- Falls getpw die angegebene Benutzernummer findet, liefert es unter dem Zeiger *puf* die zugehörige Zeile aus /etc/passwd wie z.B.:

  sissi::18:1::/usr/sissi:bin/sh

- Aus Kompatibilitätsgründen sollten Sie die Funktionen getpwent, getpwnam und getpwuid verwenden.

Beispiel

Das Programm liefert zu einer beim Aufruf übergebenen Benutzernummer den Eintrag in /etc/passwd, falls er existiert:

```
#include <stdio.h>

main(argc,argv)
int argc;
char **argv;
{
  char buf[BUFSIZ];

  if((getpw( atoi(argv[1]),buf)) == 0)
      printf("%s\n",buf);
}
```

Dateien

/etc/passwd Liste aller Systembenutzer; Eingabedatei für getpw

> > > getpwent, getpwnam, getpwuid, setpwent, endpwent

Nächste Zeile von /etc/passwd ausgeben

#include <pwd.h>

struct passwd *getpwent()

getpwent liefert die nächste Zeile aus der "Passwortdatei" /etc/passwd.

Typ

C-Funktion

Parameter

keine

Ergebnis

Zeiger auf eine passwd Struktur
getpwent speichert die nächste Zeile aus /etc/passwd in eine Struktur, die wie folgt in <pwd.h> definiert ist:

```
struct  passwd {
        char    *pw_name;     /* Benutzername */
        char    *pw_passwd;   /* Passwort */
        int     pw_uid;       /* Benutzernummer */
        int     pw_gid;       /* Gruppennummer */
        int     pw_quota;     /* ungenutzt */
        char    *pw_comment;  /* Kommentar */
        char    *pw_gecos;    /* ungenutzt */
        char    *pw_dir;      /* Home-Datei- */
                              /* verzeichnis */
        char    *pw_shell;    /* Shell */
};
```

Nullzeiger   Dateiende oder Fehler

Achtung

getpwent schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

getpwent öffnet die "Passwortdatei" automatisch.

Beispiel

Das Programm druckt die Einträge der "Passwortdatei" mit entsprechenden Kommentaren.

```
#include <stdio.h>
#include <pwd.h>

int endpwent();
struct passwd *getpwent();
struct passwd *daten;

main()
{
  while((daten = getpwent()) != NULL)
  {
    printf(" Name des Benutzers: %s\n",daten->pw_name);
    printf(" Passwort: %s\n",daten->pw_passwd);
    printf(" Benutzernummer: %d\n",daten->pw_uid);
    printf(" Gruppennummer: %d\n",daten->pw_gid);
    printf(" Dateiverzeichnis: %s\n",daten->pw_dir);
    printf(" Shell: %s\n",daten->pw_shell);
    printf("\n\n\n");
  }
endpwent();
}
```

Dateien

/etc/passwd   Liste aller Systembenutzer; Eingabedatei für getpwent

/usr/include/pwd.h
    Definition der passwd Struktur

> > > > getpw, getpwuid, getpwnam, setpwent, endpwent, getlogin, getgrent

Benutzernamen in /etc/passwd suchen

#include <pwd.h>

struct passwd *getpwnam(name**)**
char *name;

getpwnam sucht in der "Passwortdatei" /etc/passwd den Benutzernamen *name* und liefert bei Erfolg die zugehörige Zeile zurück.

Typ

C-Funktion

Parameter

char *name   Benutzername, der in /etc/passwd gesucht werden soll.

Ergebnis

Zeiger auf passwd Struktur
Falls getpwnam den Benutzernamen findet, speichert es die zugehörige Zeile aus /etc/passwd in eine Struktur, die wie folgt in <pwd.h> definiert ist:

```
struct  passwd {
        char    *pw_name;     /* Benutzername */
        char    *pw_passwd;   /* Passwort */
        int     pw_uid;       /* Benutzernummer */
        int     pw_gid;       /* Gruppennummer */
        int     pw_quota;     /* ungenutzt */
        char    *pw_comment;  /* ungenutzt */
        char    *pw_gecos;    /* Funktion */
        char    *pw_dir;      /* Home-Datei- */
                              /* verzeichnis */
        char    *pw_shell;    /* Shell */
};
```

Nullzeiger   Dateiende oder Fehler

Achtung

getpwnam schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

getpwnam öffnet die "Passwortdatei" automatisch.

Beispiel

Das Programm druckt für einen eingelesenen Benutzernamen die Einträge in /etc/passwd mit entsprechenden Kommentaren:

```
#include <stdio.h>
#include <pwd.h>

struct passwd *getpwnam();
struct passwd *daten;
char name[25];

main()
{
  printf("Über wen wollen sie die Information?\n");
  scanf("%s",name);
  if ((daten = getpwnam(name)) != NULL);
     {
       printf("Benutzername : %s\n",daten->pw_name);
       printf("Passwort : %s\n",daten->pw_passwd);
       printf("Benutzernummer : %d\n",daten->pw_uid);
       printf("Gruppennummer : %d\n",daten->pw_gid);
       printf("Home-Dateiverzeichnis : %s\n",daten->pw_dir);
       printf("Shell : %s\n",daten->pw_shell);
     }
}
```

Dateien

/etc/passwd　Liste aller Systembenutzer; Eingabedatei für getpwnam

/usr/include/pwd.h
　Definition der passwd Struktur

>>>>getpwent, getpwuid, setpwent, endpwent, getlogin, getgrent, getpw

Benutzernummer in /etc/passwd suchen

#include <pwd.h>

struct passwd *getpwuid(nr)
int nr;

getpwuid sucht in der "Passwortdatei" /etc/passwd die Benutzernummer *nr* und liefert bei Erfolg die zugehörige Zeile zurück.

Typ

C-Funktion

Parameter

int nr — Benutzernummer, die in /etc/passwd gesucht werden soll.

Ergebnis

Zeiger auf passwd Struktur

Falls getpwnam die Benutzernummer findet, speichert es die zugehörige Zeile aus /etc/passwd in eine Struktur, die wie folgt in <pwd.h> definiert ist:

```
struct  passwd {
char    *pw_name;      /* Benutzername */
char    *pw_passwd;    /* Passwort */
int     pw_uid;        /* Benutzernummer */
int     pw_gid;        /* Gruppennummer */
int     pw_quota;      /* ungenutzt */
char    *pw_comment;   /* ungenutzt */
char    *pw_gecos;     /* Funktion */
char    *pw_dir;       /* Home-Datei- */
                       /* verzeichnis */
char    *pw_shell;     /* Shell */
};
```

Nullzeiger — Dateiende oder Fehler

Achtung

getpwuid schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

getpwuid öffnet die "Passwortdatei" automatisch.

Beispiel

Das Programm druckt für eine eingelesene Benutzernummer die Einträge in /etc/passwd mit entsprechenden Kommentaren:

```
#include <stdio.h>
#include <pwd.h>

struct passwd *getpwuid();
struct passwd *daten;
int i;

main()
{
  printf("Über wen wollen sie die Information?\n");
  scanf("%d",&i);
  if((daten = getpwuid(i)) != NULL)
    {
      printf("Benutzername : %s\n",daten->pw_name);
      printf("Passwort : %s\n",daten->pw_passwd);
      printf("Benutzernummer : %d\n",daten->pw_uid);
      printf("Gruppennummer : %d\n",daten->pw_gid);
      printf("Home-Dateiverzeichnis : %s\n",daten->pw_dir);
      printf("Shell : %s\n",daten->pw_shell);
    }
}
```

Dateien

/etc/passwd
Liste aller Systembenutzer; Eingabedatei für getpwuid

/usr/include/pwd.h
Definition der passwd Struktur

>>>>getpwent, getpwnam, setpwent, endpwent, getlogin, getgrent, getpw

Zeichenreihe von Standardeingabe einlesen

#include <stdio.h>

char *gets(s)
char *s;

gets liest von der Standardeingabe Zeichen bis zum nächsten Zeilenende und speichert die gelesene Zeichenreihe in *s*. Das Zeilenende wird ersetzt durch das Nullbyte (\0).

Typ

C-Funktion (s)

Parameter

← char *s	Ergebniszeichenreihe

Ergebnis

Ergebniszeichenreihe:	Zeiger auf das erste gelesene Zeichen gets schließt die eingelesene Zeichenreihe mit den Nullbyte (\0) ab.
Nullzeiger	Dateiende oder Fehler

Achtung

Den Speicherplatz für die Ergebniszeichenreihe müssen Sie explizit bereitstellen!

Hinweis

Im Unterschied zu fgets löscht gets das Zeilenende-Zeichen.

Beispiel

siehe Beispiel bei fgets

Dateien

/usr/include/stdio.h
Definitionen für Standardein/Ausgabe

> > > > fgets, getc, scanf, ungetc, getw, fread, fgetc

Reale Benutzernummer abfragen

int getuid()

getuid liefert die **reale** Benutzernummer des Prozesses. Diese Nummer ist die Benutzernummer des Aufrufers der Programmdatei.

Typ

Systemaufruf

Parameter

keine

Ergebnis

reale Benutzernummer

Hinweis

getuid ist nützlich bei Programmen, die das s-Bit für Eigentümer gesetzt haben, um die Benutzernummer des Programmaufrufers zu ermitteln.

Beispiel

Schreiben Sie die Objektdatei a.out zu diesem Programm unter Ihrer Kennung (nicht root) und setzen Sie mit dem chmod Kommando
(z.B. chmod 4777 a.out) das s-Bit für Eigentümer.
Rufen Sie dann das lauffähige Programm a.out unter einer anderen Kennung (z.B. root) auf.

```
main()
{

  printf("reale Benutzernr:%d\neffektive Benutzernr:%d\n",getuid(),geteuid());
}
```

>>>> getegid, geteuid, getgid

Wort von Datei einlesen

#include <stdio.h>

int getw(dz)
FILE *dz;

getw liest ein Maschinenwort aus der Datei mit Dateizeiger *dz* und positioniert den Lese/Schreibzeiger hinter das gelesene Wort.

Typ

C-Funktion (s)

Parameter

FILE *dz	Eingabedatei

Ergebnis

gelesenes Wort als integer Wert
bei Erfolg

EOF Dateiende oder Fehler

Achtung

Weil Wortlänge und Anordnung der Bytes maschinenabhängig sind, können unter Umständen Dateien, die mit putw auf einem Prozessor beschrieben wurden, nicht mit getw auf einem anderen Prozessor gelesen werden.

Hinweis

Da getw Dateiende oder Fehler nicht explizit anzeigt, sollten Sie die Funktionen feof und ferror verwenden, um diese Bedingungen zu überprüfen.

Beispiel

Programmstück, das wortweise aus der Datei mit Dateizeiger *dz* einliest, bis Dateiende erreicht ist:

```
int buf[Max];
int i = 0;
File *dz;

while(!feof(dz))
     buf[i++] = getw(dz);
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/Ausgabe

> > > > fgets, getc, scanf, ungetc, gets, fread, fgetc

Aktuelle Zeit (GMT) als Struktur

#include <time.h>

struct tm *gmtime(sek_zg**)**
long *sek_zg;

gmtime interpretiert den Wert, auf den *sek_zg* zeigt, als Zeit in Sekunden seit dem 1.Januar 1970 00:00:00 (GMT). gmtime berechnet daraus Datum und Uhrzeit und speichert das Ergebnis in einer Struktur tm.

Typ

C-Funktion

Parameter

long *sek_zg
Zeiger auf die Zeitangabe in Sekunden

Ergebnis

Zeiger auf die berechnete Struktur
gmtime legt sein Ergebnis in einer Struktur ab, die wie folgt in <time.h> definiert ist:

```
struct tm{
int tm_sec;   /* Sekunden */
int tm_min;   /* Minuten */
int tm_hour;  /* Stunden (1-24) */
int tm_mday;  /* Monatstag(1-31) */
int tm_mon;   /* Monat (0-11) */
int tm_year;  /* Jahr (minus 1900) */
int tm_wday;  /* Wochentag (0-6, 0=Sonntag) */
int tm_yday;  /* Jahrestag  (0-365) */
int tm_isdst; /* Flag (immer 0) */
}
```

Achtung

- gmtime schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Außerdem verwenden gmtime und localtime **denselben** Datenbereich, d.h., wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben!

Hinweis

Der Aufruf:
clock = time(0l);
asctime(gmtime(&clock));
liefert die aktuelle Zeit (GMT) in englischem Standard, wie z.B.:
WED MAY 15 12:32:54 1985\n\0.

Der Aufruf
clock = time(0L);
meztime(gmtime(&clock));
liefert die aktuelle Zeit (GMT) in deutschem Standard, wie z.B.:
Mi 15.Mai.1985 12:32:54\n\0.

Beispiel

```
#include <time.h>

long time();
struct tm *gmtime();

struct tm *zeit;
char *daten;
long clock;

main()
{
  clock = time(0L);
  zeit = gmtime(&clock);
  printf("Jahr: 19%d\n",zeit->tm_year);
  printf("Uhrzeit in Stunden: %d\n",zeit->tm_hour);
  printf("Jahrestag: %d\n",zeit->tm_yday);
  daten=asctime(zeit);
  printf("%s",daten);
}
```

Dateien

/usr/include/time.h
Definition der Struktur tm.

> > > > ctime, time, asctime, localtime, timezone, gctime, meztime

Bildschirmeigenschaften abprüfen

#include <sgtty.h>

int gtty(dk,sssp**)**
int dk;
struct sgttyb *sssp;

gtty bestimmt die Eigenschaften der seriellen Schnittstelle, die über die Dateikennzahl *dk* angesprochen wird und legt die Werte in einer Struktur sgttyb ab.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl für die serielle Schnittstelle

← struct *sgttyb sssp
Zeiger auf die Struktur, in die gtty die Eigenschaften der seriellen Schnittstelle ablegt.
Die Struktur sgttyb ist in <sgtty.h> wie folgt definiert:

```
struct sgttyb {
char    sg_ispeed;    /* Eingabegeschwindigkeit */
char    sg_ospeed;    /* Ausgabegeschwindigkeit */
char    sg_erase;     /* Korrekturzeichen  */
char    sg_kill;      /* Abbruchzeichen */
int     sg_flags;     /* Merkmale */
};
```

Die einzelnen Bits der Komponente sg_flags sind wie folgt festgelegt:

```
#define TANDEM  01      /* automatische Flußkontrolle */
#define CBREAK  02      /* cbreak Modus: jedes Zeichen */
                        /* wird sofort weitergeleitet */
#define LCASE   04      /* Großbuchstaben in Klein- */
                        /* buchstaben umwandeln */
#define ECHO    010     /* Echo (vollduplex) */
#define CRMOD   020     /* CR auf LF setzen, CR und */
                        /* LF werden als CR-LF aus- */
                        /* gegeben */
#define RAW     040     /* raw Modus: keine Zeichen- */
                        /* bearbeitung (roh, 8-Bit) */
#define ODDP    0100    /* ungerade Parität */
#define EVENP   0200    /* gerade Parität */
#define ANYP    0300    /* keine Parität, aber */
                        /* 7-Bit-Übertragung */
#define NLDELAY 001400 /* Verzögerung für NL */
#define TBDELAY 006000 /* Verzögerung für TAB */
#define XTABS   06000  /* Tabulator expandieren */
#define CRDELAY 030000 /* Verzögerung für CR */
#define VTDELAY 040000  /* Verzögerung für FF */
                        /* VT */
#define BSDELAY 0100000 /* Verzögerung für BS */
#define ALLDELAY 0177400 /* Maske */
```

die Kürzel (CR,LF,...) haben die übliche Bedeutung (Wagenrücklauf, Zeileneinschub,...).

Ergebnis

0	bei Erfolg
-1	sonst

Achtung

Wie immer in solchen Fällen müssen Sie den Speicherplatz für die Ergebnisstruktur explizit bereitstellen!

Hinweis

Ist die serielle Schnittstelle ein Bildschirm, dann sind die Aufrufe gtty(dk,sssp) und ioctl(dk,TIOCGETP,sssp) äquivalent.

Beispiel

Zusammen mit stty kann gtty dazu verwendet werden, Bildschirmeigenschaften neu zu definieren.
Wir zeigen, wie Sie vom Programm aus, das Echo abschalten und den cbreak-Modus einschalten können:

```
#include <sgtty.h>

int stty();
int gtty();

struct sgttyb bild;

main()
{
   struct sgttyb *bilds;
   bilds = &bild;
   gtty(0,bilds);
   bilds->sg_flags &=  ~ECHO;
   bilds->sg_flags |= CBREAK;
   stty(0,bilds);
}
```

Nach Ablauf dieses Programms wird jedes Eingabezeichen sofort weitergeleitet und nicht mehr auf dem Bildschirm ausgegeben.

Dateien

/usr/include/sgtty.h
Definition der Struktur sgttyb

/dev/tty — Kontrolldatei

/dev/tty* — Gerätedatei für die jeweilige Datensichtstation

/dev/console
Gerätedatei für die Konsole

> > > > ioctl, stty, stty(Kommando)

Euklidischer Abstand

#include <math.h>

double hypot(x,y**)**
double x,y;

hypot berechnet den euklidischen Abstand

Typ

C-Funktion

Parameter

double x,y	Koordinaten des Punktes, dessen Abstand berechnet werden soll

Ergebnis

sqrt(x*x + y*y)
Wurzel aus dem Quadrat der Koordinaten

Beispiel

```
#include <stdio.h>
#include <math.h>

main()
{
  double x,y;
  double alpha, r, pi;

  printf("Koordinaten x und y eingeben:");
  scanf("%lf %lf",&x,&y);

  pi = 2 * asin(1.0);

  if(x > 0.0)
    alpfa = atan(y/x);
  else if (x < 0.0)
```

```
          if (y >= 0.0)
             alpha = atan(y/x) + pi;
          else alpha = atan(y/x) - pi;
       else if (y > 0)
              alpfa = pi/2.0;
            else if (y < 0)
                   alpha = -pi/2.0
                 else
                 {
                   printf("Winkel nicht definiert!");
                   exit(1);
                 }

  r = hypot(x,y);
  alpha = alpha * (180.0/pi);

  printf("Die Polarkoordinaten lauten:\n");
  printf("Abstand vom Nullpunkt: %g\n",r);
  printf("Winkel zur x-Achse:");
  printf("%g Grad\n",((y < 0.0)? alpha + 360 : alpha) );
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

>>> cabs, sqrt

Erstes Auftreten eines Zeichens in einer Zeichenreihe

char *index(s,c)
char *s, c;

index sucht das erste Vorkommen des Zeichens *c* in der Zeichenreihe *s* und liefert bei Erfolg einen Zeiger auf die Position in *s.*

Typ

C-Funktion

Parameter

char *s	Zeichenreihe, in der das Zeichen gesucht werden soll
char c	Zeichen, das gesucht werden soll

Ergebnis

Zeiger auf das	erste Vorkommen von c falls *c* in *s* vorkommt
Nullzeiger	sonst

Beispiel

Finde das doppelte 'k':

```
main()
{
   char *s = "was für ein Spaß im kkühlen Naß!";
   printf("%s\n",s);
   printf("Wo steckt der Fehler? %s\n",index(s,'k'));
}
```

> > > > rindex, strcat, strcmp, strcpy, strlen, strdup,

Geräteschnittstelle

int ioctl(dk,aktion,arg)
int dk, aktion;
<typ> arg;

ioctl ist die (low level) Schnittstelle für zeichenorientierte Geräte.
Es gibt eine Vielzahl von Funktionen, die - abgestimmt auf die speziellen Eigenschaften eines Gerätes - die Kontrolle eines Gerätes ermöglichen.
Da ioctl äußerst implementierungsabhängig ist, ist eine genauere Beschreibung im Rahmen dieses Manuals nicht sinnvoll.
Wenn Sie tatsächlich den Zugang zu einem Gerät über ioctl wünschen, müssen Sie zusätzlich Spezialliteraratur hinzuziehen, in der die an Ihrem System angeschlossenen Geräte und die darauf abgestimmten ioctl Funktionen genau beschrieben sind.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl für eine (geöffnete) Gerätedatei
int aktion	symbolische Konstante, mit der Sie eine Funktion auswählen, die auf die speziellen Eigenschaften des Gerätes abgestimmt ist, das Sie mit *dk* ansprechen.
<typ> arg	Datentyp und Wert des Argumentes *arg* hängen von dem Gerät und der in *aktion* angegebenen Funktion ab.

Ergebnis

0 der Aufruf war erfolgreich

–1 Fehler, wenn

- *dk* keine gültige Dateikennzahl ist oder
- *dk* kein zeichenorientiertes Gerät bezeichnet oder
- *aktion* oder *arg* unzulässige Werte haben oder
- der Aufruf durch ein Signal unterbrochen wurde.

Fehlermeldung

Bei Ergebnis -1 wird in errno ein entsprechender Fehlercode abgelegt:

EBDAF : Unzulässige Dateinummer
ENOTTY : Nur bei zeichenorientierten Geräten möglich
EINVAL : Unzulässiges Argument
EINTR : Systemaufruf wurde unterbrochen

Dateien

/dev/tty Kontrolldatei für zeichenorientierte Geräte

> > > > gtty,stty

Buchstabe oder Ziffer?

#include <ctype.h>

int isalnum(c)
int c;

isalnum überprüft, ob das Zeichen *c* alphanumerisch ist, d.h. ein Buchstabe (A-Z oder a-z) oder eine Ziffer (0-9).

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist alphanumerisch
0	das Zeichen ist nicht alphanumerisch

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isalnum(c))? "alphanumerisch " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalpha, isascii, iscntrl, isdigit, islower, isprint, ispunct, isspace isupper, isxdigit

Buchstabe?

```
#include <ctype.h>

int isalpha(c)
int c;
```

isalpha überprüft, ob das Zeichen *c* ein Buchstabe (A-Z oder a-z) ist.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein Buchstabe
0	das Zeichen ist kein Buchstabe

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isalpha(c))? "Buchstabe " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isdigit, islower, isprint, ispunct, isspace
isupper, isxdigit

ASCII Zeichen?

#include <ctype.h>

int isascii(c)
int c;

isascii überprüft, ob das Zeichen *c* ein ASCII Zeichen ist, d.h. ein Zeichen, dessen Oktalwert zwischen 000 und 177 liegt.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein ASCII Zeichen
0	das Zeichen ist kein ASCII Zeichen

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isascii(c))? "ASCII ZEICHEN " : "SONSTIGES"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isalpha, iscntrl, isdigit, islower, isprint, ispunct, isspace isupper, isxdigit

Dateikennzahl mit Gerätedatei verbunden?

int isatty(dk**)**
int dk;

isatty prüft, ob die Dateikennzahl *dk* einer Gerätedatei für ein zeichenorientiertes Gerät zugeordnet ist.

Typ

C-Funktion

Parameter

int dk	Dateikennzahl

Ergebnis

1	*dk* ist die Dateikennzahl einer Gerätedatei für ein zeichenorientiertes Gerät.
0	sonst

Hinweis

isatty überprüft, ob über die Dateikennzahl *dk* ein zeichenorientiertes Gerät angesprochen wird. Es liefert daher auch das Ergebnis 1 für Gerätedateien, die einem Drucker oder Magnetband zugeordnet sind.

Beispiel

siehe Beispiel bei ttyname

> > > > ttyname, ttyslot

Kontrollzeichen?

```
#include <ctype.h>

int iscntrl(c)
int c;
```

iscntrl überprüft, ob das Zeichen *c* ein Kontrollzeichen ist, d.h. einen Oktalwert zwischen 000 und 037 oder 177 hat.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein Kontrollzeichen
0	das Zeichen ist kein Kontrollzeichen

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((iscntrl(c))? "Kontrollzeichen":"Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, isalpha, isdigit, islower, isprint, ispunct, isspace isupper, isxdigit

Ziffer?

#include <ctype.h>

int isdigit(c)
int c;

isdigit überprüft, ob das Zeichen *c* eine Ziffer (0-9) ist.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist eine Ziffer
0	das Zeichen ist keine Ziffer

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isdigit(c))? "Ziffer " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isalpha, islower, isprint, ispunct, isspace isupper, isxdigit

Kleinbuchstabe?

```
#include <ctype.h>

int islower(c)
int c;
```

islower überprüft, ob das Zeichen *c* ein Kleinbuchstabe (a-z) ist.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein Kleinbuchstabe
0	das Zeichen ist kein Kleinbuchstabe

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((islower(c))? "Kleinbuchstabe " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isdigit, isalpha, isprint, ispunct, isspace isupper, isxdigit

Abdruckbares Zeichen?

#include <ctype.h>

int isprint(c)
int c;

isprint überprüft, ob das Zeichen *c* ein abdruckbares Zeichen ist, d.h. einen Oktalwert zwischen 040 (Leerzeichen) und 176 (ß) hat.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein abdruckbares Zeichen
0	das Zeichen ist kein abdruckbares Zeichen

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isprint(c))? "Zeichen" : "kann nicht drucken"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > >isalnum, isascii, iscntrl, isdigit, islower, isalpha, ispunct, isspace
isupper, isxdigit

Sonderzeichen?

```
#include <ctype.h>

int ispunct(c)
int c;
```

ispunct überprüft, ob das Zeichen *c* ein Sonderzeichen ist, d.h. weder ein Kontrollzeichen noch ein alphanumerisches Zeichen.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein Sonderzeichen
0	das Zeichen ist kein Sonderzeichen

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((ispunct(c))? "Sonderzeichen" : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isdigit, islower, isalpha, isprint, isspace isupper, isxdigit

Zwischenraum?

```
#include <ctype.h>

int isspace(c)
int c;
```

isspace überprüft, ob das Zeichen *c* ein Zwischenraum ist, d.h. ein Leerzeichen (040), Tabulator (011, 013), Wagenrücklauf (015), Zeilenvorschub (012) oder Seitenvorschub (014) ist.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein Zwischenraum
0	das Zeichen ist kein Zwischenraum

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isspace(c))? "Zwischenraum " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isdigit, islower, isprint, ispunct, isalpha, isupper, isxdigit

Großbuchstabe?

#include <ctype.h>

int isupper(c)
int c;

isupper überprüft, ob das Zeichen *c* ein Großbuchstabe (A-Z) ist.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist ein Großbuchstabe
0	das Zeichen ist kein Großbuchstabe

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isupper(c))? "Großbuchstabe " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isdigit, islower, isprint, ispunct, isspace, isalpha, isxdigit

Hexadezimale Ziffer?

```
#include <ctype.h>

int isxdigit(c)
int c;
```

isxdigit überprüft, ob das Zeichen *c* eine hexadezimale Ziffer [0-9], [A-F] oder [a-f] ist.

Typ

Makro

Parameter

int c	Eingabezeichen

Ergebnis

ungleich 0	das Zeichen ist eine hexadezimale Ziffer
0	das Zeichen ist keine hexadezimale Ziffer

Hinweis

Die Makros zur Zeichenklassifizierung bis auf isascii sollten nur auf Zeichen des ASCII-Zeichensatzes angewendet werden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

main()
{
  int c;
  while((c = getchar()) != EOF)
  printf("%s : %c\n",((isxdigit(c))? "Hexadezimale Ziffer " : "Sonstiges"),c);
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenklassifizierung

> > > > isalnum, isascii, iscntrl, isalpha, islower, isprint, ispunct, isspace isupper, isdigit

Besselfunktionen der ersten Art

#include < math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n,x)
int n;
double x;

Die Funktionen j0, j1 und jn berechnen die Besselfunktionen der ersten Art für Gleitkommawerte x und ganzzahlige Ordnungen 0, 1 bzw. *n*.

Typ

C-Funktion

Parameter

double x	Gleitkommawert
int n	ganzzahlige Ordnung

Ergebnis

Besselfunktion für x

Hinweis

Wenn Sie in Ihrem Programm eine Besselfunktion verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

>>>> y0, y1, yn

Signal an Prozesse schicken

```
#include <signal.h>

int kill(pnr,sig)
int pnr;
int sig;
```

kill sendet das Signal *sig* an einen Prozeß oder eine Prozeßgruppe.
Die effektive Benutzernummer des sendenden Prozesses muß i.a. mit der realen Benutzernummer des empfangenden Prozesses übereinstimmen. Dies gilt nicht, falls der Prozeß sich selbst ein Signal schickt oder die effektive Benutzernummer die des Systemverwalters ist.
Prozesse mit der Prozeßnummer 0 und 1 sind spezielle Systemprozesse.

Typ

Systemaufruf

Parameter

int pnr	ganze Zahl, mit der Sie angeben, welche Prozesse das Signal empfangen sollen. Sie haben folgende Möglichkeiten:
> 0	das Signal wird an den Prozeß mit Prozeßnummer *pnr* (auch 1 möglich) geschickt.
0	das Signal wird an alle Prozesse aus der Prozeßgruppe des Senders (siehe setpgrp) geschickt, außer an die speziellen Systemprozesse.
-1	falls die effektive Benutzernummer des Senders nicht die des Systemverwalters ist, wird das Signal an alle Prozesse geschickt, deren reale Benutzernummer mit der effektiven Benutzernummer des Senders übereinstimmt, aber nicht an die speziellen Systemprozesse.
	falls die effektive Benutzernummer die des Systemverwalters ist, wird das Signal an alle Prozesse außer die speziellen Systemprozesse geschickt.

< -1	das Signal wird an alle Prozesse geschickt, deren Prozeßgruppennummer mit dem Absolutbetrag von *pnr* übereinstimmt
int sig	Signal, das an den Prozeß geschickt werden soll. Sie haben folgende Möglichkeiten (siehe <signal.h>):

0	es wird kein Signal geschickt, sondern abgeprüft, ob *pnr* eine gültige Prozeßnummer ist
SIGHUP 1	Verbindung zu einer Datensichtstation (V24) unterbrochen
SIGINT 2	Unterbrechung von der Datensichtstation: Drücken der Taste [DEL]
SIGQUIT 3	Abbruch von der Datensichtstation: Drücken derTaste [CTRL]
SIGILL 4	unzulässiger Befehl
SIGTRAP 5	ptrace : Unterbrechungssignal
SIGIOT 6	IOT Befehl
SIGEMT 7	EMT Befehl
SIGFPE 8	Fehler bei einer Gleitkommaoperation
SIGKILL 9	kill: unbedingter Prozeßabbruch kann weder abgefangen noch ignoriert werden!
SIGBUS 10	Adreßfehler auf dem System-Bus
SIGSEGV 11	Adreßfehler wegen unerlaubtem Segmentzugriff
SIGSYS 12	ungültiges Argument für einen Systemaufruf
SIGPIPE 13	Ausgabe auf eine Pipe, deren Leseseite geschlossen ist
SIGALRM 14	alarm: Ablauf einer Zeitspanne
SIGTERM 15	Programmbeendigung wird von kill benützt
SIGUSR1 16	vom Benutzer definiert
SIGUSR2 17	vom Benutzer definiert
SIGCLD 18	Sohnprozeß gestorben
SIGPWR 19	Stromausfall
SIGDVZ 20	Division durch 0
SIGXCPU 21	CPU-Zeit überschritten
SIGPROF 22	profil: Unterbrechungssignal
SIGBPT 23	Haltepunkt
SIGNMI 24	Unterbrechung bei MMU debugging

Ergebnis

0 das Signal wurde erfolgreich geschickt

-1 das Signal konnte nicht gesendet werden, weil

- *sig* keine gültige Signalnummer ist oder
- der Prozeß weder sich selbst ein Signal schickt noch die effektive Benutzernummer des Systemverwalters hat und die effektive Benutzernummer nicht mit der realen des Empfängers übereinstimmt oder
- kein Prozeß existiert, auf den *pnr* zutrifft.

Fehlermeldung

Im Fehlerfall wird errno mit einem entsprechenden Fehlercode besetzt:

EINVAL : Unzulässiges Argument
EPERM : Hat anderen Eigentümer
ESRCH : Prozeß unbekannt

Beispiel

Ein Programm, das sich selbst abbricht:

```
main()
{
 for(;;)
    kill(getpid(),9);
}
```

Dateien

/usr/include/signal.h
Definition der Signale

> > > > signal, getpid, setpgrp, kill(Kommando)

Umwandlung in long Integer

void l3tol(longp,dreip,n**)**
long *longp;
char *dreip;
int n;

dreip zeigt auf eine Liste von *n* 3 Bytes langen Integer.
l3tol wandelt jedes Listenelement in einen long Integer und gibt einen Zeiger auf den Anfang der Liste zurück.

Typ

C-Funktion

Parameter

← long *longp
Zeiger auf die Ergebnisliste der long Integer

char *dreip
Zeiger auf die 3 Bytes langen Integer

int n — Anzahl der Listenelemente

Hinweis

Plattenadressen im Indexeintrag einer Datei sind 3 Bytes lang.
l3tol und ltol3 können zur Verwaltung von Dateisystemen eingesetzt werden.

Dateien

/usr/include/sys/filsys.h
Defintionen für die Implementierung des Dateisystems

/usr/include/sys/inode.h
Definitionen für Indexeinträge

> > > > ltol3

Wert im Zweiersystem berechnen

#include <math.h>

double ldexp(wert, ex**)**
double wert;
int ex;

ldexp berechnet aus seinen Argumenten die Zahl:

wert * 2***ex*

Typ

C-Funktion

Parameter

double wert
Mantisse

int ex ganzzahliger Exponent

Ergebnis

wert * 2**ex

Hinweis

Wenn Sie in Ihrem Programm ldexp verwenden, müssen Sie den Überset-
zer mit:
cc progname -lm aufrufen.

Beispiel

ldexp ist die Umkehrfunktion zu frexp:
frexp zerlegt sein Gleitkommaargument in Mantisse und Exponent zur Basis 2, und ldexp berechnet aus diesen Teilen wieder den ursprünglichen Wert im Dezimalsystem, hier vorgeführt für die Zahl 5.342:

```
#include <math.h>

double frexp();

int ex;

main()
{
  double x;

  x = frexp(5.3421,&ex);
  printf("Mantisse : %f\nExponent : %d\n",x,ex);
  printf("Ausgangswert : %f\n",ldexp( x,ex));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > frexp, modf

Verweis auf eine Datei einrichten

int link(name1,name2**)**
char *name1, *name2;

link richtet für die Datei *name1* einen neuen Verweis *name2* ein.
Ein Verweis auf eine Datei ist ein Eintrag in einem Dateiverzeichnis.
Beide Namen dürfen beliebige Pfadnamen sein.

Typ

Systemaufruf

Parameter

char *name1

Name der Datei, auf die Sie einen Verweis eintragen wollen. Diese Datei muß bereits vorhanden sein.
Auf Dateiverzeichnisse darf nur der Systemverwalter mehrfach verweisen!

char *name2

neuer Name für die Datei
Ist kein Pfadname angegeben, wird der Verweis im aktuellen Dateiverzeichnis eingetragen.

Ergebnis

0 link war erfolgreich und hat den Verweis eingetragen

–1 link hat keinen Verweis eingetragen, weil einer (oder mehrere) der folgenden Fehler auftrat:

- *name1* ist nicht vorhanden, oder
  name1 oder *name2* ist kein gültiger Pfadname
- den Verweis *name2* gibt es bereits
- ein Dateiverzeichnis auf dem Pfad von *name1* oder *name2* darf nicht durchsucht werden, oder
  für das Dateiverzeichnis, in das der Verweis eingetragen werden soll, besteht keine Schreiberlaubnis

- der Benutzer ist nicht Systemverwalter und versucht, einen Verweis auf ein Dateiverzeichnis zu machen
- *name1* und *name2* stehen nicht in demselben Dateisystem
- es sind bereits {LINK_MAX} Verweise auf die Datei eingetragen.

Fehlermeldung

Liefert link das Ergebnis -1, so wird zusätzlich in errno ein Fehlercode abgelegt und zwar:

ENOENT : Datei oder Dateiverzeichnis unbekannt
EEXIST : Datei existiert
EACCES : Zugriff untersagt
EPERM : Hat anderen Eigentümer
EXDEV : Unzulässige Referenz über Gerätegrenzen
EMLINK : Zu viele Referenzen

Hinweis

Das Kommando rm löscht nur einen Verweis. Erst mit dem letzten Verweis löscht rm die Datei wirklich.

Beispiel

Peter möchte seinem Freund Carsten das "superspiel" zukommen lassen:

```
main()
{
   int i;
   i = link("superspiel","/usr/carsten/p_superspiel");
   printf("%s : %d\n",((!i)? "Freu dich Carsten!":"es geht leider nicht"),i);
}
```

Damit sein Wunsch auch in Erfüllung geht, muß Peter dieses Programm schreiben und natürlich das "superspiel" in seinem Dateiverzeichnis haben und berechtigt sein, in Carsten's Dateiverzeichnis zu schreiben!

> > > > unlink, ln(Kommando)

Datum und Uhrzeit (MEZ) berechnen

#include <time.h>

struct tm *localtime(sek_zg**)**
long *sek_zg;

localtime interpretiert den Wert, auf den *sek_zg* zeigt, als Zeit in Sekunden seit dem 1.Januar 1970 00:00:00 (GMT). Es berechnet daraus Datum und Uhrzeit (MEZ) und legt das Ergebnis in einer Struktur tm ab.

Typ

C-Funktion

Parameter

long *sek_zg

Zeiger auf die Zeitangabe in Sekunden

Ergebnis

Zeiger auf die berechnete Struktur

localtime legt sein Ergebnis in einer Struktur ab, die wie folgt in <time.h> definiert ist:

```
struct tm {
       int tm_sec;   /* Sekunden */
       int tm_min;   /* Minuten */
       int tm_hour;  /* Stunden (1-24) */
       int tm_mday;  /* Monatstag(1-31) */
       int tm_mon;   /* Monat (0-11) */
       int tm_year;  /* Jahr (minus 1900) */
       int tm_wday;  /* Wochentag (0-6, 0=Sonntag) */
       int tm_yday;  /* Jahrestag  (0-365) */
       int tm_isdst; /* Flag (immer 0) */
}
```

Achtung

- localtime schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Außerdem verwenden gmtime und localtime **denselben** Datenbereich, d.h., wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben!

Hinweis

- localtime und ctime machen dasselbe, sie unterscheiden sich lediglich im Ausgabeformat.
- Die Aufrufe asctime(localtime(sek_zg)) und ctime(sek_zg) sind äquivalent.
- Die Aufrufe meztime(localtime(sek_zg)) und gctime(sek_zg) sind äquivalent.

Beispiel

```
#include <time.h>

long time();
struct tm *localtime();

struct tm *zeit;
long clock;

main()
{
  clock = time(0L);
  zeit = localtime(&clock);
  printf("Jahr: 19%d\n",zeit->tm_year);
  printf("Uhrzeit in Stunden: %d\n",zeit->tm_hour);
  printf("Jahrestag: %d\n",zeit->tm_yday);
}
```

Dateien

/usr/include/time.h
Definition der Struktur tm

> > > > ctime, time, asctime, gmtime, timezone, gctime, meztime

Teilbereiche einer Datei sperren oder freigeben

#include <sys/locking.h>

int locking(dk,modus,anz**)**
int dk, modus;
long anz;

locking ermöglicht es, einem Prozeß Teilbereiche einer Datei ausschließlich zur Verfügung zu stellen und sie nach Gebrauch wieder freizugeben. Andere Prozesse, die auf gesperrte Bereiche zugreifen (lesen oder schreiben) wollen, müssen solange warten bis der Sperrensetzer die Bereiche wieder freigibt.

Typ

Systemaufruf
SINIX spezifisch

Parameter

int dk — Dateikennzahl der Datei, die gesperrt werden soll

int modus — Mit *modus* geben Sie an, wie Sie die Datei sperren bzw. freigeben wollen.
Es gibt folgende fünf Möglichkeiten, die symbolischen Konstanten sind in <sys/locking.h> definiert:

LK_UNLK 0
Freigabe
Der aufrufende Prozeß gibt einen Teilbereich frei, den er zuvor gesperrt hatte.

LK_LOCK 1
Sperren für **fremde Lese- oder Schreibzugriffe**;
Der angegebene Bereich wird dem aufrufenden Prozeß ausschließlich zur Verfügung gestellt.
Andere Prozesse, die auf denselben Bereich zugreifen wollen, schlafen solange, bis der Sperrensetzer die Sperre wieder aufhebt.

Sollten beim Aufruf von locking Teile des Bereichs noch von einem anderen Prozeß gesperrt sein, so schläft der Sperrenanwärter bis der Bereich vollständig freigegeben wurde.

LK_NBLCK 2

Sperren für **fremde Lese/Schreibzugriffe ohne Blockierungsgefahr**
Falls ein anderer Prozeß noch Teile des angeforderten Bereichs gesperrt hält, liefert locking das Ergebnis -1 und in errno:
EACCES : Zugriff untersagt.
Im Gegensatz zu LK_LOCK wartet hier also der Sperrenanwärter nicht auf die Freigabe eines noch gesperrten Bereiches (non-blocking).

LK_RLCK 3

Sperren für **fremde Schreibzugriffe**
Andere Prozesse dürfen den gesperrten Bereich weiterhin lesen, ansonsten ist die Wirkung wie bei LK_LOCK.

LK_NBRLCK 4

Sperren für **fremde Schreibzugriffe ohne Blockierungsgefahr**
Die Wirkung ist wie bei LK_NBLCK, außer daß wie bei LK_RLCK andere Prozesse den gesperrten Bereich weiterhin lesen dürfen.

long anz

Größe des Bereiches in Bytes
Der Bereich ist zusammenhängend und beginnt bei der aktuellen Position des Lese/Schreibzeigers. Falls

aktuelle Position + *anz* > Dateiende,

darf lediglich der aufrufende Prozeß auf den Bereich hinter Dateiende zugreifen.

0L

die ganze Datei wird gesperrt

Ergebnis

0 der angeforderte Bereich konnte gesperrt/freigegeben werden

−1 Fehler, falls:

- Modus LK_NBLCK oder LK_NBRLCK angegeben wurde und Teile des angeforderten Bereichs noch gesperrt sind oder
  dk die Dateikennzahl eines Dateiverzeichnisses ist oder
- eine Verklemmung ausgelöst würde (s. Hinweis) oder die Grenze für maximal zulässige Sperren {LOCK_MAX} erreicht ist.

Fehlermeldung

Liefert locking das Ergebnis -1, so wird in errno zusätzlich ein Fehlercode abgelegt und zwar:

EACCES : Zugriff untersagt
EDEADLOCK : Gefahr einer Systemverklemmung

Achtung

- Nur der Prozeß, der einen Bereich gesperrt hat, kann diesen auch wieder freigeben!
- Dateiverzeichnisse dürfen nicht gesperrt werden!

Hinweis

- Falls ein Prozeß, der einen Bereich gesperrt hält, auch noch auf Bereiche zugreifen will, die andere Prozesse gesperrt haben, kann es zu Verklemmungen kommen. Um dies zu verhindern, prüfen die Funktionen locking, read und write ab, ob durch das Warten auf die Freigabe eines gesperrten Bereichs eine Verklemmung ausgelöst werden kann und liefern gegebenenfalls ein entsprechendes Fehlerergebnis.

- Eine unzulässige Freigabe hat keine Wirkung.

- Sperren auf Gerätedateien oder Pipes lösen keinen Fehler aus, sind aber für Lese/Schreiboperationen wirkungslos.

- Sollten aufeinanderfolgende Sperranforderungen desselben Prozesses überlappende Bereiche betreffen, so wird daraus ein zusammenhängender Sperrbereich gemacht. Was den Sperrmodus betrifft, gibt es zwei Möglichkeiten:

  a) stimmen die Anforderungen in *modus* überein, dann erhält der ganze Sperrbereich diesen Modus.
  b) stimmen die Anforderungen nicht überein, dann wird die letzte Anforderung vorangig erfüllt, z.B:

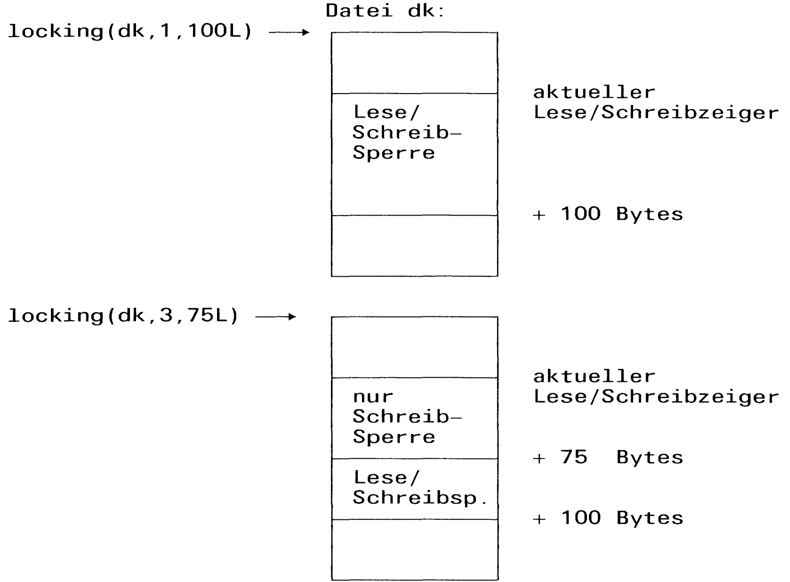


Bild 2-8 Wirkung von zwei aufeinanderfolgenden Sperranforderungen

- Teilfreigaben lösen die Sperre nur in den Bereichen, die angegeben sind.
  Soll allerdings ein mittlerer Bereich freigegeben werden, so wird zusätzlich ein gesperrtes Element benötigt, um die jetzt getrennten Restbereiche zu verwalten. Dabei kann es vorkommen, daß die Tabelle für die Sperreinträge bereits voll ist. Dies führt zu einem Fehlerergebnis und der Bereich wird nicht freigegeben.

- Bei Prozeßbeendigung werden alle gesperrten Bereiche eines Prozesses automatisch freigegeben.

Beispiel

Der Sohnprozeß erzeugt die Datei *dat* und beschreibt sie.
Der Vaterprozeß wartet solange bis der Sohn *dat* freigibt und liest dann die Datei:

```
#include <stdio.h>

FILE *dz1,*dz2;
int c;

main()
{
  int pid;

  switch(pid = fork()) {
                          /* Fehler */
        case -1:  break;
        case 0 :          /* Sohn */
                  dz2=fopen("dat","w");
                          /* "dat" für den Vater sperren */
                  locking(fileno(dz2),1,0L);
                  printf("erwarte eingabe\n");
                  setbuf(dz2,NULL);
                  while((c=getchar()) != EOF)
                        putc(c,dz2);
                          /* Sperre aufheben */
                  locking(fileno(dz2), 0,0L);
                  fclose(dz2);
                  exit(1);
       default:           /* Vater */
                          /* Warten, damit der Sohn
                             seine Sperre setzen kann */
                 sleep(2);
                          /* Vater liest erst, wenn der
                             Sohn die Datei wieder freigibt */
                 dz1=fopen("dat","r");
                 while((c=getc(dz1)) != EOF)
                       putchar(c);
                 fclose(dz1);
                 break;
  }
}
```

Dateien

/usr/include/sys/locking.h
Definition der Modi für den locking Aufruf

>>>> read, write

Natürlicher Logarithmus

#include <math.h>

double log(x)
double x;

log berechnet den natürlichen Logarithmus zur Basis e.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl

Ergebnis

log(x)	für positive *x* im zulässigen Gleitkommaintervall
0	falls *x* kleiner oder gleich 0 ist

Fehlermeldung

Bei einem Fehler steht in errno der Fehlercode

EDOM : Argument zu groß

Hinweis

Wenn Sie in Ihrem Programm log verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Dateien

/usr/include/math.h
Deklaration von mathematischen Funktionen

>>>> log10, exp

Logarithmus zur Basis 10

#include <math.h>

double log10(x**)**
double x;

log10 berechnet den Logarithmus zur Basis 10.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl

Ergebnis

log(x)	für positive *x* im zulässigen Gleitkommaintervall
0	falls *x* kleiner oder gleich 0 ist

Fehlermeldung

Bei einem Fehler steht in errno der Fehlercode:

EDOM : Argument zu groß

Hinweis

Wenn Sie in Ihrem Programm log10 verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Dateien

/usr/include/math.h
Deklaration von mathematischen Funktionen

>>>> log, exp

Nicht lokaler Sprung

#include <setjmp.h>

void longjmp(zst,wert**)**
jmp_buf zst;
int wert;

longjmp und setjmp sind Funktionen, die bei der Unterbrechungsbehandlung eingesetzt werden (siehe signal).
Mit Hilfe dieser Funktionen können Sie bestimmen, an welcher Stelle das Programm nach einer Unterbrechungsbehandlung wieder fortgesetzt werden soll. longjmp setzt auf der Funktion setjmp auf. Bei einem setjmp Aufruf wird der aktuelle Programmzustand (Adresse im Laufzeitkeller, Befehlszähler, Registerinhalte) in einer Variablen vom Typ jmp_buf gespeichert.
Erfolgt dann später im Programm in einer Fehler- oder Unterbrechungsbehandlung ein longjmp Aufruf, dann wird der Keller bis zu der durch setjmp markierten Stelle geleert (pop) und die Programmausführung beginnt wieder mit der Anweisung, die dem setjmp Aufruf unmittelbar folgt.
Die Wirkung von longjmp kann man mit der eines Rücksprungs mittels "goto" vergleichen. Man spricht deswegen auch von einem nicht-lokalen Sprung.

Typ

C-Funktion

Parameter

jmp_buf zst
: Feld, in das setjmp seine Werte abgelegt hat. Der Typ jmp_buf ist in <setjmp.h> definiert.

int wert
: ganze Zahl, die beim Wiederaufsetzen der Programmausführung als Rückgabewert des setjmp Aufrufs interpretiert wird.

Hinweis

- Wenn longjmp mit einem Argument *zst* aufgerufen wird, das nicht zuvor durch einen setjmp Aufruf belegt wurde, so dürfen Sie sich auf ein absolutes Chaos gefaßt machen!
- das Programm, das setjmp aufruft, muß natürlich noch aktiv sein, wenn nach dem longjmp Aufruf die Programmausführung fortgesetzt werden soll.
- Beim Wiederaufsetzen der Programmausführung sind die Programmvariablen wie nach einem "goto" belegt:
  - globale Variablen haben die Werte, die sie zum Zeitpunkt des longjmp Aufrufes hatten
  - Registervariablen sind undefiniert
  - sonstige lokale Variablen werden u.U. erneut initialisiert.

Beispiel

Einen typischen Anwendungsfall für (setjmp, longjmp) kann man sich in einem interaktiven Texteditor vorstellen, der Textausgabe macht.
Wird während der Ausgabe das Programm unterbrochen, indem von außen ein Signal geschickt wird (z.B. Drücken der Taste [DEL]), dann heißt das, daß die Textausgabe beendet wird, ansonsten aber der Texteditor weitermachen soll.
Wie das mit setjmp und longjmp realisiert werden kann, sehen Sie in folgendem Programm (nur Signalbehandlung kein Editor!):

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

FILE *dz;
jmp_buf zst;

interrupt()
{
   printf("\n   ***** Sie wollen den Text nicht? ****** \n");
   longjmp(zst,0);
}

main()
{
   int c;
   char antwort;
   setjmp(zst);
   signal(SIGINT,interrupt);
   printf("Textausgabe? (j|n):\n");
   scanf("%1s",&antwort);
   if(antwort == 'j')
     {
     dz = fopen("locking.c","r");
     while((c=getc(dz)) != EOF)
           putc(c,stdout);
     }
   else printf("dann eben nicht\n");
}
```

Dateien

/usr/include/setjmp.h
    Typdefinition für jmp_buf

>>>> setjmp, signal

Lese/Schreibzeiger positionieren

long lseek(dk,distanz,ort**)**
int dk;
long distanz;
int ort;

lseek positioniert den Lese/Schreibzeiger für die Datei mit Dateikennzahl *dk* gemäß den Angaben in *distanz* und *ort*. Sie haben damit die Möglichkeit, eine Datei auch nicht-sequentiell zu bearbeiten.

Typ

Systemaufruf

Parameter

int dk
: Dateikennzahl der Datei, deren Lese/Schreibzeiger positioniert weden soll.

long distanz
: Anzahl der Bytes, um die der aktuelle Lese/Schreibzeiger verschoben werden soll.

  positive Zahl
  : Verschiebung Richtung Dateiende

  negative Zahl
  : Verschiebung Richtung Dateianfang

int ort
: gibt an, wo die Verschiebung stattfinden soll:

  0 Dateianfang
  1 aktuelle Position
  2 Dateiende

Ergebnis

neue Position des Lese/Schreibzeigers
bei Erfolg

–1 Fehler, falls:

- *dk* keiner geöffneten Datei zugeordnet ist oder
- *dk* einer Pipe zugeordnet ist oder
- vor den Dateianfang positioniert werden soll oder
- *ort* ungültig ist

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EBDAF : Unzulässige Dateinummer
ESPIPE : Unzulässige Positionierung
EINVAL : Unzulässiges Argument
EINVAL : Unzulässiges Argument

und Signal

SIGSYS : Ungültiges Argument für Systemaufruf

Hinweis

- Wenn Sie weit hinter das Dateiende positionieren, entsteht ein "Loch" zwischen den letzen physikalisch gespeicherten Daten und den neu geschriebenen Daten. Lesen aus dem "Loch" liefert 0.
- lseek hat keinen Effekt bei Gerätedateien
- Die Aufrufe lseek(dk, 0L, 1) und tell(dk) sind äquivalent.

Beispiel

Das Programm liest ab Position 10 aus der Datei, die als erstes Argument beim Aufruf übergeben wird und fügt den Inhalt ans Ende der Datei an, falls ein zweites Argument angegeben wird oder schreibt auf Standardausgabe.

```
#include <stdio.h>

long lseek();
int dk1, dk2, ort;
long distanz;
char c;

main(argc,argv)
int argc;
char **argv;
{
      if((dk1 = open (argv[1],0)) < 0) exit(1);
      if(argc < 3)
              dk2 = 1;
      else
              dk2 = creat(argv[2], 2);
      lseek(dk1,10L,0);
      printf("aktuelle Position in Datei1 : %ld\n",tell(dk1));
      lseek(dk2,0L,2);
      while(read(dk1, &c, 1) > 0)
              write(dk2, &c, 1);
      close(dk1);
      close(dk2);
}
```

> > > > tell, fseek, ftell

Umwandlung in 3 Bytes lange Integer

void ltol3(dreip,longp,n)
char *dreip;
long *longp;
int n;

ltol3 ist die Umkehrfunktion zu l3tol.
longp zeigt auf eine Liste von *n* long Integer.
ltol3 wandelt jedes Listenelement in einen 3 Bytes langen Integer und legt einen Zeiger auf den Anfang der Liste in *dreip* ab.

Typ

C-Funktion

Parameter

← char *dreip
Zeiger auf die Ergebnisliste der 3 Bytes langen Integer

char *longp
Zeiger auf die long Integer

int n
Anzahl der Listenelemente

Hinweis

Plattenadressen in einem Indexeintrag einer Datei sind 3 Bytes lang.
ltol3 und l3tol können zur Verwaltung von Dateisystemen eingesetzt werden.

Dateien

/usr/include/sys/filsys.h
Definitionen für die Implementierung des Dateisystems

>>>> l3tol

Speicherplatz reservieren

char *malloc(anz**)**
unsigned anz;

malloc beschafft zur Ausführungszeit zusammenhängenden Speicherplatz. Der neue Datenbereich ist *anz* Bytes groß.
Wenn malloc in der Liste der freien Blöcke nicht genug Platz findet, ruft es sbrk auf, um vom System mehr Speicher zu bekommen.

Typ

C-Funktion

Parameter

unsigned anz
angeforderte Größe in Bytes

Ergebnis

Zeiger auf den Anfang des neuen Speicherbereichs
falls malloc neuen Speicherplatz zuweisen konnte; der Zeiger kann für einen beliebigen Datentyp verwendet werden

Nullzeiger
malloc konnte den Platz nicht beschaffen, weil

- der noch vorhandene Speicherplatz für die Anforderung nicht ausreicht oder
- ein Fehler auftrat

Hinweis

- Der neue Datenbereich beginnt auf Wortgrenze.
- Um sicherzugehen, daß Sie ausreichend Platz für eine Variable anfordern, sollten Sie die Funktion sizeof verwenden.

Beispiel

Dynamische Speicherplatzreservierung für einen Gebrauchtwagen:

```
#include <stdio.h>
#define MAX 20;

struct pkw {
       char *typ;
       int alter;
       long kilometer;
       char tuev[6];
       int zust;
       int preis;
       struct pkw *n;
       } *list;

char *malloc();
char *calloc();

main()
{
  int mark;
  if((list = (struct pkw *) malloc(sizeof(*list))) == NULL)
    {
      printf("speicherplatz aufgebraucht\n");
      exit(1);
    }
                          /* achtung !!
                            Beim vorigen malloc Aufruf wurde
                            für die Komponente typ lediglich der
                            Platz für einen Zeiger (2 Bytes)
                            bereitgestellt . Der Platz für die Typbezeich-
                            nung selbst muß noch beschafft werden: */

  if((list->typ = calloc(1,20)) == NULL)
      exit(1);            /* Fehler */

                          /* Gebrauchtwagen einlesen */
  scanf("%20s %d",list->typ,&list->alter);
  scanf("%d %6s %d %d",&list->kilometer, list->tuev, &list->zust, &list->preis);
  list->n = NULL;

                          /* eingelesene Werte ausdrucken */
  printf("%s\n%d\n",list->typ,list->alter);
  printf("%d\n%.6s\n%d\n%d",list->kilometer,list->tuev,list->zust,list->preis);

                          /* Speicherplatz wieder freigeben */
  free(list);
}
```


> > > > calloc, free, realloc

Datum mit Uhrzeit in Deutsch

#include <time.h>

char *meztime(tm**)**
struct tm *tm;

meztime ist das deutsche Gegenstück zu asctime.
meztime wandelt eine gemäß der Struktur tm aufgeschlüsselte Zeitangabe in eine ASCII-Zeichenreihe um mit dem Format einer deutschen Datum-mit-Uhrzeit-Angabe:

```
                   Wochentag  Tag.Monat.Jahr, Std:Min:Sek
zum Beispiel:             FR   21.  Jun.1985,  13: 54: 32\n\0
```

Typ

C-Funktion

Parameter

struct tm *tm

Zeiger auf eine Struktur tm, die eine Zeitangabe enthält.
Die Struktur ist in <time.h> wie folgt definiert:

```
struct tm{
      int tm_sec;   /* Sekunden */
      int tm_min;   /* Minuten */
      int tm_hour;  /* Stunden (1-24) */
      int tm_mday;  /* Monatstag(1-31) */
      int tm_mon;   /* Monat (0-11) */
      int tm_year;  /* Jahr (minus 1900) */
      int tm_wday;  /* Wochentag (0-6, 0=Sonntag) */
      int tm_yday;  /* Jahrestag  (0-365) */
      int tm_isdst; /* nicht unterstützt */
}
```

Ergebnis

Zeiger auf die erzeugte ASCII-Zeichenreihe

Die Ergebniszeichenreihe hat die Länge 26 und das gleiche Format wie bei gctime

Achtung

- meztime schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!
- Außerdem verwenden meztime und asctime denselben Datenbereich d.h., wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben!

Hinweis

Die Aufrufe meztime(localtime(sek_zg)) und gctime(sek_zg) sind äquivalent.

Beispiel

Aktuelles Datum und Uhrzeit (MEZ) in Englisch und Deutsch ausgeben:

```
#include <time.h>

long time();
char *asctime();
char *meztime();
struct tm *localtime();
struct tm *zeit;
char *daten;
long clock;

main()
{
   clock = time(0L);
   zeit = localtime(&clock);
   printf("Jahr: 19%d\n",zeit->tm_year);
   printf("Uhrzeit in Stunden: %d\n",zeit->tm_hour);
   printf("Jahrestag: %d\n",zeit->tm_yday);
   daten=asctime(zeit);
   printf("%s\n",daten);
   printf("%s",meztime(zeit));
}
```

Dateien

/usr/include/time.h
Definition der Struktur tm.

> > > > asctime, gctime, ctime, localtime, gmtime, datum(Kommando)

Gerätedatei oder Dateiverzeichnis einrichten

int mknod(name,modus,ger**)**
char *name;
int modus, ger;

Nur für den Systemverwalter!
mknod richtet eine neue Datei mit Dateinamen *name* ein, deren Eigenschaften gemäß *modus* festgelegt werden. Im Gegensatz zu creat kann der Systemverwalter mit mknod Dateiverzeichnisse und Gerätedateien einrichten.

Typ

Systemaufruf

Parameter

char *name — Dateiname der neuen Datei

int modus — *modus* ist eine fünfstellige Oktalzahl, die den Typ und die Zugriffsrechte der neuen Datei festgelegt:
Die höchste Ziffer bestimmt den Typ nach folgender Tabelle:

Oktalzahl	Bedeutung
000000	normale Datei
020000	zeichenorientiertes Gerät
040000	Dateiverzeichnis
060000	blockorientiertes Gerät

Die zweit höchste Stelle gibt s- und sticky-Bit an und wird wie bei chmod und creat nach folgender Tabelle gedeutet:

Oktalzahl	Bedeutung
001000	sticky-Bit (Datei sharable)
002000	s-Bit für Gruppe

004000	s-Bit für Eigentümer

Aus den restlichen drei Ziffern errechnen sich wie bei creat die Zugriffsrechte für Eigentümer, Gruppe und Andere:
Die tatsächliche Schutzbitbelegung ergibt sich aus dem bitweisen UND der Binärdarstellung dieser Ziffern und dem Komplement der Prozeßmaske.

Standardmaske: 002 — keine Schreiberlaubnis für Andere (siehe auch umask).

Das Ergebnis (als Oktalzahl) wird wie bei chmod nach folgender Tabelle gedeutet:

Oktalziffer		Zugriffsrecht
000400	Eigentümer:	lesen
000200		schreiben
000100		ausführen/durchsuchen
000040	Gruppe:	lesen
000020		schreiben
000010		ausführen/durchsuchen
000004	Andere:	lesen
000002		schreiben
000001		ausführen/durchsuchen

int ger — Angabe des Gerätes, falls eine Gerätedatei eingerichtet werden soll.
Die erste Blockadresse im Indexeintrag der neuen Datei wird mit *ger* initialisiert:

0 — für normale Datei und Dateiverzeichnis

major- und minor-Nummer
für Gerätedatei

Ergebnis

0	die Datei wurde eingerichtet
-1	Fehler, wenn

- die effektive Benutzernummer des Prozesses nicht die des Systemverwalters ist oder
- eine Komponente in *name* kein Dateiverzeichnis ist oder
- eine Komponente auf dem Pfad zu *name* nicht existiert oder
- ein Dateiverzeichnis auf dem Pfad zu *name* nicht durchsucht werden darf oder
- die Datei bereits existiert

Fehlermeldung

Bei Ergebnis -1 wird in errno ein entsprechender Fehlercode abgelegt:

EPERM :	Hat anderen Eigentümer
ENOTDIR :	Kein Dateiverzeichnis
ENOENT :	Datei oder Dateiverzeichnis unbekannt
EACCES :	Zugriff untersagt
EEXIST :	Datei existiert

Achtung

Im Unterschied zu dem Kommando mkdir trägt der Systemaufruf mknod beim Einrichten eines Dateiverzeichnisses die beiden Verweise auf das übergeordnete und auf das Verzeichnis selbst nicht ein!
Es ist sehr ratsam, diese Verweise mit link einzurichten, da Sie sonst mit dem neu erzeugten Dateiverzeichnis nicht vernünftig arbeiten können!

Hinweis

Für die neue Datei wird die Datei-Eigentümer-Kennung auf die effektive Benutzernummer des Prozesses gesetzt.

Beispiel

Folgendes Programm kann nur unter der Kennung des Systemverwalters laufen!

Es richtet für die Benutzerin *sissi* ein neues Dateiverzeichnis *frust* ein, wobei Gruppe und Andere das Verzeichnis lesen und durchsuchen dürfen. Weil Sie mit link wie oben erwähnt die Verweise auf das neue Verzeichnis selbst und auf das Vaterverzeichnis eintragen sollten, müssen Sie folgendes Programm unter der Kennung des Systemverwalters laufen lassen (siehe link)!

```
#include <stdio.h>

main()
{
   if(mknod("usr/sissi/frust",040755,0) == 0)
      {
       printf("korrekt\n");
       link("/usr/sissi/frust","/usr/sissi/frust/.");
       link("/usr/sissi","/usr/sissi/frust/..");
      }
   else printf("Fehler\n");
}
```

Dateien

/usr/include/sys/param.h
: Definitionen für major- und minor-Nummer einer Gerätedatei

>>>> link, mkdir(Kommando), /etc/mknod(Kommando)

Eindeutiger Dateiname

char *mktemp(vorlage**)**
char *vorlage;

mktemp erzeugt aus der Zeichenreihe *vorlage* einen eindeutigen Dateinamen, indem es sechs 'X' am Ende der Zeichenreihe durch die Prozeßnummer ersetzt.

Typ

C-Funktion

Parameter

← char *vorlage
Zeichenreihe, die auf sechs 'X' endet, z.B.: "temp/testXXXXXX"

Ergebnis

Zeiger auf den neuen Dateinamen	bei Erfolg
Nullzeiger	falls kein eindeutiger Name erzeugt werden konnte

Beispiel

Erzeugung einer Datei mit Dateinamen "temp/testprozeßnummer". Anschließend wird Eingabe von stdin in die neue Datei geschrieben:

```
#include <stdio.h>

char *mktemp();
char vorlage[] = "temp/testXXXXXX";
FILE *fp;
int c;

main()
{
   printf("%s\n",mktemp(vorlage));
            /* Anlegen der neuen Datei und
               Öffnen zum Schreiben */
   fp = fopen(vorlage, "w");
   while((c=getc(stdin))!=EOF)
          putc(c, fp);
   fclose(fp);
}
```

Das Programm setzt voraus, daß es im aktuellen Dateiverzeichnis ein Unterverzeichnis *temp* gibt, für das schreiben erlaubt ist!

> > > > getpid

Aufspalten einer Zahl in ihren ganzzahligen und gebrochenen Teil

double modf(z,g_zg**)**
double z;
double *g_zg;

modf zerlegt eine Gleitkommazahl *z* in ihren ganzzahligen und gebrochenen Teil. Als Ergebnis liefert es den Bruchteil mit Vorzeichen. Der ganzzahlige Anteil kann indirekt über *g_zg* angesprochen werden.

Typ

C-Funktion

Parameter

double z Gleitkommazahl

← double *g_zg
Zeiger auf den ganzzahligen Anteil

Ergebnis

Bruchteil von z mit Vorzeichen

Achtung

Den Speicherplatz für den ganzzahligen Anteil müssen Sie explizit bereitstellen!

Beispiel

Zerlegung der Zahl -456.789:

```
#include <stdio.h>

double modf();
double g;

main()
{
  double x;
  x = modf(-456.789,&g);
  printf("Bruchteil : %g\nGanzteil  : %g\n",x,g);
}
```

> > > > frexp, ldexp

Statistische Auswertung einer Programmausführung

void monitor(u _ adr,o _ adr,puffer,p _ größe,nauf**)**;
int (*u _ adr**)()**, **(***o _ adr**)()**;
short puffer**[]**;
int p _ größe, nauf;

monitor ist eine benutzerfreundlichere Schnittstelle zu dem Systemaufruf profil. Sie können damit die Ausführung eines C-Programms statistisch auswerten.

Typ

C-Funktion

Parameter

int (*u _ adr) ()
: Zeiger auf eine Funktion:
  Das Ergebnis dieser Funktion wird die untere Adreßgrenze für die Programmauswertung

int (*o _ adr) ()
: Zeiger auf eine Funktion:
  Das Ergebnis dieser Funktion wird die obere Adreßgrenze für die Programmauswertung

← short puffer[]
: vom Benutzer bereitgestellter Puffer, in den monitor seine Daten schreibt und zwar Histogramme von

  - periodisch aufgenommenen Werten des Programmzählers und
  - Funktionsaufrufen
    Nur Funktionen, die mit dem -p Schalter übersetzt wurden, werden gezählt.

int p _ größe
: Größe des bereitgestellten Puffers

int nauf
: Anzahl der maximal möglichen Funktionsaufrufe

Hinweis

- Wenn Sie den Übersetzer mit cc progname -p aufrufen, so erfolgt automatisch ein monitor Aufruf mit geeigneten Parametern.
- Um das gesamte Programm auswerten zu lassen, genügen folgende Anweisungen:

```
extern etext();
...
monitor((int)2,etext,puffer,p_größe,nauf),
```

  da etext das Ende des Programmtextsegmentes kennzeichnet.
- Der Aufruf monitor(0) beendet die Auswertung und schreibt die Ergebnisse in die Datei mon.out.
- damit das Ergebnis bei kleinen, häufig gebrauchten Funktionen noch aussagekräftig bleibt, sollte der Puffer nicht viel kleiner sein als das auszuwertende Adreßintervall.

Externe Größen

Bei Programmstart sind drei externe Größen verfügbar, die die Endadressen der einzelnen Programmsegmente enthalten (siehe brk).
Eine davon ist:

extern etext()
: erste Adresse über dem Programmtextsegment

Dateien

mon.out
: Ergebnisdatei für den Aufruf monitor(0);
kann von dem Kommando prof ausgewertet werden

> > > > profil, cc(Kommando), prof(Kommando)

Dateisystem einhängen

int mount(gerät,name,modus**)**
char *gerät, *name;
int modus;

Nur für den Systemverwalter!

mount hängt in das bestehende Dateiverzeichnis *name* das Dateisystem auf dem blockorientierten Datenträger ein, der über die Gerätedatei *gerät* angesprochen wird.
Sobald das Dateisystem eingehängt ist, wird es wie ein Unterbaum behandelt. Jeder Bezug auf das Verzeichnis *name* verweist jetzt auf die Wurzel des eingehängten nichtresidenten Dateisystems. Lediglich Verweise (link) zwischen den beiden Dateisystemen sind nicht erlaubt.

Typ

Systemaufruf

Parameter

char *gerät
: Name der Gerätedatei, auf der sich das Dateisystem befindet, z.B.

/dev/fl2
: falls sich das Dateisystem auf einer Diskette befindet

char *name
: Name des Dateiverzeichnisses, in das das Dateisystem eingehängt werden soll. Das Dateiverzeichnis muß vorhanden sein und ist üblicherweise vor dem Einhängen leer. Falls nicht, kann auf den ursprünglichen Inhalt während des mount Vorganges nicht zugegriffen werden. Wenn das einzuhängende Dateisystem lediglich aus einer Datei besteht, kann *name* der Dateiname einer normalen Datei sein.

int modus
: ganze Zahl, die angibt, ob für das eingehängte Dateisystem Schreibberechtigung erteilt wird:

ungleich 0 — keine Schreiberlaubnis
0 — Schreiben ist erlaubt

Ergebnis

0	das Dateisystem wurde eingehängt
–1	das Dateisystem wurde nicht eingehängt, weil

- das Programm nicht unter der Kennung des Systemverwalters läuft oder
- auf *gerät* nicht zugegriffen werden kann oder *gerät* kein block-orientiertes Gerät bezeichnet oder
- das in *gerät* angegebene Gerät nicht existiert oder
- das Dateisystem auf *gerät* bereits eingehängt ist oder das Dateiverzeichnis *name* zur Zeit nicht frei ist oder bereits {MOUNT-MAX} Dateisysteme eingehängt sind oder
- *name* kein Dateiverzeichnis bezeichnet oder
- auf das Dateiverzeichnis *name* nicht zugegriffen werden darf oder
- das Dateisystem defekt ist.

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EPERM :	Hat anderen Eigentümer
ENOTBLK :	Nur bei block-orientierten Geräten möglich
ENXIO :	Gerät oder Adresse unbekannt
EBUSY :	Gerät oder Dateiverzeichnis noch nicht frei
ENOTDIR :	Kein Dateiverzeichnis
ENOENT :	Datei oder Dateiverzeichnis unbekannt
EUCLEAN :	Struktur des Dateisystems muß bereinigt werden

Achtung

Dateisysteme auf physikalisch schreib-geschützten Geräten oder auf Magnetplatte müssen **ohne** Schreiberlaubnis eingehängt werden. Denn sonst würden Fehler auftreten, selbst wenn Sie nicht ausdrücklich in das Dateisystem schreiben, nämlich wenn vom System die Zugriffszeiten auf den neuesten Stand gebracht werden.

Hinweis

Mit dem Kommando /etc/fsck können Sie ein Dateisystem bereinigen

Beispiel

Dateisystem von Diskette einhängen:

```
#include <errno.h>

main()
{
  extern int errno;
  extern int sys_nerr;
  extern char *sys_errlist[];

  if (mount("/dev/fl2", "disk", 0) == -1)
     perror("Fehler");
  else printf("Dateisysteme auf Diskette ist jetzt über Dateiverzeichnis \
             disk ansprechbar\n");
             /* Verarbeitung der Daten auf /dev/fl2 */
  .
  .
  .
             /* Dateisystem wieder abhängen */
  umount("/dev/fl2");
}
```

Dateien

/etc/mtab Liste aller eingehängten Dateisysteme

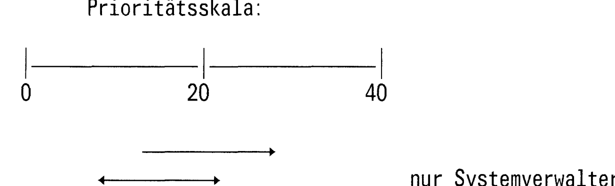
/usr/include/sys/param.h
Definition der Konstanten {MOUNT_MAX}

>>>> umount, /etc/mount(Kommando), /etc/mknod(Kommando), /etc/fsck(Kommando), /etc/mkfs(Kommando)

Priorität eines Prozesses ändern

int nice(zahl**)**
int zahl;

nice ändert die Priorität des aufrufenden Prozesses.
Priorität 20 ist voreingestellt.
Die niedrigste Priorität ist 39, die höchste 0.
Nur der Systemverwalter darf die Priorität erhöhen!



Typ

Systemaufruf

Parameter

int zahl — ganze Zahl, um die die aktuelle Priorität erniedrigt bzw. erhöht werden soll.

positive Zahl
Priorität erniedrigen

Nur Systemverwalter:

negative Zahl
Priorität erhöhen

Ergebnis

tatsächliche relative Veränderung
: bei Erfolg gibt nice die Zahl zurück, um die die aktuelle Priorität verändert wurde (s. Hinweis).

-1
: Wenn die Priorität erhöht werden soll, der Prozeß aber nicht unter der Kennung des Systemverwalters läuft.

Fehlermeldung

Bei Ergebnis -1 steht in errno der Fehlercode:

EPERM : Hat anderen Eigentümer

Hinweis

- Wenn durch die Angabe in *zahl* die Priorität außerhalb des zulässigen Intervalls [0,39] liegen würde, wird *zahl* automatisch erhöht bzw. erniedrigt, so daß die Priortät auf 0 bzw. 39 gesetzt wird.
- Für sehr große Programme wird die Priorität 10 empfohlen.
- Um priviligierte Prozesse wieder auf Normalstatus zurückzusetzen, sollten Sie (als Systemverwalter) nacheinander nice mit folgenden Argumenten aufrufen:

  –40: Erhöhung der Priorität
  Der Prozeß erhält dadurch auf alle Fälle die Priorität 0.

  20: setzt Priorität auf Standardeinstellung 20.
- Prioritäten werden bei fork vom Vater an den Sohn vererbt und bei exec an das aufgerufene Programm weitergegeben.

> > > > exec, fork, nice(Kommando)

Werte aus der Symboltabelle suchen

#include <a.out.h> (1.0B, 1.0C)
#include <nlist.h> (2.0)

void nlist(ld_name,nl**)**
char *ld_name;
struct nlist nl[];

nlist durchsucht die Symboltabelle in der ausführbaren Datei *ld_name*, die der Lader erzeugt hat, nach den Symbolnamen, die in den Elementen des Vektors *nl* aufgeführt sind.
Zu jedem Namen, den nlist in der Symboltabelle findet, trägt es den Symboltyp und den Symbolwert in der Struktur des entsprechenden Elementes von *nl* ein.

Typ

C-Funktion

Parameter

char *ld_name

ausführbare Datei mit Symboltabelle, die durchsucht werden soll

← struct nlist nl[]

Vektor, der die Symbolnamen enthält, die gesucht werden sollen. Jedes Vektorelement ist eine Struktur, die folgende Komponenten enthält:

n_name	Symbolname
n_type	Symbolklasse
n_value	Symbolwert

Die möglichen Werte für die Komponente n_type sind maschinenabhängig. Sie finden die notwendigen Definitionen in <a.out.h> (1.0B und 1.0C) bzw. <nlist.h> (2.0)

In die Komponente n_value (Symbolwert) wird die Adresse des Symbols gespeichert.

Der Vektor endet mit einem Element, dessen Komponente n_name statt einem Symbolnamen lediglich das Nullbyte ('\0') enthält.

Hinweis

- nlist setzt die n_type Komponente von jedem Vektorelement auf 0, falls
  - nlist die Datei *ld_name* nicht findet oder
  - *ld_name* keine Symboltabelle enthält oder
  - *nl* keine gültigen Namenseinträge enthält
- Sie können mit nlist aus der Datei /?????, in der die Namensliste des Systems steht, die aktuellen Systemadressen erfahren.
- nlist kann nur für globale Größen verwendet werden

Beispiel

Folgendes Programm zeigt, wie Sie die globale Variable *j* mit Hilfe von nlist in der Symboltabelle (in "datei") auffinden können. Die gefundenen Einträge werden auf Standardausgabe geschrieben.

```
#include <stdio.h>
#include <a.out.h>

struct nlist nl[2];
int j=6;

main()
{
  int i=0;
                             /* Initialisierung von nl mit _j */
  nl[0].n_name[0] = '_';
  nl[0].n_name[1] = 'j';
  nl[0].n_name[2] = '\0';
  nl[1].n_name[0] = '\0';
                             /* Suche j in der Symboltabelle */
  nlist("datei",nl);
                             /* Ausgabe der gefundenen Einträge */
  while(nl[i].n_name[0] != '\0')
       {
       printf("%s::%o::%u\n",nl[i].n_name,nl[i].n_type,nl[i].n_value);
       i++;
       }
}
```

Wenn Sie eine globale Variable *name* suchen wollen, müssen Sie den Vektor *nl* für nlist mit *_name* initialisieren!

Dateien

/usr/include/a.out.h
Definition der Struktur nlist und der Symbolklassen

>>>> adb(Kommando), nm(Kommando)

Datei öffnen (elementar)

```
#include <sys/fcntl.h>
```

```
int open(d_name,modus [, zugriff] )
char *d_name;
int modus, zugriff;
```

open öffnet die Datei *d_name* für elementare Lese/Schreiboperationen und positioniert den Lese/Schreibzeiger an den Dateianfang. open gibt eine gültige Dateikennzahl zurück, die später in elementaren Zugriffsoperationen (read, write) auf die Datei zur Bezeichnung der Datei benutzt wird.

Typ

Systemaufruf

Parameter

char *d_name
: Name der Datei, die geöffnet werden soll.
d_name kann ein beliebiger Pfadname sein.

int modus
: ganze Zahl, die in Form von Bits Informationen für den Dateistatus enthält.
Die einzelnen Möglichkeiten sind in der Datei <sys/fcntl.h> aufgelistet:

O_RDONLY 0000
: lesen

O_WRONLY 0001
: schreiben

O_RDWR 0002
: lesen und schreiben

O_NDELAY 0004
: Dieses Bit kann bei späteren read- und write-Aufrufen Auswirkungen haben (siehe read, write). Außerdem wirkt es sich beim Eröffnen einer Gerätedatei für ein zeichenorientiertes Gerät wie folgt aus:

  wenn O_NDELAY gesetzt ist:
  ein open Aufruf kehrt sofort mit -1 zurück, falls das Gerät nicht vorhanden ist.

  wenn O_NDELAY nicht gesetzt ist:
  ein open Aufruf blockiert solange, bis das entsprechende Gerät vorhanden ist.

O_APPEND 0010
: der Lese/Schreibzeiger wird vor jedem write auf Dateiende gesetzt

O_SYNCW 0100
: bei jedem write wird sofort physikalisch auf Platte geschrieben, d.h. die System-interne Pufferung wird ausgeschaltet

O_CREAT 00400
int zugriff
: wenn die Datei bereits existiert, hat dieses Bit keine Bedeutung, ansonsten wird die Datei wie bei creat neu eingerichtet:

  - die effektive Benutzer- bzw. Gruppennummer des Prozesses wird die Eigentümer- bzw. Gruppenkennung der neuen Datei

  - die Schutzbiteinstellung im Indexeintrag der neuen Datei errechnet sich wie bei creat aus den Bits in *zugriff* und der Prozeßmaske (alle Bits, die in der Prozeßmask gesetzt sind, werden gelöscht). Die resultierende Bitkombination wird wie in der Tabelle bei creat und chmod interpretiert (siehe creat, chmod, umask). Ist in *zugriff* das sticky-Bit gesetzt, wird es gelöscht.

O_TRUNC 01000
: wenn die Datei existiert, wird ihre Länge auf 0 gesetzt, Eigentümer und Zugriffsrechte bleiben.

O_EXCL 02000
wenn O_EXCL und O_CREAT gleichzeitig gesetzt sind, liefert open Fehler (-1), wenn die Datei bereits existiert.

Ergebnis

Dateikennzahl
bei Erfolg liefert open eine gültige Dateikennzahl, d.h. eine Zahl aus dem Intervall [0, {PDAT_MAX}]

–1
open kann *d_name* nicht öffnen, weil

- eine Komponente des Pfadnamens *d_name* kein Dateiverzeichnis ist oder
- die Datei nicht existiert oder
- ein Dateiverzeichnis auf dem Pfad zu *d_name* nicht gelesen werden darf oder
  die Schutzbitbelegung im Indexeintrag der Datei die gewünschte Zugriffsart nicht gestattet oder
- bereits {PDAT_MAX} Dateien geöffnet sind oder
- die Systemtabelle für die offenen Dateien im System voll ist ({SDAT_MAX}) oder
- ein Signal während dem open Aufruf abgefangen wurde (siehe signal) oder
- *d_name* ein Dateiverzeichnis ist und in *modus* O_WRONLY oder O_RDWR gesetzt oder
- *d_name* in einem "read-only"-Dateisystem steht und in *modus* O_WRONLY oder O_RDWR gesetzt ist oder
- *d_name* eine Gerätedatei ist und das entsprechende Gerät nicht existiert oder
- *d_name* ausführbaren Code enthält, der gerade ausgeführt wird und in *modus* O_WRONLY oder O_RDWR gesetzt ist oder
- in *modus* O_CREAT und O_EXCL gesetzt ist und die Datei bereits existiert oder
- in *modus* O_NDELAY und O_WRONLY gesetzt ist, und kein Prozeß die Datei zum Lesen geöffnet hat.

Fehlermeldung

Bei Ergebnis -1 wird in errno ein entsprechender Fehlercode abgelegt:

ENOTDIR : Kein Dateiverzeichnis
ENOENT : Datei oder Dateiverzeichnis unbekannt
EACCES : Zugriff untersagt
EMFILE : Zu viele offene Dateien im System
ENFILE : Überlauf der Dateien Tabelle im System
EINTR : Systemaufruf wurde unterbrochen
EISDIR : Ist ein Dateiverzeichnis
EROFS : Dateisystem darf nur gelesen werden
ENXIO : Gerät oder Adresse unbekannt
ETXTBSY : Programm wird gerade ausgeführt
EEXIST : Datei existiert
ENXIO : Gerät oder Adresse unbekannt

Hinweis

- Die Standarddateien für Eingabe, Ausgabe und Fehlerausgabe werden beim Start eines Prozesses automatisch mit folgenden Dateikennzahlen geöffnet:

  Standardeingabe : 0
  Standardausgabe : 1
  Fehlerausgabe : 2

- Nach einem fork Aufruf haben Vater- und Sohnprozeß einen gemeinsamen Lese/Schreibzeiger für jede Datei, die zur Zeit des fork Aufrufes geöffnet ist.
- Das sbe-Bit wird standardmäßig auf 0 gesetzt d.h., die Datei bleibt nach einem exec Aufruf geöffnet (siehe fcntl).
- Eine Folge von '/' in *d_name* wird zu einem '/' zusammengefaßt.
- Sie können dieselbe Datei mehrfach öffnen, indem Sie open mehrmals hintereinander mit demselben Dateinamen aufrufen. open liefert bei jedem Aufruf eine neue Dateikennzahl (siehe Beispiel).

Beispiel

Öffnen der Datei *späße* zum Lesen:

```
#include <stdio.h>

int dk1,dk2;
char c;
int n;

main()
{
                /* Datei "späße" das erste Mal
                   zum Lesen öffnen */
   if((dk1=open("späße",0)) == -1)
                /* Fehler beim ersten Öffnen */
      printf("fehler1\n");

                /* Datei "späße" zum
                   zum zweiten Mal zum Lesen öffnen */
   if((dk2=open("späße",0)) == -1)
                /* Fehler beim zweiten Öffnen */
      printf("fehler2\n");

                /* bis zum ersten 'a' wird über dk1 gelesen */
   while((n=read(dk1,&c,1)) > 0 && (c != 'a'))
                /* Ausgabe des gelesenen Zeichens auf
                   Standardausgabe */
         write(1,&c,n);

                /* jetzt wird über dk2 von Dateianfang!!
                   bis Dateiende gelesen */
   while((n=read(dk2,&c,1)) > 0)
                /* Ausgabe des gelesenen Zeichens auf
                   Standardausgabe */
         write(1,&c,n);

                /* über dk1 wird nach dem ersten 'a'
                   weitergelesen bis Dateiende */
   while((n=read(dk1,&c,1)) > 0)
                /* Ausgabe des gelesenen Zeichens auf
                   Standardausgabe */
         write(1,&c,n);
}
```

Dateien

/usr/include/sys/param.h
Definition der Konstanten {PDAT_MAX}

/usr/include/sys/fcntl.h
Definition der *modus*-Bits

>>>> creat, read, write, dup, dup2, fcntl, close, pipe

Prozeß anhalten bis Signal eintrifft

int pause()

pause hält den aufrufenden Prozeß solange an bis ein Signal eintrifft, das von kill oder alarm geschickt wurde.
Das Signal darf allerdings vom aufrufenden Prozeß nicht ignoriert werden!

Typ

Systemaufruf

Parameter

keine

Ergebnis

kein Ergebnis
Wenn das eintreffende Signal Prozeßabbruch bewirkt, kehrt pause natürlich nicht zurück

-1
Wenn das Signal abgefangen wird, geht nach der Signalbehandlung die Abarbeitung des Prozesses hinter dem pause Aufruf weiter (siehe signal).

Fehlermeldung

Bei Ergebnis -1 steht in errno der Fehlercode:

EINTR : Systemaufruf wurde unterbrochen

Beispiel

```
              /* Das Programm simuliert die Funktionsweise
                 von sleep mit Hilfe von pause */

#include <stdio.h>
#include <signal.h>

               /* Funktion zum Signal-Abfang */
abfang(sig)
int sig;
{
   printf("Signal wurde abgefangen !\n");
}

main()
{
   unsigned sek;
   int (*speich) ();
   printf("Prozess begonnen.\n");

               /* --------Simulationsanfang--------- */
               /* Zunächst wird die vorherige Signalbehandlung
                  gespeichert */
    speich = signal(SIGALRM,abfang);
    if((sek = alarm(15)) < 15);
       alarm(sek);
               /* Prozeß auf Eis bis SIGALRM eintrifft */
    pause();
               /* Alte Signalbehandlung wieder einstellen */
    signal(SIGALRM,speich);
    if(sek > 15)
       alarm(sek -15);
    else if(sek >0)
           alarm(1);
               /* --------Simulationsende--------- */
    printf("Prozess beendet.\n");
               /* Das obige Programmstueck enstpricht
                 einem sleep(15) Aufruf */
   sleep(15);
   printf("Nochmal 15 Sekunden gewartet !\n");
}
```

> > > > alarm, kill, kill(Kommando), signal, wait

Dateiverbindung zu einem Kommando schließen

#include <stdio.h>

int pclose(dz**)**
FILE *dz;

pclose schließt die Dateiverbindung über den Dateizeiger *dz*, die mittels popen zwischen dem aufrufendem Prozeß und einem Kommando eingerichtet wurde.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger, der von popen geliefert wurde

Ergebnis

Endestatus des mittels popen aufgerufenen Kommandos	bei Erfolg
-1	der Dateizeiger *dz* wurde nicht von popen eingerichtet.

Hinweis

Im Gegensatz zu fclose sollte man pclose auch vor exit immer explizit aufrufen, da sonst das aufgerufene Kommando möglicherweise erst nach dem aufrufenden Prozeß beendet wird.

Beispiel

siehe Beispiel bei popen

Dateien

/usr/include/stdio.h
    Defintionen für Standardein/ausgabe

> > > > popen, pipe, fopen, fclose, system, wait

Fehlermeldung ausgeben

extern int sys_nerr;
extern char *sys_errlist[];

void perror(s)
char *s;

perror schreibt auf die Standardfehlerausgabe eine Fehlermeldung, die ein fehlerhafter Systemaufruf in dem Programm verursacht hat.
Bei mehreren fehlerhaften Aufrufen wird nur der letzte berücksichtigt.
perror schreibt eine Zeile:

s : <kurze Fehlermeldung>

und schließt die Ausgabe mit Neue-Zeile ab.
Die Meldungen sind Standardfehlermeldungen, die vom System in dem externen Vektor sys_errlist eingetragen sind.

Typ

C-Funktion

Parameter

char *s	Zeichenreihe, die vor die Fehlermeldung geschrieben werden soll

Beispiel

Wenn Sie in Ihrem Dateiverzeichnis keine Datei *chaos* haben (weil bei Ihnen alles in Ordnung ist), dann liefert folgendes Programm den Ausdruck:

chaos: Datei oder Dateiverzeichnis unbekannt

```
int sys_nerr;
char *sys_errlist[];

main()
{
  FILE *dz,*fopen();
  if((fp = fopen("chaos", "r")) == NULL)
     perror("chaos");
}
```

Externe Größen

Das System stellt drei externe Größen bereit, die bei der Fehlerauswertung verwendet werden können:

char *sys_errlist[]
: Vektor, in dem die Standardfehlermeldungen stehen

int sys_nerr
: Anzahl der Einträge in sys_errlist

int errno
: Variable, die bei einem fehlerhaften Systemaufruf mit einem Fehlercode belegt wird.
sys_errlist[errno] enthält die entsprechende Fehlermeldung.
Dabei sollten Sie zuvor mit sys_nerr die Anzahl der Einträge überprüfen, da das System eventuell mehr Fehlercodes kennt, als Einträge in der Tabelle sind.

Dateien

/usr/include/errno.h
Definition der Fehlercodes. Derzeit sind 37 Fehlercodes definiert:

FEHLERCODE GEMÄS <errno.h>		FEHLERMELDUNG IN sys_errno[errno]
	0	Fehler 0
EPERM	1	Hat anderen Eigentuemer
ENOENT	2	Datei oder Dateiverzeichnis unbekannt
ESRCH	3	Prozess unbekannt
EINTR	4	Systemaufruf wurde unterbrochen
EIO	5	Ein/Ausgabe Fehler!
ENXIO	6	Geraet oder Adresse unbekannt
E2BIG	7	Liste der Argumente zu lang
ENOEXEC	8	Fehlerhaftes exec-Format
EBADF	9	Unzulaessige Dateinummer
ECHILD	10	Keine Kindprozesse
EAGAIN	11	Keine weiteren Prozesse moeglich
ENOMEM	12	Arbeitsspeicher unzureichend
EACCES	13	Zugriff untersagt
EFAULT	14	Unzulaessige Adresse
ENOTBLK	15	Nur bei block-orientierten Geraeten moeglich
EBUSY	16	Geraet oder Dateiverzeichnis noch nicht frei
EEXIST	17	Datei existiert
EXDEV	18	Unzulaessige Referenz ueber Geraetegrenzen
ENODEV	19	Geraet unbekannt
ENOTDIR	20	Kein Dateiverzeichnis
EISDIR	21	Ist ein Dateiverzeichnis
EINVAL	22	Unzulaessiges Argument
ENFILE	23	Ueberlauf der Dateien Tabelle im System
EMFILE	24	Zu viele offene Dateien im System
ENOTTY	25	Nur bei zeichen-orientierten Geraeten moeglich
ETXTBSY	26	Programm wird gerade ausgefuehrt
EFBIG	27	Datei zu gross
ENOSPC	28	Speicherkapazitaet erschoepft
ESPIPE	29	Unzulaessige Positionierung
EROFS	30	Dateisystem darf nur gelesen werden
EMLINK	31	Zu viele Referenzen
EPIPE	32	Pipeline unterbrochen
EDOM	33	Argument zu gross
ERANGE	34	Resultat zu gross
EUCLEAN	35	Struktur des Dateisystems muss bereinigt werden
EDEADLOCK	36	Gefahr einer Systemverklemmung
	37	nicht unterstuetzt
ENAVAIL	38	Nicht verfuegbar

Pipe einrichten

int pipe(pdk)
int pdk**[2]**;

pipe richtet eine Pipe ein, d.h. einen Einweg-Ein/Ausgabe-Kanal, über den kooperierende Prozesse Daten austauschen können.

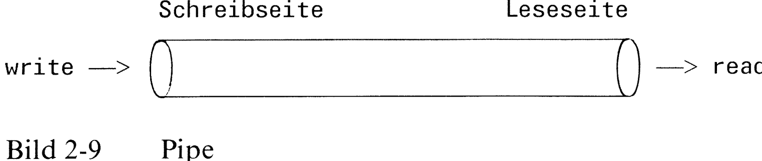


Bild 2-9 Pipe

Statt explizit Dateien zum Austausch der Daten einzurichten, wird beim Pipe-Mechanismus im Hauptspeicher ein Puffer für {PIPE_MAX} Bytes angelegt, über den die Daten nach dem Warteschlange-Prinzip (FIFO = first in, first out) transportiert werden.
Ein Prozeß darf entweder nur auf eine Pipe schreiben oder nur von einer Pipe lesen. Sollen Daten in beiden Richtungen ausgetauscht werden, dann müssen zwei Pipes eingerichtet werden.
Eine Pipe für zwei kooperierende Prozesse muß von einem Vater durch einen pipe Aufruf angelegt werden.
Die Prozeßbearbeitung wird dabei so gesteuert, daß ein Prozeß, der aus dem noch leeren Puffer lesen will, warten muß bis Daten in den Puffer geschrieben werden und ein Prozeß, der in den schon zu vollen Puffer schreiben will, warten muß, bis durch einen Lesevorgang Daten aus dem Puffer entfernt wurden.
Ähnlich wie open und creat liefert pipe als Verbindung zu dem Puffer Dateikennzahlen und zwar eine für die Leseseite (*pdk[0]*) und eine für die Schreibseite (*pdk[1]*). Wie bei einer gewöhnlichen Datei werden diese Dateikennzahlen in späteren elementaren Lese/Schreiboperationen als Dateiargument verwendet.

Typ

Systemaufruf

Parameter

← int pdk[2]

Vektor, in den pipe die Dateikennzahlen speichert:

pdk[0] : Dateikennzahl für Leseseite
pdk[1] : Dateikennzahl für Schreibseite

Ergebnis

0 es wurde eine Pipe eingerichtet

–1 es wurde keine Pipe eingerichtet, weil

- bereits {PDAT_MAX} Dateien geöffnet sind oder
- Die Systemtabelle für alle offenen Dateien im System bereits voll ist ({SDAT_MAX}).

SIGPIPE

Signal, das gesendet wird, falls auf eine Pipe geschrieben werden soll, deren Leseseite geschlossen ist

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EMFILE : Zu viele offene Dateien im System
ENFILE : Überlauf der Dateien-Tabelle im System

Hinweis

- Der Versuch, in eine Pipe zu schreiben, deren Leseseite geschlossen ist (kein angeschlossener Prozeß hat noch eine offene Leseverbindung), löst das Signal SIGPIPE aus.
- ein Prozeß, der über *pdk[1]* in die Pipe schreiben will, darf solange schreiben bis der Puffer voll ist bevor er angehalten wird. Dabei gilt ein Schreibvorgang (bis zu {PIPE_MAX} Bytes) als atomar, d.h. kein anderer Prozeß kann den Vorgang unterbrechen.
- Die Prozeßbearbeitung wird im allgemeinen von SINIX so gesteuert, daß nicht mehr als {PIPE_MAX} Bytes für die Pipe gespeichert werden müssen. Sollten mehr als {PIPE_MAX} Bytes benötigt werden, tritt eine Verklemmung ein.
- Ein Lesevorgang holt über *pdk[0]* die Daten aus dem Puffer.
  Ist der Puffer leer und die Schreibseite geschlossen (kein angeschlossener Prozeß hat noch eine offene Schreibverbindung), wird Dateiende zurückgegeben.
- Die offenen Pipes eines Prozesses sind Bestandteil seiner Umgebung und werden wie gewöhnliche Dateien vererbt:
  Nach einem fork Aufruf haben Vater- und Sohnprozeß Zugriff auf die Pipes, die vor dem fork Aufruf mittels pipe eingerichtet wurden.
  Nach einem exec Aufruf übernimmt das aufgerufene Programm die Pipes von dem aufrufenden Programm, das überlagert wird.
- Auf eine Pipe können Sie nicht mit lseek oder fseek positionieren.

Beispiel

Eine Einweg-Prozeßverbindung vom Vater zum Sohn baut man wie folgt auf:

1) Der Vaterprozeß erzeugt eine Pipe durch Aufruf pipe(pdk).
2) Der Vaterprozeß erzeugt einen Sohnprozeß mittels fork.
3) Der Vaterprozeß schließt die Leseseite der Pipe durch Aufruf close(pdk[0]).
4) Der Sohnprozeß schließt die Schreibseite der Pipe durch Aufruf close(pdk[1])

In folgendem Programm ist dieses Schema ausgeführt:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int pdk[2];
FILE *inpipe,*outpipe,*fdopen();
int sohn;                    /* Sohnprozeß */
int ch;
struct stat puffer;

main()
{
  if(pipe(pdk) != 0)
    exit(1);                 /* Pipe konnte nicht eroeffnet werden */

                             /* Information über die Pipe ausgeben */
  fstat(pdk[1],&puffer);
  printf("Benutzernummer : %d\n",puffer.st_uid);
  printf("Groesse in Bytes : %ld\n",puffer.st_size);
  printf("Indexnummer : %d\n",puffer.st_ino);

  switch(sohn=fork())
        {
                             /* Fehler */
        case -1 :  exit(1);
                   break;
        case  0 :            /* Sohn liest aus Pipe, quittiert den Empfang */
                             /* Dateizeiger für Leseseite einrichten */
                   if((inpipe = fdopen(pdk[0],"r")) == NULL)
                             /* Fehler */
                      exit(1);
                             /* stdin wird nicht mehr benötigt */
                   fclose(stdin);
                             /* Sohn schließt die Schreibseite */
```

```
                close(pdk[1]);
                        /* Pufferung umgehen,
                           damit der Effekt deutlicher wird */
                setbuf(inpipe,NULL);
                        /* Sobald der Vater die Schreibseite schließt,
                           empfängt der Sohn EOF */
                while((ch = getc(inpipe)) != EOF)
                    printf("zeichen empfangen \n");
                fclose(inpipe);
                exit(1);
        default :       /* Vater */
                        /* Dateizeiger für Schreibseite einrichten */
                if((outpipe = fdopen(pdk[1],"w")) == NULL)
                        /* Fehler */
                  exit(1);
                        /* stdout wird nicht mehr benötigt */
                fclose(stdout);
                        /* Leseseite schließen */
                close(pdk[0]);
                        /* Vater soll auf die Pipe schreiben */
                        /* Pufferung abstellen */
                setbuf(outpipe,NULL);
                while((ch=getchar()) != EOF)
                    putc(ch, outpipe);
                fclose(outpipe);
                        /* ======================================== */
                        /* Sohn erkennt EOF und terminiert sich auch */
                        /* ======================================== */
        }
}
```

Wenn Sie den Einwegkanal in die andere Richtung bauen wollen, müssen Sie nur die Rollen vertauschen.

> > > > read, write, sh(Kommando), popen, pclose, fork, exec

Kommando aufrufen und Pipe einrichten

#include <stdio.h>

FILE *popen(kmd,modus**)**
char *kmd, *modus;

popen erzeugt einen Sohnprozeß, der das shell-Kommando ausführt, das in der Zeichenreihe *kmd* steht und richtet eine Pipe ein zwischen dem aufrufenden Prozeß und dem Kommando.
Die Angabe in *modus* bestimmt die Richtung der Pipe.
Als Ergebnis liefert popen einen Dateizeiger, der dazu benutzt werden kann, auf die Standardeingabe des Kommandos zu schreiben bzw. von der Standardausgabe des Kommandos zu lesen, je nachdem, ob *modus* schreiben oder lesen angegeben wurde.

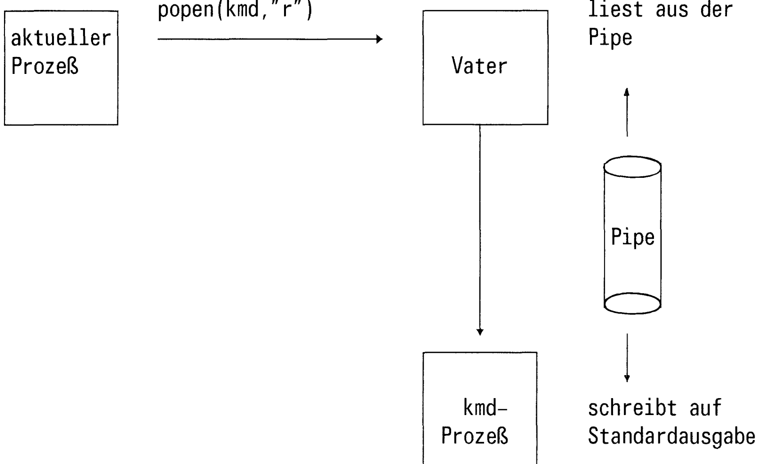


Bild 2-10 Anschauliche Darstellung für popen(kmd, "r")

Typ

C-Funktion (s)

Parameter

char *kmd — Zeichenreihe, die ein shell-Kommando enthält, etwa:
"wc -l" zum Zählen der Zeilen, die eingegeben werden

char *modus
Zeichenreihe, die die Richtung der Pipe angibt:

"r" — Pipe vom Kommando zum aufrufenden Prozeß;
der aufrufende Prozeß soll von der Standardausgabe des Kommandos lesen

"w" — Pipe vom aufrufenden Prozeß zum Kommando;
der aufrufende Prozeß soll auf die Standardeingabe des Kommandos schreiben

Ergebnis

Dateizeiger
bei Erfolg liefert popen einen Dateizeiger, der in späteren Zugriffsoperationen entsprechend *modus* als Dateiargument verwendet werden kann.

Nullzeiger — popen war nicht erfolgreich, weil

- keine Dateien geöffnet werden dürfen oder
- kein Prozeß erzeugt werden konnte oder
- das Kommando *kmd* nicht ausgeführt werden kann

Hinweis

Weil nach einem fork Aufruf Vater- und Sohnprozeß die geöffneten Dateien und speziell Standardein/ausgabe gemeinsam benützen, kann ein Programm mit popen(kmd,"r") als Eingabefilter und mit popen(kmd,"w") als Ausgabefilter verwendet werden.

Beispiel

a) Ein Text wird von der Standardeingabe gelesen und in die Datei *datei* geschrieben, wobei Großbuchstaben in Kleinbuchstaben verwandelt werden (Eingabefilter):

```
#include <stdio.h>

main()
{
  FILE *in,*popen(), *dz;
  int c;
  dz = fopen("datei","w");

               /* Pipe wird zum Lesen geoeffnet */
               /* der aufrufende Prozeß liest seine Eingabe
                  von der Standardausgabe des Kommandos tr */
  in = popen("tr \"A-Z\" \"a-z\"","r");
  while((c=getc(in)) != EOF)
     fprintf(dz,"%c",c);
  pclose(in);
  fclose(dz);
}
```

b) Ein Text aus *datei* wird auf Standardausgabe geschrieben, wobei alle Kleinbuchstaben in Großbuchstaben verwandelt werden (Ausgabefilter):

```
#include <stdio.h>

main()
{
  FILE *out, *popen(), *dz;
  int c;
  dz = fopen("datei","r");

               /* Pipe zum Schreiben öffnen */
  out = popen("tr \"a-z\" \"A-Z\"","w");
  while((c = getc(dz)) != EOF)
        putc(c,out);
  pclose(out);
  fclose(dz);
}
```

Dateien

/usr/include/stdio.h
Definition für Standard Ein/Ausgabe

> > > > pclose, system, wait, fopen, fclose

allgemeine Exponentialfunktion

#include <math.h>

double pow(x,y)
double x,y;

pow berechnet x**y.

Typ

C-Funktion

Parameter

double x	Basis
double y	Exponent falls *x* Null ist, muß *y* positiv sein, falls *x* negativ ist, muß *y* ganzzahlig sein

Ergebnis

x**y	falls *x*, *y* und das Ergebnis im zulässigen Gleitkommaintervall liegen
0	falls – *x* Null ist und *y* nicht positiv oder – *x* negativ ist und *y* nicht ganzzahlig
+HUGE	falls das Ergebnis Überlauf hervorruft
–HUGE	falls das Ergebnis Unterlauf hervorruft

Fehlermeldung

Falls pow fehlerhaft endet, wird in errno ein Fehlercode abgesetzt:

EDOM : Argument zu groß
ERANGE : Resultat zu groß

Hinweis

Wenn Sie in Ihrem Programm pow verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Berechne x**y für eingelesene Argumente *x* und *y*:

```
#include <math.h>


main()
{
  double x,y;
  scanf("%F %F",&x,&y);
  printf("%g**%g : %g\n",x,y,pow(x,y));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > exp, log, log10, sqrt

Formatierte Ausgabe auf Standardausgabe

#include <stdio.h>

int printf(format [,arg]...**)**
char *format;

printf gibt die Argumente gemäß den Formatanweisungen in *format* auf Standardausgabe (Dateizeiger stdout) aus.

Typ

C-Funktion (s)

Parameter

char *format

Die Formatzeichenreihe enthält drei Klassen von Zeichen:

a) Formatsteuerzeichen

- Neue Zeile ('\n')
- Rücksetzen ('\b')
- Tabulator ('\t')
- Seitenwechsel ('\f')
- Wagenrücklauf ('\r')

b) Zeichen, die 1:1 ohne Umwandlung ausgegeben werden

c) Formatangaben für die Ausgabe der Argumente von der Form:

```
                              ⎧ [l]{d|o|x|u}   ⎫
  ⎡+⎤      ⎡n⎤  ⎡.m⎤          ⎪                ⎪
% ⎢ ⎥ [0]  ⎢ ⎥  ⎢  ⎥          ⎨   {D|O|X|U}    ⎬
  ⎣-⎦      ⎣*⎦  ⎣.*⎦          ⎪                ⎪
                              ⎩ {c|e|f|g|s|%}  ⎭
```

dabei bedeutet:

+	das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben
–	linksbündig ausrichten
0	mit Nullen auffüllen Standard: Leerzeichen werden links angefügt
n	minimale Gesamtfeldbreite (inklusive Dezimalpunkt), falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung
*	die Gesamtfeldbreite ist variabel; der aktuelle Wert (Typ: unsigned) steht vor dem dazugehörenden Argument in der Argumentenliste
.m	*m* gibt die Stellen nach dem Komma für eine e- oder f-Konvertierung an oder die maximale Anzahl von Zeichen, die von einer Zeichenreihe ausgegeben werden sollen.
.*	Möglichkeit, die Stellen nach dem Komma oder die Länge einer Zeichenreihe variabel anzugeben; der aktuelle Wert (Typ: unsigned) steht vor dem dazugehörenden Argument in der Argumentenliste.
l vor d, o, u, x	Format für den Typ long int;

Folgende Parameter legen die eigentliche Umwandlung fest:

c	einzelnes Zeichen (char); das Zeichen '\0' wird ignoriert
d	Dezimaldarstellung einer ganzen Zahl (int)
D	Dezimaldarstellung einer ganzen Zahl (long int)
e	Gleitpunktzahl (float oder double) im Format [-]d.ddd**e**{+ |-}dd Die Anzahl der Stellen nach dem Komma hängt von der Genauigkeitsangabe *.m* ab. Standard (keine Angabe): 5 Stellen
f	Gleitkommazahl (float oder double) im Format [-]ddd.ddd Die Anzahl der Stellen nach dem Komma hängt von der Genauigkeitsangabe in *.m* ab: Standard (keine Angabe) : 6 Stellen 0 : ganzahlige Ausgabe ohne Dezimalpunkt
g	Gleitkommazahl (float oder double) im d, f oder e Format, je nachdem, welche Darstellung unter Wahrung der Genauigkeit am wenigsten Platz beansprucht
o	Oktale Darstellung einer ganzen Zahl (int)
O	Oktale Darstellung einer ganzen Zahl (long int)
s	Format für Zeichenreihen Die Zeichenreihe sollte mit '\0' abgeschlossen sein. printf schreibt so viele Zeichen der Zeichenreihe, wie in *.m* angegeben ist: Standard (keine Angabe) : printf schreibt alle oder 0 : Zeichen bis '\0'
u	Vorzeichenlose Dezimalzahl (unsigned). Format zur Ausgabe von Zeigerwerten
x	hexadezimale Darstellung einer ganzen Zahl (int)
X	hexadezimale Darstellung einer ganzen Zahl (long int)
%	Ausdruck von %, keine Umwandlung

Ergebnis

0	bei Erfolg
negativ	Bei Fehler

Beispiel

Die gängigsten printf-Formate erklären sich durch ihre Anwendung in den übrigen Beispielprogrammen von selbst. Nachstehend finden sie einige weitere Formatangaben einschließlich ihrer Wirkung aufgelistet.
Zur Verdeutlichung ist das umgewandelte Ergebnis in > < eingerahmt.

Formatangabe	Argument(e)	Ergebnis
%.6s	"Konstanz"	>Konsta<
%10.5s	"Konstanz"	> Konst<
%-10.5s	"Konstanz"	>Konst <
%15.15s	"Konstanz"	> Konstanz<
%*.*s	20,7,"Konstanz"	> Konstan<
%-*.*s	15,10,"Konstanz"	>Konstanz <
%8d	721932	> 721932<
%-8d	721932	>721932 <
%+d		>+d<
%-*.*f	3,2,27.31928	>27.32<
%-0*.*f	1,12,19.84	>19.840000000000<
%04.*f	12,10.60	>10.600000000000<
%-0*.*g	1,12,19.84	>19.84<
%e	1712.1961	>1.71220e+03<
%.10e	1712.1961	>1.71220e+03<
%10.10e	1712.1961	>1.7121961000e+03<

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fprintf, sprintf, putc, putchar, puts, scanf, fwrite

Histogramm der Prozeßausführung

void profil(puff,p _ größe,dist,skal)
char *puff;
int p _ größe, dist;
unsigned skal;

profil wird normalerweise nur in der Entwicklungsphase eines Anwenderprogramms benutzt, um das Verhalten einzelner Funktionen zu testen. Nach einem profil Aufruf wird der Befehlszähler bei jedem Takt der internen Uhr ({HZ} _ tel Sekunde) überprüft. Die Zahl *dist* wird dann vom Befehlszähler subtrahiert und das Ergebnis mit *skal* multipliziert. Entspricht die Ergebniszahl einem Index in *puff*, so wird der Eintrag an dieser Stelle um eins hochgezählt.

Typ

Systemaufruf

Parameter

← char *puff
: Zeiger auf einen Speicherbereich, in dem die gezählten Werte gespeichert werden

int p _ größe
: Länge (in Bytes) des Speicherbereichs, auf dessen Anfang *puff* zeigt

0
: macht den profil Aufruf wirkungslos

int dist
: ganze Zahl, die vom aktuellen Wert des Befehlszählers abgezogen werden soll

unsigned skal
: Skalierungsfaktor

0 oder 1
: Auswertung abstellen

Hinweis

- Bei einem exec Aufruf, wird profil abgestellt.
- Bei einem fork Aufruf läuft profil im Vater- und Sohnprozeß weiter.

> > > > monitor, prof(Kommando)

Prozeßüberwachung

#include <signal.h>

int ptrace(anfrage,pnr,adr,daten**)**
int anfrage, pnr;
int *adr;
int daten;

ptrace befähigt einen Vaterprozeß die Ausführung eines Sohnprozesses zu steuern (tracing). Der Hauptanwendungsbereich dieser Funktion ist die Fehlersuche in Programmen mit Hilfe von Haltepunkten (breakpoint debugging). Ein getraceter Sohnprozeß läuft normal ab, bis er durch ein Signal unterbrochen wird. Dieses wird entweder extern eingegeben oder intern z.B. durch einen Fehler (illegal instruction) hervorgerufen.
Der Vaterprozeß erfährt durch die Funktion wait den Stopzustand des Sohnprozesses (siehe wait, *status* = 0177).
Der Vaterprozeß bearbeitet mit der Funktion ptrace den Speicherabzug des Sohnes, allerdings nur, wenn der Sohn gestoppt hat!!!
Er kann auch mit ptrace den Sohnprozeß unterbrechen, weiterlaufen lassen, oder beenden.
Um Fehlern zuvorzukommen, funktioniert ptrace nicht im Falle eventueller exec Aufrufe beim Sohnprozeß mit gesetztem s-Bit. Solch ein getraceter Prozeß stoppt mit dem Signal SIGTRAP.

Typ

Systemaufruf

Parameter

int anfrage

Der Parameter *anfrage* bestimmt die Tätigkeit, die die Funktion ptrace ausführen soll und hat einen von folgenden Werten.

0
: Diese *anfrage* muß der Sohnprozeß benutzen, um anzugeben, daß er bereit ist, von seinem Vaterprozeß getracet zu werden (trace flag gesetzt).
pnr, *adr* und *daten* werden ignoriert. Es wird kein Wert zurückgegeben. Eigenartige Ergebnisse treten auf, wenn der Vaterprozeß den Sohnprozeß nicht tracet.
Dies ist die einzige *anfrage*, die der Sohnprozeß benutzen kann. Alle anderen werden nur vom Vaterprozeß benutzt.

1,2
: Hier wird der Inhalt an der Stelle *adr* im Adressraum des Sohnrozesses gelesen und an den Vaterprozeß übergeben. Wenn der Daten- und der Textbereich getrennt angelegt sind, spricht 1 den Textbereich und 2 den Datenbereich an. Andernfalls ergibt 1 oder 2 das gleiche Ergebnis. *daten* wird ignoriert. *adr* muß gerade sein, sonst wird -1 an den Vaterprozeß übergeben und errno vom Vater wird auf EIO: Ein/Ausgabe-Fehler gesetzt.

3
: Hier wird der Inhalt an der Stelle *adr* im Prozeßverwaltungsbereich des Betriebssystems, wo die Daten über den Sohnprozeß stehen, (= Prozeßspezifische Register und andere Informationen über den Prozeß) gelesen und an den Vaterprozeß übergeben.
adr muß gerade und kleiner als 512 sein. Fehlerbehandlung wie bei 1,2.

4,5
: Umgekehrt wie in 1,2 wird hier auf den Inhalt an der Stelle *adr* geschrieben. *adr* muß gerade sein. Der geschriebene Wert (*daten*) wird an den Vater zurückgegeben.
Bei getrennten Text- und Datenbereichen steht 4 für Text- und 5 für Datenbereich, sonst besteht kein Unterschied. Diese beiden Werte von *anfrage* funktionieren nicht, wenn *adr* in einem reinen Textbereich liegt und ein anderer Prozeß diesen Raum benutzt.
Sonst Fehler wie bei 1,2.

6	Schreibt auf den in 3 erwähnten Bereich. *daten* ist der zu schreibende Wert. *adr* gibt die zu beschreibende Stelle an. Es können nur einige wenige Stellen beschrieben werden.
7	Der Sohnprozeß wird hier nach einer Unterbrechung wieder gestartet. *daten* wird als Signal betrachtet und der Sohnprozeß beginnt an der Stelle *adr*. Normalerweise ist *daten* 0, um anzugeben, daß das Signal,das den Sohn stoppte, nun ignoriert wird. Wenn *daten* eine gültige Signalnummer ist, macht der Sohnprozeß weiter, als ob er dieses Signal erhalten hätte und alle anderen Signale werden ignoriert (insbesondere die, die den Sohnemann stoppten). Wenn *adr* Zeiger auf 1 ist (adr = (int*)1;), beginnt der Sohn an der Stelle, an der er unterbrochen wurde. Bei erfolgreicher Ausführung dieser *anfrage* wird *daten* an den Vaterprozeß übergeben. Bei Fehler (*daten* ungleich 0 oder ungültige Signalnummer) wird -1 an den Vaterprozeß übergeben und errno auf EIO gesetzt.
8	Der getracete Prozeß (Sohn) wird wie bei exit beendet.
9	Diese *anfrage* ist maschinenabhängig. Sie funktioniert wie *anfrage* 7, außer daß sie nach jedem einzelnen Programmschritt den Sohn wieder stoppt (single step tracing). Das Stopsignal ist dann SIGTRAP. Diese *anfrage* wird benutzt, wenn man bei der Fehlersuche (debugging) Haltepunkte (breakpoints) verwendet.
int pnr	Prozeßnummer des getraceten Prozesses, der ein direkter Sohn vom tracenden Prozeß sein muß.
int adr	*adr* ist eine Adresse aus dem Adreßraum des Sohnprozesses. Sie muß eine gerade ganze Zahl sein. Adreßraum kann sein: Datensegment, Textsegment oder Prozeßverwaltungsbereich.
int daten	*daten* ist der Inhalt einer Adresse. *daten* kann mit ptrace aus einer Adresse gelesen oder in eine Adresse geschrieben werden. Ausnahme siehe *anfrage* 7: *daten* ist hier eine Signalnummer.

Ergebnis

Bei Erfolg sind die Rückgabewerte der ptrace Aufrufe in den einzelnen *anfragen* angegeben.

–1 es liegt ein Fehler vor, wenn einer der folgenden Punkte erfüllt ist:

- *anfrage* hat ungültigen Wert.
- *pnr* ist kein tracebarer Prozeß (entweder existiert der Prozeß nicht oder er hat keinen ptrace Aufruf mit *anfrage* 0 durchgeführt).
- Die Adresse *adr* liegt außerhalb ihres Gültigkeitsbereichs.
- *daten* bezeichnet eine ungültige Signalnummer.

Hinweis

ptrace sollte nicht vom Anwender benutzt werden.
ptrace wird nur in Programmen zur Softwarefehlersuche verwandt und ist hardwareabhängig.
Die Fehleranzeige -1 ist ein legaler Funktionsausgabewert.
Zu bemerken ist noch, daß die im folgenden Beispielprogramm benutzte Funktion wait mehrfach im Vaterprozeß aufgerufen werden muß, obwohl es nur einen Sohnprozeß gibt. Sie wird deshalb so benutzt, weil der Vaterprozeß jedesmal warten soll, bis der Sohnprozeß durch ein internes Signal gestoppt wird. Der Vaterprozeß soll nicht in seiner ganzen Länge parallel zum Sohnprozeß ablaufen. Diese mehrfachen wait Aufrufe beziehen sich hier also alle auf ein und denselben Sohnprozeß!!

Beispiel

```
#include <stdio.h>
#include <signal.h>

main()
{
  int status, x, pnr, adr, i, inh;
  int const = 0;
  adr = (int*)1;

  printf("Hier ist der Vaterprozess vor der Verzweigung\n");
  pnr = fork();

  if (pnr ==-1 )
     { printf("Sohnprozess nicht erzeugt\n");
       exit(1);
     }

  if (pnr == 0 )
                              /*Hier im Sohnprozess*/
     { ptrace(0);
       printf("Hier im Sohnprozess!!!\n");
       printf("Ergebnis ist: %d\n",256/16);
       x = 1/const;
                              /*Hier wird durch Fehler aus dem Sohn
                                gesprungen */

       printf("Hier zum 2.Mal im Sohnprozess\n");
       x = 1/const;
                              /*Hier wird zum zweiten Mal rausgesprungen*/

       printf("Hier zum 3. Mal im Sohnprozess\n");

       for(i=0; i<5; i++)
       { printf("HALLO  "); }
       printf("Hallo\n");
       x=1/const;             /*Rausspringen durch SIGILL*/

       printf("Sohnschluss\n");
                              /*Dieses Print-Kommando wird nicht mehr
                                ausgefuehrt, da zu diesem Zeitpunkt das
                                Tracing vom Vater bereits beendet ist*/

     }

  else                        /*Hier im Vaterprozess*/
     {
```

```
    while(wait(&status) != pnr)
              ;
                        /*Vater soll warten, bis Sohn stoppt!*/

    printf("Hier im Vater nach 1. Sohnaustritt\n");
    ptrace(7,pnr,adr,0);




    while(wait(&status) != pnr)
              ;
    printf("Hier im Vater nach 2. Sohnaustritt\n");

    for(i=0; i<8; i=i+2)
    {inh =  ptrace(1,pnr,i);
    printf("Inhalt von adr ist: %x\n",inh); }

    ptrace(7,pnr,adr,0);

    while(wait(&status) != pnr)
              ;
    printf("Hier im Vater nach 3. Sohnaustritt\n");
    ptrace(8);
                        /*Beendigung des Tracing*/

    printf("Hier im Vaterprozess nach Tracing\n");
   }
}
```

Dateien

/usr/include/signal.h
: Definition der Signale und Standardsignalbehandlungen.

> > > > adb(Kommando), exec, fork, perror, signal, wait;

Zeichen ausgeben

#include <stdio.h>

int putc(c,dz)
char c;
FILE *dz;

putc schreibt das Zeichen *c* in die Datei mit Dateizeiger *dz* an die aktuelle Lese/Schreibposition.

Typ

Makro (s)

Parameter

char c	Zeichen, das geschrieben werden soll
← FILE *dz	Dateizeiger für die Ausgabedatei

Ergebnis

c, das geschriebene Zeichen	bei Erfolg
EOF	sonst

Hinweis

Im Unterschied zu fputc ist putc ein Makro und keine Funktion und kann daher nicht einer anderen Funktion als Argument übergeben werden.

Beispiel

Text von der Standardeingabe zeichenweise in *datei* schreiben:

```
#include <stdio.h>

FILE *fp;
char c;

main()
{
  fp = fopen("datei","w");
  while((c=getchar()) != EOF)
        putc(c,fp);
  fclose(fp);
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fputc, printf, putchar, putw

Zeichen auf Standardausgabe schreiben

#include <stdio.h>

int putchar(c)
char c;

putchar schreibt das Zeichen *c* auf Standardausgabe und ist definiert als:

putc(c,stdout)

Typ

Makro (s)

Parameter

char c	Zeichen, das ausgegeben werden soll

Ergebnis

c, das geschriebene Zeichen	bei Erfolg
EOF	sonst

Beispiel

siehe Beispiel bei putc

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > putc, fputc, putw, printf, fopen, setbuf

Zeichenreihe auf Standardausgabe ausgeben

#include <stdio.h>

int puts(s)
char *s;

puts schreibt die Zeichenreihe *s* auf Standardausgabe und schließt die Ausgabe mit Neue Zeile ab.

Typ

C-Funktion (s)

Parameter

char *s	Zeichenreihe, die ausgegeben werden soll

Ergebnis

0	bei Erfolg
EOF	sonst

Hinweis

Das abschließende Nullbyte von *s* wird nicht mitausgegeben.

Beispiel

puts wird verwendet, um zeilenweise auf Standardausgabe zu schreiben. Dieses Beispiel zeigt den unterschiedlichen Abschluß der Ausgabe bei puts und fputs:

```
#include <stdio.h>

main()
{
 FILE *fp;
 char s[BUFSIZ];
 fp=fopen("datei","w");
 while(gets(s) != NULL)
       {
         fputs(s,fp);
         puts(s);
       }
}
```

Wenn Sie nach Ablauf dieses Programmes *datei* anschauen, stellen Sie fest, daß die Zeilen aus der Eingabe hintereinander und nicht zeilenweise geschrieben wurden.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fputs, sprintf, fopen, setbuf

Wortweise in eine Datei schreiben

#include <stdio.h>

int putw(w,dz**)**
int w;
FILE *dz;

putw schreibt ein Maschinenwort in die Datei mit Dateizeiger *dz* an die aktuelle Lese/Schreibposition.

Typ

C-Funktion (s)

Parameter

int w	Wort, das geschrieben werden soll.
← FILE *dz	Dateizeiger für Ausgabedatei

Ergebnis

w, das geschriebene Wort	bei Erfolg
EOF	sonst

Achtung

Weil Wortlänge und Anordnung der Bytes maschinenabhängig sind, können unter Umständen Dateien, die mit putw auf einem Prozessor beschrieben wurden, nicht mit getw auf einem anderen Prozessor gelesen werden.

Hinweis

Da putw Fehler nicht explizit anzeigt (-1 ist ein gültiger Integer Wert), sollten Sie zusätzlich ferror verwenden, um abzuprüfen, ob vor oder nach dem Schreiben ein Fehler auftrat.

Beispiel

Datei *ein* wortweise auf Datei *aus* übertragen:

```
#include <stdio.h>

FILE *dz, *dz1;
int w;

main()
{
  dz = fopen("ein","r");
  dz1 = fopen("aus","w");
  while(!feof(dz) && !ferror(dz) && !ferror(dz1))
       {
         w = getw(dz);
         putw(w,dz1);
       }
  fclose(dz);
  fclose(dz1);
}
```

Dateien

/usr/include/stdio.h
Definitionen für die Standardein/ausgabe

> > > > putc, puts

Quicksort

void qsort(feld,n,anz,vergl**)**
char *feld;
int n, anz;
int (*vergl**)()**;

qsort sortiert die *n* Elemente von *feld* nach dem Quicksort-Verfahren. Jedes Element des Feldes beansprucht *anz* Bytes.
Um das Feld sortieren zu können, benötigt qsort eine vom Benutzer bereitgestellte Funktion *vergl*, die zwei Elemente miteinander vergleicht.

Typ

C-Funktion

Parameter

← char *feld
Zeiger auf das erste Element des zu sortierenden Feldes

int n — Anzahl der Elemente

int anz — Größe eines Elementes in Bytes

int (*vergl)()
Zeiger auf eine Funktion, die zwei Elemente vergleicht. Die Funktion muß eine ganze Zahl als Ergebnis liefern, die wie folgt gedeutet wird:

< 0	Argument1 ist kleiner als Argument2
= 0	Argument1 und Argument2 sind gleich
> 0	Argument1 ist größer als Argument2

Die Funktion hat zwei Parameter, zwei Zeiger auf den Typ der Vektorelemente.
Die Funktion kann etwa wie folgt definiert sein:

```
int vergl(a,b)          /*vergleicht  zwei int Werte */
char *a, *b;
{
   if( *((int *)a) < *((int *)b) )
      return(-1);
   else if( *((int *)a) > *((int *)b) )
         return(1);
   return(0);
}
```

Beispiel

Sortieren eines Zahlenfeldes und Ausgabe auf Standardausgabe:

```
int vergl(a,b)
char *a, *b;
{
   if( *((int *)a) < *((int *)b) )
     return -1;
   else if( *((int *)a) > *((int *)b) )
          return 1;
   return 0;
}

main()
{
  int i,j;
  static int feld[]={7,4,2,1,54,9,2,3,1,23};

  qsort( (char *)feld,10,2,&vergl);
  for(j=0; j<10; j++)
     printf("%d ",feld[j]);
}
```

> > > > sort(Kommando)

Zufallszahlgenerator

int rand()

rand liefert eine positive, ganze Zufallszahl aus dem Bereich [0, 2^{15}-1]. Ein rand Aufruf wählt Werte aus einer Folge von Pseudo-Zufallszahlen aus unter Verwendung eines multiplikativen, kongruenten Zufallszahlgenerators. Der Generator hat eine Periode von 2^{32}.

Typ

C-Funktion

Parameter

keine

Ergebnis

Zufallszahl aus [0, 2^{15}-1]

Hinweis

Man kann den Zufallszahlgenerator mit srand initialisieren.

Beispiel

Generiere zweimal diegleichen fünf Zufallszahlen:

```
int i;

main()
{
   for(i=1;i<=10;++i)
      {
        printf("%d\n",rand());
        if(i==5)
          srand(1);
      }
}
```

>>>> srand

Auf Eingabedaten abprüfen

int rdchk(dk**)**
int dk;

rdchk prüft, ob ein Prozeß aus der Datei mit Dateikennzahl *dk* lesen kann.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl der Datei, die auf Eingabedaten überprüft werden soll.

Ergebnis

1	ein korrekter Lesezugriff kann ausgeführt werden: es sind Eingabedaten vorhanden oder das nächste Zeichen ist Dateiende.
0	der Prozeß würde beim Einlesen blockieren, weil keine Daten vorhanden sind
–1	Fehler, wenn

– der Prozeß auf die Datei mit Dateikennzahl *dk* nicht zugreifen kann, weil sie nicht vorhanden ist oder
– der Prozeß keine Zugriffsberechtigung hat

Fehlermeldung

Liefert rdchk das Ergebnis -1, steht in errno ein entsprechender Fehlercode:

EBADF : Unzulässige Dateikennzahl
EACCES : Zugriff untersagt

Hinweis

Es kann sein, daß der rdchk Aufruf bei diversen zeichenorientierten Gerätedateien abgewiesen wird und einen Fehlercode wie etwa:

ENODEV : Gerät unbekannt
EINVAL : Unzulässiges Argument

in errno ablegt.

Beispiel

Die richtige Verbindung von rdchk und read sehen Sie an folgendem Programm:

```
#define STANDARDAUS 1      /* Dateikennzahl für die Standardausgabe */

int dk;
char c;

main()
{
  dk = open("datei","0");
  if(rdchk(dk) > 0)
    {
      while(read(dk,&c,1) > 0)
           write(STANDARDAUS,&c,1);
    }
  else
      printf("sorry\n");
  close(dk);
}
```

>>>> read

Elementare Einleseoperation

int read(dk,puf,anz**)**
int dk;
char *puf;
int anz;

read ist die elementare Einleseoperation.
read liest aus der Datei mit Dateikennzahl *dk* maximal *anz* Zeichen, in den Bereich, auf den *puf* zeigt.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl für die Eingabedatei
char *puf	Zeiger auf den Bereich, in den die gelesenen Daten geschrieben werden sollen. Der Bereich sollte mindestens *anz* Bytes groß sein.
int anz	Anzahl der Bytes, die gelesen werden sollen. Es ist nicht sichergestellt, daß read tatsächlich *anz* Bytes liest.

Ergebnis

Anzahl der tatsächlich gelesenen Bytes	bei Erfolg
0	Dateiende
–1	read hat nichts gelesen, weil ein Fehler vorliegt:

- physikalischer Ein/Ausgabefehler oder
- der Puffer ist nicht korrekt angegeben oder
- die Datei kann nicht gelesen werden, weil sie nicht vorhanden ist oder keine Zugriffsberechtigung besteht oder

- *anz* ist unmöglich oder
- während dem read Aufruf wurde ein Signal abgefangen

Fehlermeldung

Bei Ergebnis -1 wird in errno ein entsprechender Fehlercode abgelegt:

EIO :	Ein/Ausgabefehler
EFAULT :	Unzulässige Adresse
EBADF :	Unzulässige Dateinummer
EINVAL :	Unzulässiges Argument
EINTR :	Systemaufruf wurde unterbrochen

Hinweis

- Die Anzahl der tatsächlich gelesenen Bytes kann kleiner sein als die Angabe in *anz*, weil Dateiende erreicht ist oder von einem Bildschirm eingelesen wird, wo im Normalfall nur bis zum nächsten Neue-Zeile-Zeichen gelesen wird.
- Die zwei häufigsten Angaben für *anz* sind:

1 :	für ungepuffertes zeichenweises Einlesen
BUFSIZ :	für gepuffertes Einlesen; BUFSIZ entspricht der physikalischen Blockgröße der meisten blockorientierten Geräte.

- Um sicherzugehen, daß nicht mehr Bytes gelesen werden, als der Puffer aufnehmen kann, können Sie sizeof verwenden.
- Das Bit O_NDELAY im System-Dateistatus-Byte (siehe fcntl, open) wirkt sich beim Lesen von einer Pipe oder einem zeichenorientierten Gerät wie folgt aus:

  Wenn O_NDELAY nicht gesetzt ist, blockiert der read Aufruf, bis Daten vorhanden sind.
  Ist O_NDELAY gesetzt, kehrt read sofort mit -1 zurück, falls keine Daten vorhanden sind.

Beispiel

Programm, das die Standardeingabe auf Standardausgabe kopiert. Wenn Sie den Umlenkmechanismus ausnützen, können Sie damit von einer beliebigen Quelle auf ein beliebiges Ziel kopieren:

```
#include <stdio.h>

main()
{
  char buf[BUFSIZ];
  int n;

  while((n = read(0, buf, sizeof(buf))) > 0)
        write(1, buf, n);
}
```

Dateien

/usr/include/stdio.h
Definition von BUFSIZ

> > > > open, creat, pipe, dup

Speicherplatz verändern

char *realloc(sp,anz)
char *sp;
unsigned anz;

realloc setzt die Größe des Speicherplatzes, der zuvor von malloc oder calloc zugeteilt wurde, auf *anz* Bytes.

Typ

C-Funktion

Parameter

char *sp	Zeiger, der von malloc oder calloc geliefert wurde; zeigt auf den Anfang des zugewiesenen Speicherplatzes.
unsigned anz	Größe, die neu gesetzt werden soll

Ergebnis

Zeiger auf den Anfang des geänderten Speibereiches
Wenn realloc die Größe ändert, kann es vorkommen, daß der zugewiesene Block verschoben wird. Der Inhalt des Bereiches bleibt bis zum Minimum der alten und neuen Größe erhalten.

Nullzeiger: realloc hat nichts verändert, weil

- nicht genügend Platz vorhanden ist oder
- ein Fehler auftrat

Achtung

Wenn realloc den Nullzeiger liefert, kann eventuell der Block, auf den *sp* zeigt, zerstört worden sein!

Hinweis

realloc kann auch verwendet werden, wenn seit dem letzten malloc oder calloc mit free bereits Platz freigegeben wurde. In diesem Sinn kann realloc eingesetzt werden, um Speicher zusammenzuschieben.

> > > > malloc, calloc, free

Auf Dateianfang positionieren

#include <stdio.h>

void rewind(dz**)**
FILE *dz;

rewind positioniert den Lese/Schreibzeiger der Datei mit Dateizeiger *dz* auf Dateianfang.

Typ

C-Funktion (s)

Parameter

File *dz	Dateizeiger der Datei

Hinweis

Die Aufrufe rewind(dz) und fseek(dz,0L,0) sind äquivalent, außer daß rewind kein Ergebnis hat.

Beispiel

Zuerst den Rest der Datei, dann den Dateianfang ausgeben:

```
#include <stdio.h>

main()
{
   FILE *dz;
   int c,i;

   dz = fopen("datei","r");
            /* die ersten 10 Zeichen überlesen */
   fseek(dz,10L,0);
   while((c=getc(dz)) != EOF)
         putc(c,stdout);
            /* auf Dateianfang positionieren */
   rewind(dz);
   for(i=0; i<10; i++)
      {
       c=getc(dz);
       putc(c,stdout);
      }
   fclose(dz);
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fseek

Letztes Auftreten eines Zeichens in einer Zeichenreihe

char *rindex(s,c)
char *s,c;

rindex sucht das letzte Vorkommen des Zeichens *c* in der Zeichenreihe *s* und liefert bei Erfolg einen Zeiger auf die Position in *s*.

Typ

C-Funktion

Parameter

char *s	Zeichenreihe, in der das Zeichen gesucht werden soll
char c	Zeichen, das gesucht werden soll

Ergebnis

Zeiger auf die	Position des letzten Vorkommens von c falls *c* in *s* vorkommt
Nullzeiger	sonst

Beispiel

```
main()
{
 char *s = "was für ein Spaß im kkühlen Naß!";
  printf("%s\n",s);
  printf("wo steckt der Fehler? %s\n",rindex(s,'k'));
}
```

> > > > index, strcat, strcmp, strcpy, strlen, strdup,

Größe des Datensegmentes verändern

char *sbrk(inkr**)**
int inkr;

sbrk verändert die Größe des vom Benutzer zugänglichen Datensegmentes (siehe auch brk).

Typ

Systemaufruf

Parameter

int inkr — Anzahl der Bytes:

negative ganze Zahl
: Herabsetzen des "break" um *inkr* Bytes, also Verkleinerung des Datensegmentes

positive ganze Zahl
: Hochsetzen des "break" um *inkr* Bytes, also Vergrößerung des Datensegmentes

0
: der Aufruf sbrk(0) liefert die aktuelle Adresse des "break".

Ergebnis

Zeiger auf den alten "break" Wert
: sbrk konnte das Datensegment entsprechend *inkr* verändern

-1
: sbrk hat das Datensegment nicht verändert, weil mehr Speicher angefordert wird, als das System zuläßt (siehe auch ulimit).

Fehlermeldung

Bei Ergebnis -1 steht in errno der Fehlercode:

ENOMEM : Arbeitsspeicher unzureichend

Hinweis

Wenn Sie mittels sbrk mehr Speicher anfordern, dann ist der Inhalt des neu zugewiesenen Speicherbereiches anfangs undefiniert.

Externe Größen

extern end — erste Adresse oberhalb des Datensegmentes (Erklärung siehe brk).

> > > > brk, exec, malloc, calloc, ulimit

Formatiertes Einlesen von Standardeingabe

#include <stdio.h>

int scanf(format [,arg_zg]...**)**
char *format;
<typ> *arg_zg, ...;

scanf liest von der Standardeingabe. scanf wandelt ein Eingabefeld gemäß seiner zugehörigen Formatanweisung um und speichert das Ergebnis an die Stelle, die der entsprechende Ergebniszeiger angibt.

Typ

C-Funktion (s)

Parameter

char *format

Die Formatzeichenreihe kann drei Klassen von Zeichen enthalten:

a) Zwischenraum

- Tabulator
- Leerzeichen
- Zeilenwechsel

Zwischenraumzeichen in der Eingabe werden außer bei %c ignoriert.

b) beliebiges Zeichen außer '%'

das Zeichen muß mit dem nächsten Zeichen aus der Eingabe übereinstimmen, ansonsten wird die Eingabebearbeitung abgebrochen.

c) Formatangaben für die Umwandlung der eingelesenen Zeichen. Eine Formatangabe ist von der Form:

```
          ⎧ [l]{d|e|f|o|x}        ⎫
          ⎪                       ⎪
          ⎪    {D|E|F|O|X}        ⎪
% [*][n]  ⎨                       ⎬
          ⎪    {c|s|%}            ⎪
          ⎪                       ⎪
          ⎩    {[...]|[^...]}     ⎭
```

Zu einer Formatangabe gehört ein Eingabefeld, d.h. eine Zeichenreihe ohne Leerzeichen. Führende Tabulator- und Leerzeichen werden übersprungen (beachte jedoch Format %c!). Die Länge der Zeichenreihe ist entweder explizit durch die Feldbreite *n* festgelegt oder ergibt sich implizit, sobald das erste Zeichen gelesen wird, das nicht zu der Formatangabe paßt.
Das erste '%' kennzeichnet die Formatangabe, die restlichen Zeichen werden wie folgt interpretiert:

* überspringe eine Zuweisung:
das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber nicht abgespeichert.

n maximale Länge der umzuwandelnden Eingabezeichenreihe; tritt vorher ein Zeichen auf, das nicht zur Formatangabe paßt, wird die Länge entsprechend gekürzt.

l Angabe für doppelte Länge:

vor d, o, x
Umwandlung einer Dezimal-, Oktal- oder Hexadezimalzahl in long int
← long *arg_zg

vor e, f
Gleitkommazahl
Umwandlung in double
← double *arg_zg

Folgende Formatelemente legen die eigentliche Umwandlung fest:

Formatelement/ Ergebniszeiger	erwartete Eingabe
c ← char *arg_zg	einzelnes (abdruckbares) Zeichen; Achtung ein einzelnes Zwischenraumzeichen wird auch gelesen und nicht wie sonst übersprungen!
nc ← char arg_zg[]	Feld von *n* (abdruckbaren) Zeichen Achtung führende Tabulator- und Leerzeichen werden übersprungen, aber Zwischenraumzeichen innerhalb des Eingabefeldes nicht (sind keine Trenner mehr)!
d ← int *arg_zg	ganze Dezimalzahl (mit oder ohne führende Null)
D ← long *arg_zg	ganze Dezimalzahl (mit oder ohne führende Null) Umwandlung wie bei ld
e ← float *arg_zg	Gleitkommazahl im Format: [+|-]dd[.dd{e|E}[+|-]dd]
E ← double *arg_zg	Gleitkommazahl wie bei e Umwandlung wie bei le
f ← float *arg_zg	Gleitkommazahl im Format: dd[.ddd]
F ← double *arg_zg	Gleitkommazahl wie bei f Umwandlung wie bei lf
o ← int *arg_zg	ganze Oktalzahl (mit oder ohne führende Null)

Formatelement/ Ergebniszeiger	erwartete Eingabe
o	ganze Oktalzahl (mit oder ohne führende Null) Umwandlung wie bei lo
← long *arg_zg	
s	Zeichenreihe; Das zugehörige Eingabefeld wird durch ein Zwischenraumzeichen abgeschlossen, falls eine Feldbreite *n* angegeben ist, werden *n* Zeichen oder bis zum nächsten Zwischenraum gelesen, je nachdem was vorher kommt (Zwischenraumzeichen sind hier also Trenner, vgl %nc!) Die eingelesene Zeichenreihe wird mit '\0' abgeschlossen. Führende Tabulator- und Leerzeichen werden übersprungen, also
1s	liest das erste Nicht-Leerzeichen (vgl %c!).
← char arg_zg[]	
x	ganze Hexadezimalzahl (mit oder ohne führende Null) (aber nicht 0xddd!)
← int *arg_zg	
X	ganze Hexadezimalzahl (mit oder ohne führende Null) Umwandlung wie bei lx
← long *arg_zg	
[...]	Zeichenreihe, die nur aus Zeichen besteht, die in [...] vorkommen, z.B: [a] : Zeichenreihe, die nur aus 'a' besteht. Es wird eingelesen bis zum ersten Zeichen, das nicht in [...] vorkommt.
← char arg_zg[]	
[^...]	Zeichenreihe, die nur aus Zeichen besteht, die nicht in [^...] vorkommen, z.B: [^a]: Zeichenreihe, in der kein 'a' vorkommt. Es wird eingelesen bis das erste Zeichen auftritt, das in [^...] vorkommt.
← char arg_zg[]	
%	das Zeichen '%' selbst.

Ergebnis

Anzahl der eingelesenen und erfolgreich umgewandelten Datenelemente
: Mit dem Ergebnis eines erfolgreichen scanf Aufrufes können Sie feststellen, wieviele Datenelemente tatsächlich eingelesen wurden.

EOF
: Dateiende

0
: Konvertierungsfehler

Achtung

Wenn als letzte Angabe in der Formatzeichenreihe ein Neue-Zeile-Zeichen steht: scanf("...\n",...), müssen Sie die Eingabe am Bildschirm mit der Taste [END] abschließen, weil die Taste [↵] ignoriert wird!

Hinweis

- Das Ergebnis eines scanf Aufrufes sollten Sie immer abfragen, um sicher zu sein, daß kein Fehler passiert ist!
- Der nächste scanf Aufruf startet mit dem Lesen unmittelbar nach dem Zeichen, das als letztes vom vorherigen Aufruf verarbeitet wurde.
- Wenn ein Eingabezeichen nicht dem angegebenen Format entspricht, wird es in den Eingabepuffer zurückgeschrieben. Es muß dort mit getc abgeholt werden, sonst erhält der nächste scanf Aufruf das gleiche Zeichen wieder.
- scanf läßt Neue Zeile ('\n') in der Eingabe stehen. Wenn Sie also nach einem scanf Aufruf z.B. mit gets die nächste Zeile lesen wollen, müssen Sie zunächst z.B. mittels getchar das Zeichen Neue Zeile "weglesen".
- Wenn Sie Ihre Daten für scanf am Bildschirm eingeben, müssen Sie die Eingabe mit der Taste [↵] abschließen. Erst dann erhält das Programm die Eingabe!

Beispiel

Der Aufruf:

```
int i;
float x;
char name[20];
scanf("%2d %f %*d %6s",&i,&x,name);
```

```
weist die Eingabedaten

234567 678 Zipfelxy

wie folgt zu:
i    : 23
x    : 4567.0
name : Zipfel\0
```

Die Zuweisung von 678 wird wegen der %*d Angabe nicht ausgeführt. Das Zeichen, das als nächstes eingelesen wird, ist 'x'.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fscanf, sscanf

Ein/Ausgabepuffer zuordnen

#include <stdio.h>

void setbuf(dz,puf**)**
FILE *dz;
char *puf;

setbuf legt einen Ein/Ausgabepuffer für die Datei mit Dateizeiger *dz* an.
Es gibt drei Arten der Pufferung:

- ungepuffert:
  wenn die Ausgabe ungepuffert ist, erscheint sie in der Ausgabedatei oder auf dem Bildschirm, sobald sie geschrieben wurde.
- blockweise Pufferung:
  Wenn die Ausgabe blockweise gepuffert ist, werden die Ausgabezeichen erst in einem Block gesammelt und anschließend der gesamte Block ausgegeben.
- zeilenweise Pufferung
  wenn die Ausgabe zeilenweise gepuffert ist, werden die Zeichen gesammelt, bis ein Neue-Zeile-Zeichen erscheint.

Wenn eine Datei mit einer Funktion aus der Standardein/ausgabe-Bibliothek eröffnet wurde (z.B. mit fopen), beschafft das System automatisch bei der ersten Lese/Schreiboperation (getc, putc, ...) mit malloc einen Speicherbereich, in dem die Daten für diese Datei gepuffert werden.
Mit setbuf können Sie den Puffer explizit neu- oder umdefinieren.
Dazu müssen Sie setbuf vor der ersten Lese/Schreiboperation aufrufen.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger auf die geöffnete Datei, der explizit ein Ein/Ausgabepuffer zugeordnet werden soll.
char *puf	Zeiger auf den Bereich, der als Puffer verwendet werden soll.
Nullzeiger	Wenn Sie hier den konstanten Zeiger NULL angeben, wird der Puffer auf 0 gesetzt und Ein/Ausgabe folglich nicht gepuffert.

Hinweis

– Die in <stdio.h> definierte Größe BUFSIZ gibt die Standardgröße für den Puffer an.

– Bezüglich Pufferung gelten folgende Voreinstellungen:

Bildschirm :	Pufferung zeilenweise
stderr :	ungepuffert
sonst :	volle Pufferung

– Sie können die Funktion freopen verwenden, um den Puffer für eine Datei von ungepuffert oder zeilenweise Pufferung auf blockweise Pufferung zu setzen.
  Sie können die Pufferung wieder ausschalten, indem Sie freopen und anschließend setbuf(dz,NULL) aufrufen.

Beispiel

Die Pipe wird auf ungepuffert gesetzt. Damit erscheint nach Drücken der return-Taste sofort die gemäß dem tr Kommando veränderte Eingabe am Bildschirm:

```
#include <stdio.h>

main()
{
   FILE *out,*popen();
   int c;

               /* Pipe zwischen Prozeß und tr Kommando
                  einrichten */
               /* Prozeß schreibt auf die Standardeingabe
                  des Kommandos */
   out = popen("tr \"a-z\" \"A-Z\"","w");

               /* Ausgabe auf die Pipe ungepuffert */
   setbuf(out,NULL);

   while((c = getchar()) != EOF)
         putc(c,out);
   pclose(out);
}
```

Wenn Sie in obigem Programm den setbuf Aufruf weglassen, wird die Ausgabe auf *out* gepuffert, d.h. erst wenn der Puffer vollgeschrieben ist, erfolgt eine Ausgabe auf dem Bildschirm.

Dateien

/usr/include/stdio.h
: Definitonen für Standardein/ausgabe

> > > > fopen, getc, putc, malloc, setbuffer, setlinebuf

Ein/Ausgabepuffer zuordnen

#include <stdio.h>

void setbuffer(dz,puf,anz**)**
FILE *dz;
char *puf;
int anz;

setbuffer ist identisch zu setbuf, außer, daß Sie hier die Möglichkeit haben, in *anz* die Größe des Puffers anzugeben.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger auf die geöffnete Datei
char *puf	Zeiger auf den Bereich, der als Puffer verwendet werden soll.
Nullzeiger	Wenn Sie hier den konstanten Zeiger NULL angeben, wird die Puffergröße auf 0 gesetzt und Ein/Ausgabe folglich nicht gepuffert.

Hinweis

- Die in <stdio.h> definierte Größe BUFSIZ gibt die Standardgröße für den Puffer an.
- Bezüglich Pufferung gelten folgende Voreinstellungen:

Bildschirm :	Pufferung zeilenweise
stderr :	ungepuffert
sonst :	volle Pufferung

- Sie können die Funktion freopen verwenden, um den Puffer für eine Datei von ungepuffert oder zeilenweise Pufferung auf blockweise Pufferung zu setzen.

Beispiel

siehe Beispiel bei setbuf.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > fopen, getc, putc, malloc, setbuf, setlinebuf

Gruppennummer des Prozesses setzen

int setgid(gnr)
int gnr;

setgid setzt die reale und effektive Gruppennummer des Prozesses auf *gnr*. Die Gruppennummer *gnr* muß mit der realen Gruppennummer des Prozesses übereinstimmen; eine andere Zuordnung darf nur der Systemverwalter vornehmen.

Typ

Systemaufruf

Parameter

int gnr	neue effektive und reale Gruppennummer des Prozesses

Ergebnis

0	setgid hat die Gruppennummer neu gesetzt
-1	sonst

> > > > getgid, getegid

Auf den Dateianfang von /etc/group positionieren

int setgrent()

setgrent positioniert den Lese/Schreibzeiger der "Gruppendatei" /etc/group auf Dateianfang. Die Funktion wird verwendet, um wiederholtes Durchsuchen mittels getgrent, getgrgid und getgrnam zu ermöglichen.

Typ

C-Funktion

Parameter

keine

Ergebnis

0	Lese/Schreibzeiger von /etc/group zeigt auf Dateianfang

Hinweis

Die Verwendung von setgrent ist sinnvoll in Verbindung mit getgrent, getgrid und getgrnam.

Beispiel

Erst wird mit getgrnam nach dem Gruppennamen *other* gesucht und bei Erfolg der Eintrag ausgegeben. Anschließend werden alle eingetragenen Gruppennamen ausgegeben:

```
#include <stdio.h>
#include <grp.h>

struct group *getgrent();
struct group *getgrnam();
struct group *kennung;

main()
{
   int i;
   if((kennung = getgrnam("other")) != NULL)
      {
        printf(" Name : %s\n",kennung->gr_name);
        printf(" Passwort : %s\n",kennung->gr_passwd);
        printf(" Gruppennummer : %d\n",kennung->gr_gid);
        i=0;
        while(kennung->gr_mem[i] != NULL)
              printf(" Mitglied : %s\n",kennung->gr_mem[i++]);
      }
   setgrent();
   while((kennung = getgrent()) != NULL)
         printf("Gruppenname : %s\n",kennung->gr_name);

}
```

Dateien

/etc/group   Gruppeneinträge

> > > > getgrent, getgrgid, getgrnam, endgrent

"Marke" für nicht-lokale Sprünge setzen

#include <setjmp.h>

int setjmp(zst**)**
jmp_buf zst;

setjmp speichert den aktuellen Programmzustand (Adresse im Laufzeitkeller, Befehlszähler, Registerinhalte) in einem Feld vom Typ jmp_buf.
Ein späterer longjmp Aufruf richtet den von setjmp gespeicherten Programmzustand wieder ein und beginnt die Programmausführung erneut von diesem Zustand aus.

Typ

C-Funktion

Parameter

← jmp_buf zst

Feld, in das setjmp den aktuellen Zustand speichert. Der Typ jmp_buf ist in <setjmp.h> definiert

Ergebnis

0

Beispiel

siehe Beispiel bei longjmp

Dateien

/usr/include/setjmp.h

Typdefinition für jmp_buf

> > > > longjmp, signal

Einstellen des "DES-Algorithmus"

void setkey(schl)
char *schl;

setkey initialisiert den DES (Data Encryption Standard) Algorithmus. Ein späterer encrypt Aufruf benutzt diese Einstellung, um Binärfelder zu ver- oder entschlüsseln.

Typ

C-Funktion

Parameter

char *schl
: Feld, das 64 Zeichen (char) enthält, die nur die Werte 0 und 1 darstellen. Mit diesem Bitfeld wird der DES Algorithmus initialisiert.

Beispiel

siehe Beispiel bei encrypt.

> > > > crypt, encrypt, getpass, passwd(Kommando), login(Kommando)

Ein/Ausgabepuffer für zeilenweise Pufferung zuordnen

```
#include <stdio.h>

void setlinebuf(dz)
FILE *dz;
```

setlinebuf wird dazu verwendet, die Standardausgabe (stdout) und die Standardfehlerausgabe (stderr) zeilenweise zu puffern.

Typ

C-Funktion (s)

Parameter

FILE *dz	Dateizeiger auf die Datei, für die Ein/Ausgabe zeilenweise gepuffert werden soll

Hinweis

- Anders als setbuf und setbuffer können Sie setlinebuf jederzeit aufrufen.
- Ausgabe auf den Bildschirm und auf die Standardfehlerausgabe ist standardmäßig ungepuffert.

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

>>>> fopen, getc, putc, malloc, setbuf, setbuffer

Prozeßgruppe definieren

int setpgrp()

setpgrp definiert eine neue Prozeßgruppe. Die Prozeßnummer des aufrufenden Prozesses wird die neue Prozeßgruppennummer.

Typ

Systemaufruf

Parameter

keine

Ergebnis

neue Prozeßgruppennummer

> > > > getpgrp, exec, fork, kill, signal, exit

Auf den Dateianfang von /etc/passwd positionieren

int setpwent()

setpwent setzt den Lese/Schreibzeiger der "Passwortdatei" /etc/passwd auf Dateianfang. Die Funktion wird verwendet, um wiederholtes Durchsuchen der "Passwortdatei" mittels getpwent, getpwuid und getpwnam zu ermöglichen.

Typ

C-Funktion

Parameter

keine

Ergebnis

0	Lese/Schreibzeiger von /etc/passwd zeigt auf Dateianfang

Hinweis

Die Verwendung von setpwent ist sinnvoll in Verbindung mit getpwent, getpwuid und getpwnam.

Beispiel

Drucke für einen eingelesenen Benutzer die Angaben in /etc/passwd mit entsprechenden Kommentaren und anschließend alle eingetragenen Benutzernamen:

```
#include <stdio.h>
#include <pwd.h>

struct passwd *getpwnam();
struct passwd *getpwent();
struct passwd *daten;
char name[25];

main()
{
   printf("Von wem wollen sie die Information?\n");
   scanf("%s",name);
   daten = getpwnam(name);
   printf("%s\n",daten->pw_name);
   printf("%s\n",daten->pw_passwd);
   printf("%d\n",daten->pw_uid);
   printf("%d\n",daten->pw_gid);
   printf("%d\n",daten->pw_quota);
   printf("%s\n",daten->pw_comment);
   printf("%s\n",daten->pw_gecos);
   printf("%s\n",daten->pw_dir);
   printf("%s\n",daten->pw_shell);

                         /* Auf Dateianfang positionieren */
   setpwent();
   while((daten = getpwent()) != NULL)
         printf("Benutzername : %s\n",daten->pw_name);
}
```

Dateien

/etc/passwd

Liste aller Systembenutzer

> > > > getpwent, getpwuid, getpwnam, endpwent

Benutzernummer des Prozesses setzen

int setuid(bnr)
int bnr;

setuid setzt die reale und effektive Benutzernummer des Prozesses auf *bnr*. Die Benutzernummer *bnr* muß mit der realen Benutzernummer des Prozesses übereinstimmen; eine andere Zuordnung darf nur der Systemverwalter vornehmen.

Typ

Systemaufruf

Parameter

int bnr	neue effektive und reale Benutzernummer des Prozesses

Ergebnis

0	setuid hat die Benutzernummer neu gesetzt
-1	sonst

> > > > getuid, geteuid

CPU anhalten

void shutdn()

Nur für den Systemverwalter!
shutdn schreibt zunächst alle Daten im Kernspeicher, die gerettet werden müssen, auf Platte. Dazu gehören geänderte Superblöcke und Indexeinträge sowie Ein/Ausgabedaten, die noch nicht übertragen wurden.
Die Superblöcke von allen beschreibbaren Dateisystemen werden als "in Ordnung" markiert, so daß die Dateisysteme ohne Bereinigungsarbeiten eingehängt werden können, wenn das System wieder hochgefahren wird.
Dann schreibt shutdn

Normal System Shutdown

auf die Konsole und hält die CPU an.

Typ

Systemaufruf

Parameter

keine

Hinweis

Shutdn sperrt alle anderen Prozesse bevor es in Aktion tritt. Trotzdem sollten alle Benutzerprozesse mit dem Kommando kill vor Aufruf von shutdn abgebrochen werden, weil eventuell Dateisysteme als "nicht in Ordnung" markiert werden können.

> > > > mount, /etc/fsck(Kommando)

Signalbearbeitung

#include <signal.h>

int (*signal(sig,fkt) **)()**
int sig;
int (*fkt **)()**;

signal ist die Funktion, die Sie zur Signalbehandlung brauchen.
Signale, die ein Prozeß empfängt, können auf verschiedene Arten erzeugt werden:

- von außen : durch den Benutzer an der Datensichtstation, der eine Unterbrechungs-Taste (DEL, QUIT,...) drückt
- von außen : durch einen anderen Prozeß, der ein Signal schickt (alarm, kill)
- von innen : durch Programmfehler (Adreßfehler, Anstoß eines ungültigen Befehls, Division durch Null,...)

Hat ein Prozeß nichts in Bezug auf ein Signal vereinbart, so wird bei Eintreffen des Signals der Prozeß abgebrochen und der Vater des Prozeses erfährt vom Prozeßende.
Ein Prozeß kann aber ein Signal auch abfangen,indem er signal mit einer Funktion *fkt* aufruft. Er hat drei Möglichkeiten auf ein Signal zu reagieren:

- Voreinstellung SIG_DFL : Prozeßabbruch
- Vordefinierte Funktion SIG_IGN : Signal ignorieren
- Signalbehandlung gemäß der Funktion *fkt*:

  Bei Eintreffen des Signals wird der aufrufende Prozeß unterbrochen und die Funktion *fkt* ausgeführt. Nach Beendigung der Signalbehandlung wird der Prozeß an der Stelle fortgesetzt, an der er unterbrochen wurde, wenn nicht in *fkt* exit aufgerufen wurde.

Typ

Systemaufruf

Parameter

int sig

Signalnummer
In <signal.h> sind derzeit folgende Signale definiert:

SIGHUP 1	logoff von einer Datensichtstation
SIGINT 2	Unterbrechung von der Datensichtstation: Drücken der Taste [DEL]
SIGQUIT 3*	Abbruch von der Datensichtstation: Drücken der Taste [CTRL] \
SIGILL 4* +	unzulässiger Befehl
SIGTRAP 5* +	ptrace : Unterbrechungssignal
SIGIOT 6*	IOT Befehl
SIGEMT 7*	EMT Befehl
SIGFPE 8*	Fehler bei einer Gleitkommaoperation
SIGKILL 9	kill: unbedingter Prozeßabbruch kann weder abgefangen noch ignoriert werden!
SIGBUS 10*	Adreßfehler auf dem System-Bus
SIGSEGV 11*	Adreßfehler wegen unerlaubtem Segmentzugriff
SIGSYS 12*	ungültiges Argument für einen Systemaufruf
SIGPIPE 13	Ausgabe auf eine Pipe, deren Leseseite geschlossen ist
SIGALRM 14	alarm: Ablauf einer Zeitspanne
SIGTERM 15	Programmbeendigung wird von kill benützt
SIGUSR1 16	vom Benutzer definiert
SIGUSR2 17	vom Benutzer definiert Die vom Benutzer definierten Signale haben keinen Einfluß, falls sie nicht abgefangen werden.
SIGCLD 18 +	Sohnprozeß gestorben
SIGPWR 19 +	Stromausfall
SIGDVZ 20	Division durch 0
SIGXCPU 21	CPU-Zeit aufgebraucht
SIGPROF 22	profil: Unterbrechungssignal
SIGBPT 23	Haltepunkt
SIGMNI 24	Unterbrechung bei MMU debugging

Die mit * markierten Signale erzeugen einen Speicherabzug (core dump) des Programms, sofern sie nicht abgefangen oder ignoriert werden.
Bei einem Speicherabzug wird eine Kopie des Prozesses, so wie er im Hauptspeicher steht, in die Datei core im aktuellen Dateiverzeichnis geschrieben (falls eine solche existiert oder erstellt werden kann).
Diese Datei kann später mit dem Kommando adb untersucht werden, um die Ursache des Signals zu ermitteln.

Die mit + markierten Signale werden nicht auf SIG_DFL zurückgesetzt

(*fkt)() Name der Funktion, die bei Eintreffen des Signals *sig* aufgerufen werden soll. Die Funktion erhält als einziges Argument die Signalnummer vom Typ int.
Die Funktion muß vor dem signal Aufruf definiert werden. In <signal.h> gibt es zwei vordefinierte Funktionen:

```
typedef int FRI();
typedef FRI *PFRI;

PFRI signal();
#define SIG_DFL (PFRI)0
#define SIG_IGN (PFRI)1
```

SIG_DFL

Die Signalbehandlung wird auf die Voreinstellung zurückgesetzt, d.h. bei Eintreffen des Signals wird der Prozeß mit folgenden Konsequenzen abgebrochen:

- alle offenen Dateien des Prozesses werden vorher geschlossen
- ein wartender Vater erfährt vom Prozeßende
- wartet der Vater nicht (wait), endet der Prozeß als "Zombie"!
- alle Sohn- und Zombieprozesse dieses Prozesses erhalten den Initialisierungsprozeß als Vater (Prozeßnummer 1)

SIG_IGN

Es wird keine Signalbehandlung aufgerufen, das Signal wird ignoriert.

Ergebnis

Name der vorherigen Funktion zur Behandlung des Signals
bei Erfolg liefert signal die letzte Einstellung der Signalbehandlung, das kann SIG_DFL, SIG_IGN oder eine vom Benutzer geschriebene Funktion sein.

–1 Fehler, wenn

- *sig* eine ungültige Signalnummer ist oder
- *fkt* auf eine unzulässige Adresse zeigt

Fehlermeldung

bei Fehler wird errno mit einem Fehlercode besetzt:

EINVAL : Unzulässiges Argument
EFAULT : Unzulässige Adresse

Hinweis

- In Signalbehandlungsfunktionen sollte die Funktion sleep nicht verwendet werden, weil es dadurch eventuell zu rekursiven sleep Aufrufen kommen könnte. Das Verhalten bei rekursivem Aufruf von sleep ist aber undefiniert.
- Außer bei SIGILL, SIGTRAP, SIGCLD und SIGPWR wird bei allen Signalen nach einmaliger Ausführung von *fkt* die Signalbehandlung wieder auf SIG_DFL (Prozeßabbruch) gesetzt. Möchten Sie das Signal jedesmal mit *fkt* abfangen, müssen Sie dies explizit angeben. Dazu gibt es zwei Möglichkeiten:

  a) Sie verwenden die Funktionen setjmp und longjmp und bauen Ihr Programm nach folgendem Muster auf:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf umgeb;

abfang();
{
   printf("\n Signal eingetroffe\n");
         /* Rücksprung an die Stelle nach setjmp Aufruf */
   longjmp(umgeb,0);
```

```
}

main()
{
    .
    .
    .
    setjmp(umgeb);
          /* An diese Stelle wird zurückgesprungen */
    signal(SIGINT, abfang);
    .
    .
}
```

b) Sie rufen signal erneut im Rumpf von *fkt* auf.
Trifft dabei das gleiche Signal nochmal ein bevor der signal Aufruf ausgeführt wurde, haben Sie leider keine Chance, das Signal abzufangen.

- Die von einer Datensichtstation erzeugten Signale werden an alle Prozesse weitergeleitet, die von dieser Station aus gestartet wurden.

- Bei einigen Systemaufrufen kann es vorkommen, daß sie frühzeitig abgebrochen werden, wenn während ihrer Ausführung ein Signal abgefangen wird. Nach Beendigung der Signalbehandlung liefert der unterbrochene Systemaufruf das Ergebnis -1 und in errno den Fehlercode:
EINTR : Systemaufruf wurde unterbrochen.
Systemaufrufe, die eventuell abgebrochen werden, sind:

  read, write (auf langsame Geräte wie z.B. Datensichtstation), open, ioctl, pause, wait.

- Nach einem fork Aufruf übernimmt der Sohnprozeß die Signalbehandlungen des Vaterprozesses.

- Nach einem exec Aufruf ist jede Signalbehandlung wieder auf die Voreinstellung (Prozeßabbruch) zurückgesetzt.

Beispiel

Programm, das die Signale SIGINT (Drücken der Taste [DEL]) und SIGALRM abfängt:

```
#include <signal.h>
#include <stdio.h>
```

```
            /* Funktion wird beim Drücken der Taste [DEL]
               (SIGINT) bzw. automatisch nach Ablauf einer
               Zeitspanne von 1 Sekunden (SIGALRM)
               ausgeführt */
abfang(sig)
int sig;
{
            /* Piepst, wenn Signal eintrifft */
   printf("\07");
   alarm(1);   /* Zeit neu einstellen */
   if (sig == SIGALRM)
            /* Signalbehandlung wird für SIGALRM neu
               aufgesetzt. Nochmaliges Drücken der
               Taste [DEL] führt dagegen zum Programmabbruch */
       signal(SIGALRM,abfang);
}

main()
{
   FILE *fp;
   int c;
           /* Signalbehandlung  einrichten */
   signal(SIGINT,abfang);
   signal(SIGALRM,abfang);
            /* Zeit einstellen, nach einer Sekunde
               Verzweigung zur Signalbehandlung */
   alarm(1);

   fp = fopen("datei","r");
   while((c=getc(fp)) != EOF)
         putc(c,stdout);
   fclose(fp);
}
```

Dateien

/usr/include/signal.h

Definition der Signale und Standardsignalbehandlungen.

>>>> alarm, kill, kill(Kommando), ptrace, setjmp, longjmp, wait, pause, sleep

Sinus

#include <math.h>

double sin(x)
double x;

sin berechnet für eine zulässige Gleitkommazahl die trigonometrische Funktion Sinus.

Typ

C-Funktion

Parameter

double x   Gleitkommazahl, die den Winkel im Bogenmaß angibt

Ergebnis

sin(x)   eine Gleitkommazahl im Intervall [-1.0, +1.0]

Hinweis

Wenn Sie in Ihrem Programm sin verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

```
                /* Die Sinuswerte im Bereich von -1.0
                   bis +1.0 ausgeben. Schrittweite 0.1 */

#include <math.h>
main()
{
   double x;
   for (x = -1.0; x < 1.1; x = x +0.1)
       printf(" sin(%1.2f) = %.4f \n ",x,sin(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

>>>> cos, asin, sinh

Sinus Hyperbolicus

#include <math.h>

double sinh(x)
double x;

sinh berechnet den Sinus Hyperbolicus für zulässige Gleitkommazahlen.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl

Ergebnis

sinh(x)	bei Erfolg
+HUGE	bei Überlauf

Hinweis

Wenn Sie in Ihrem Programm sinh verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Die Sinushyperbolicuswerte aus dem Bereich von -1 bis 1 ausgeben. Schrittweite 0.1

```
#include <math.h>
main()
{
   double x;
   for (x = -1; x < 1.1; x = x +0.1)
       printf(" sinh(%.2f) = %.4f \n ",x,sinh(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > cosh, asin, sin

Prozeß für festgesetzte Zeitspanne anhalten

int sleep(sek**)**
unsigned sek;

sleep legt den Prozeß für *sek* Sekunden auf Eis, indem es zunächst mit alarm die Alarmuhr auf *sek* Sekunden stellt und dann mit pause den Prozeß anhält. Die vorher festgelegte Signalbehandlung für SIGALRM wird danach wieder eingestellt.
Ein Problem des sleep Aufrufes besteht darin, daß alarm Aufrufe nicht gekellert werden können (siehe alarm). Wenn Sie daher vor Aufruf von sleep die Alarmuhr bereits gestellt haben, können zwei Fälle eintreten:

- die vorher eingestellte Zeit ist kleiner als die Angabe in *sek*, etwa

  ```
  alarm(2);
  .
  .
  sleep(30);
  ```

  Dann wird nach Ablauf der vorher eingestellten Zeit (im Beispiel 2 Sek) der Alarm ausgelöst und der sleep Aufruf beendet.

- die vorher eingestellte Zeit ist größer als die Angabe in *sek*, etwa

  ```
  alarm(30);
  .
  .
  sleep(5);
  ```

  Dann hat der sleep Aufruf für die vorher eingestellte Alarmzeit keine Wirkung (im Beispiel: nach dem sleep Aufruf steht die Alarmuhr auf 25).

Typ

C-Funktion

Parameter

unsigned sek
: Zeitspanne, die der Prozeß angehalten werden soll

Ergebnis

angeforderte Zeit - tatsächliche Zeit
: das Ergebnis zeigt die noch übrige Zeit, falls sleep früher beendet wurde als in *sek* angegeben (s.o.).

Achtung

In Signalbehandlungsfunktionen, die eine Rückkehr vorsehen, sollte sleep nicht verwendet werden. Es könnte nämlich durch ein erneutes Eintreffen des Signals während der Signalbehandlung zu einem rekursiven Aufruf von sleep kommen. In diesem Fall ist jedoch das Verhalten undefiniert!

Hinweis

Die Zeit, die der Prozeß tatsächlich angehalten wird, kann auch noch aus folgenden Gründen von der Angabe in *sek* abweichen:

- sie kann bis zu einer Sekunde kürzer sein, weil das "Aufwecken" in festen 1-Sekunden-Intervallen stattfindet.
- sie kann aus Prioritätsgründen beliebig länger sein; das System hat Wichtigeres zu tun!

Beispiel

siehe Beispiel bei pause.

>>>> alarm, pause, signal

Formatierte Ausgabe in eine Zeichenreihe

#include <stdio.h>

char *sprintf(s,format [,arg]...**)**
char *s;
char *format;

sprintf wandelt die Argumente gemäß den Formatanweisungen um und speichert sie in den Bereich, auf den *s* zeigt. sprintf arbeitet wie printf, außer daß die Ausgabe in eine Zeichenreihe und nicht auf Standardausgabe geschrieben wird.

Typ

C-Funktion (s)

Parameter

← char *s — Zeiger auf die Ergebniszeichenreihe; *s* wird mit '\0' abgeschlossen

char *format

Die Formatzeichenreihe enthält drei Klassen von Zeichen:

a) Formatsteuerzeichen

- Neue Zeile ('\n')
- Rücksetzen ('\b')
- Tabulator ('\t')
- Seitenwechsel ('\f')
- Wagenrücklauf ('\r')

Die Formatsteuerzeichen werden wie normale Zeichen in *s* abgespeichert. Erst bei der Ausgabe von *s* auf den Bidschirm z.B. wird die Formatsteuerung wirksam.

b) Zeichen, die 1:1 ohne Umwandlung ausgegeben werden

c) Formatangaben für die Ausgabe der Argumente von der Form:

$$\% \begin{bmatrix} + \\ - \end{bmatrix} [0] \begin{bmatrix} n \\ * \end{bmatrix} \begin{bmatrix} .m \\ .* \end{bmatrix} \begin{Bmatrix} [l]\{d|o|x|u\} \\ \{D|O|X|U\} \\ \{c|e|f|g|s|\%\} \end{Bmatrix}$$

dabei bedeutet:

+ das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben

– linksbündig ausrichten

0 mit Nullen auffüllen
Standard: Leerzeichen werden links angefügt

n minimale Gesamtfeldbreite (inklusive Dezimalpunkt), falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung

* die Gesamtfeldbreite ist variabel;
der aktuelle Wert (Typ: unsigned) steht vor dem dazugehörenden Argument in der Argumentenliste

.m *m* gibt die Stellen nach dem Komma für eine e- oder f-Konvertierung an oder die maximale Anzahl von Zeichen, die von einer Zeichenreihe ausgegeben werden sollen.

.* Möglichkeit, die Stellen nach dem Komma oder die Länge einer Zeichenreihe variabel anzugeben; der aktuelle Wert (Typ: unsigned) steht vor dem dazugehörenden Argument in der Argumentenliste.

l Format für den Typ long int;
vor d, o, u, x

Folgende Parameter legen die eigentliche Umwandlung fest:

c einzelnes Zeichen (char); das Zeichen '\0' wird ignoriert

d Dezimaldarstellung einer ganzen Zahl (int)

D	Dezimaldarstellung einer ganzen Zahl (long int)
e	Gleitpunktzahl (float oder double) im Format [-]d.ddd**e**{+|-}dd Die Anzahl der Stellen nach dem Komma hängt von der Genauigkeitsangabe *.m* ab. Standard (keine Angabe) : 5 Stellen
f	Gleitkommazahl (float oder double) im Format [-]ddd.ddd Die Anzahl der Stellen nach dem Komma hängt von der Genauigkeitsangabe in *.m* ab: Standard (keine Angabe) : 6 Stellen 0 : ganzahlige Ausgabe ohne Dezimalpunkt
g	Gleitkommazahl (float oder double) im d, f oder e Format, je nachdem, welche Darstellung unter Wahrung der Genauigkeit am wenigsten Platz beansprucht
o	Oktale Darstellung einer ganzen Zahl (int)
O	Oktale Darstellung einer ganzen Zahl (long int)
s	Format für Zeichenreihen Die Zeichenreihe sollte mit '\0' abgeschlossen sein. sprintf schreibt so viele Zeichen der Zeichenreihe, wie in *.m* angegeben ist: Standard (keine Angabe) oder 0 : sprintf schreibt alle Zeichen bis '\0'
u	Vorzeichenlose Dezimalzahl (unsigned). Format zur Ausgabe von Zeigerwerten
x	hexadezimale Darstellung einer ganzen Zahl (int)
X	hexadezimale Darstellung einer ganzen Zahl (long int)
%	Ausdruck von %, keine Umwandlung

Ergebnis

Zeiger auf die Ergebniszeichenreihe s
bei Erfolg

Achtung

Sie müssen dafür sorgen, daß der Bereich, auf den *s* zeigt, groß genug ist, um das Ergebnis abzuspeichern!

Beispiel

Sie können sprintf z.B. dazu benutzen eine Zeichenreihe zu kopieren. Damit kann man die Funktion strncpy implementieren. Das Beispiel bei strncpy würde dann wie folgt aussehen :

```
#include <stdio.h>

main ()
{
   int n;
   char *s2 = "Peter geht schwimmen !";
   char s1[BUFSIZ];
   printf("Der Satz lautet : %s \nWieviel Zeichen kopieren ?",s2);
   scanf("%d",&n);

               /* Alternativ könnte an dieser Stelle
                  folgender Aufruf stehen:
                  strncpy(s1,s2,n);          */

  sprintf(s1,"%.*s",n,s2);
  printf("%s \n",s1);
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > printf, fprintf, putc, putchar, puts, sscanf

Quadratwurzel

```
#include <math.h>

double sqrt(x)
double x;
```

sqrt berechnet die Quadratwurzel zu einer positiven Gleitkommazahl.

Typ

C-Funktion

Parameter

double x	positive Gleitkommazahl

Ergebnis

sqrt(x)	falls *x* im zulässigen Gleitkommaintervall liegt
0	falls *x* negativ ist

Fehlermeldung

Falls *x* negativ ist, steht in errno der Fehlercode:

EDOM : Argument zu groß

Hinweis

Wenn Sie in Ihrem Programm sqrt verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Berechne die Quadratwurzel für einen eingelesenen Wert *x*:

```
#include <math.h>
int errno;

main()
{
   double x;
   scanf("%lf\n",&x);
   printf("Wurzel aus %g : %g\n",x,sqrt(x));
   printf("%d\n", errno);
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > exp, pow, log, log10

Initialisierung des Zufallszahlgenerators

void srand(i)
int i;

Mit srand können Sie den Zufallszahlgenerator initialisieren, der von rand aufgerufen wird.

Typ

C-Funktion

Parameter

int i	beliebige ganze Zahl, die den Zufallszahlgenerator auf eine Zufallszahl setzt
1	Der Zufallszahlgenerator wird auf seine voreingestellte Startzahl gesetzt

Beispiel

siehe Beispiel bei rand

>>>> rand

Formatiertes Einlesen aus einer Zeichenreihe

int sscanf(s,format [,arg_zg]...**)**
char *s, *format;
<typ> *arg_zg, ...;

sscanf liest die Daten aus der Zeichenreihe *s*. Wie scanf wandelt die Funktion ein Eingabefeld gemäß seiner zugeordneten Formatanweisung um und speichert das Ergebnis an die Stelle, die der entsprechende Ergebniszeiger angibt.

Typ

C-Funktion (s)

Parameter

char *s — Zeichenreihe, die die Eingabedaten enthält

char *format

Die Formatzeichenreihe kann drei Klassen von Zeichen enthalten:

a) Zwischenraum

- Tabulator
- Leerzeichen
- Zeilenwechsel

Zwischenraumzeichen in der Eingabe werden außer bei %c ignoriert.

b) beliebiges Zeichen außer '%'

das Zeichen muß mit dem nächsten Zeichen aus der Eingabe übereinstimmen, ansonsten wird die Eingabebearbeitung abgebrochen.

c) Formatangaben für die Umwandlung der eingelesenen Zeichen. Eine Formatangabe ist von der Form:

```
                 ┌ [l] {d|e|f|o|x}       ┐
                 │     {D|E|F|O|X}       │
% [*] [n]       {      {c|s|%}           }
                 │     {[...]|[^...]}    │
                 └                       ┘
```

Zu einer Formatangabe gehört ein Eingabefeld, d.h. eine Zeichenreihe ohne Leerzeichen. Führende Tabulator- und Leerzeichen werden übersprungen (beachte jedoch Format %c!). Die Länge der Zeichenreihe ist entweder explizit durch die Feldbreite *n* festgelegt oder ergibt sich implizit, sobald das erste Zeichen gelesen wird, das nicht zu der Formatangabe paßt.
Das erste '%' kennzeichnet die Formatangabe, die restlichen Zeichen werden wie folgt interpretiert:

* überspringe eine Zuweisung:
das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber nicht abgespeichert.

n maximale Länge der umzuwandelnden Eingabezeichenreihe; tritt vorher ein Zeichen auf, das nicht zur Formatangabe paßt, wird die Länge entsprechend gekürzt.

l Angabe für doppelte Länge:

vor d, o, x

Umwandlung einer Dezimal-, Oktal- oder Hexadezimalzahl in long int

← long *arg_zg

vor e, f

Gleitkommazahl
Umwandlung in double

← double *arg_zg

Folgende Formatelemente legen die eigentliche Umwandlung fest:

Formatelement/ Ergebniszeiger	erwartete Eingabe
c ← char *arg_zg	einzelnes (abdruckbares) Zeichen; Achtung ein einzelnes Zwischenraumzeichen wird auch gelesen und nicht wie sonst übersprungen!
nc ← char arg_zg[]	Feld von *n* (abdruckbaren) Zeichen Achtung führende Tabulator- und Leerzeichen werden übersprungen, aber Zwischenraumzeichen innerhalb des Eingabefeldes nicht (sind keine Trenner mehr)!
d ← int *arg_zg	ganze Dezimalzahl (mit oder ohne führende Null)
D ← long *arg_zg	ganze Dezimalzahl (mit oder ohne führende Null) Umwandlung wie bei ld
e ← float *arg_zg	Gleitkommazahl im Format: [+|-]dd[.dd{e|E}[+|-]dd]
E ← double *arg_zg	Gleitkommazahl wie bei e Umwandlung wie bei le
f ← float *arg_zg	Gleitkommazahl im Format: dd[.ddd]
F ← double *arg_zg	Gleitkommazahl wie bei f Umwandlung wie bei lf
o ← int *arg_zg	ganze Oktalzahl (mit oder ohne führende Null)

Formatelement/ Ergebniszeiger	erwartete Eingabe
O ← long *arg_zg	ganze Oktalzahl (mit oder ohne führende Null) Umwandlung wie bei lo
s 1s ← char arg_zg[]	Zeichenreihe; Das zugehörige Eingabefeld wird durch ein Zwischenraumzeichen abgeschlossen, falls eine Feldbreite *n* angegeben ist, werden *n* Zeichen oder bis zum nächsten Zwischenraum gelesen, je nachdem was vorher kommt (Zwischenraumzeichen sind hier also Trenner, vgl %nc!) Die eingelesene Zeichenreihe wird mit '\0' abgeschlossen. Führende Tabulator- und Leerzeichen werden übersprungen, also liest das erste Nicht-Leerzeichen (vgl %c!).
x ← int *arg_zg	ganze Hexadezimalzahl (mit oder ohne führende Null) (aber nicht Oxddd!)
X ← long *arg_zg	ganze Hexadezimalzahl (mit oder ohne führende Null) Umwandlung wie bei lx
[...] ← char arg_zg[]	Zeichenreihe, die nur aus Zeichen besteht, die in [...] vorkommen, z.B: [a] : Zeichenreihe, die nur aus 'a' besteht. Es wird eingelesen bis zum ersten Zeichen, das nicht in [...] vorkommt.
[^...] ← char arg_zg[]	Zeichenreihe, die nur aus Zeichen besteht, die nicht in [^...] vorkommen, z.B: [^a]: Zeichenreihe, in der kein 'a' vorkommt. Es wird eingelesen bis das erste Zeichen auftritt, das in [^...] vorkommt.
%	das Zeichen '%' selbst.

Ergebnis

Anzahl der eingelesenen und erfolgreich umgewandelten Datenelemente

Mit dem Ergebnis eines erfolgreichen sscanf Aufrufes können Sie feststellen, wieviele Datenelemente tatsächlich eingelesen wurden.

EOF	Dateiende
0	Konvertierungsfehler

Beispiel

Der Aufruf:

```
int i;
float x;
char p[10];
char *name = "25341     234   nasowas";
sscanf(name,"%2d %f %*d %6s",&i,&x,p);
```

weist die Daten aus der Zeichenreihe *name* wie folgt zu:

```
i    : 25
x    : 341.0
p    : nasowa\0
```

Die Zuweisung von 234 wird wegen der %*d Angabe nicht ausgeführt.

> > > > fgetc, fgets, fread, fopen, setbuf, fscanf, scanf

Dateiinformationen ausgeben

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(name,dinf)
char *name;
struct stat *dinf;
```

stat liefert ausführliche Information über die Datei mit Dateinamen *name*. Der Unterschied zu fstat besteht lediglich darin, daß die Datei mit ihrem Namen statt über ihre Dateikennzahl angesprochen wird.

Typ

Systemaufruf

Parameter

char *name

Dateiname

Für die Datei sind keine Zugriffsrechte erforderlich, lediglich alle Dateiverzeichnisse auf dem Pfad zu *name* müssen durchsucht werden können

← struct stat *dinf

Zeiger auf eine Struktur, die in <sys/stat.h> wie folgt definiert ist:

```
struct  stat
{
        dev_t   st_dev;           /* major- und minor-Nummer */
                                  /* des Gerätes, das einen */
                                  /* Dateiverzeichniseintrag für */
                                  /* diese Datei enthält */
        ino_t   st_ino;           /* Indexnummer */
        unsigned short st_mode;   /* Dateityp, Zugriffsrechte */
        short   st_nlink;         /* Anzahl der Verweise */
        short   st_uid;           /* Benutzernr. des Eigentümers */
        short   st_gid;           /* Gruppennummer */
        dev_t   st_rdev;          /* major- und minor-Nummer */
                                  /* nur für zeichen- oder block- */
                                  /* orientierte Gerätedateien */
        off_t   st_size;          /* Dateigröße in Bytes */
        time_t  st_atime;         /* letzter Zugriff */
        time_t  st_mtime;         /* letzte Änderung */
        time_t  st_ctime;         /* letzte Änderung des Datei-*/
                                  /* status */
};
```

<sys/stat.h> enthält außerdem Definitionen von symbolischen Konstanten, mit denen die Bitbelegung der Komponente st_mode gedeutet werden, u.a.:

```
#define S_IFMT   0170000     /* Dateityp Maske */
#define S_IFDIR  0040000     /* Dateiverzeichnis */
#define S_IFCHR  0020000     /* Zeichen-orientiert */
#define S_IFBLK  0060000     /* Block-orientiert */
#define S_IFREG  0100000     /* normale Datei */

#define S_ISUID  0004000     /* s-Bit für Eigentümer */
#define S_ISGID  0002000     /* s-Bit für Gruppe */
#define S_ISVTX  0001000     /* sticky Bit */
#define S_IREAD  0000400     /* Leseerlaubnis, Eigentümer */
#define S_IWRITE 0000200     /* Schreiberlaubnis, Eigentümer */
#define S_IEXEC  0000100     /* Ausführerlaubnis, Eigentümer */
```

Die Bitbelegungen 0000010 bis 0000070 geben die Zugriffsrechte für Gruppe an, die Bitbelegungen 0000001 bis 0000007 die Zugriffsrechte für Andere.
Dabei bedeutet wie bei chmod:

1 : ausführen
2 : schreiben
4 : lesen

Ergebnis

0 stat hat die Dateiinformationen in *dinf* abgespeichert.

–1 Fehler, wenn

- eine Komponete auf dem Pfad zu *name* kein Dateiverzeichnis ist oder
- die Datei nicht existiert oder
- ein Dateiverzeichnis auf dem Pfad zu *name* nicht durchsucht werden darf.

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENOTDIR : Kein Dateiverzeichnis
ENOENT : Datei oder Dateiverzeichnis unbekannt
EACCES : Zugriff untersagt

Achtung

Sie müssen den Speicherplatz für die Ergebnisstruktur stat explizit bereitstellen!

Hinweis

- die Datentypen in stat sind verschiedene integer Typen. Sie sind in <sys/types.h> definiert und haben folgende Bedeutung:

ino_t	Datentyp für eine Indexnummer
time_t	Datentyp für eine Zeitangabe in Sekunden
dev_t	Datentyp für eine major- und minor-Nummer
off_t	Datentyp für die Position des Lese/Schreibzeigers

- Die drei Zeitangaben werden bei folgenden Systemaufrufen verändert:

st_atime :	creat, fcntl, locking, mknod, pipe, utime, read
st_mtime :	creat, mknod, pipe, utime, write
st_ctime :	chmod, chown, creat, link, mknod, pipe, unlink, utime, write.

- Die Komponente st_atime, die den letzten Zugriff angibt, wird nicht verändert, wenn ein Dateiverzeichnis durchsucht wird!

- Die Komponente st_mtime, die die letzte Änderung angibt, ist von Änderungen des Indexeintrages nicht betroffen!

Beispiel

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

char *gctime();

main(argc,argv)
int argc;
char **argv;
{
   struct stat infos;      /* Ergebnisstruktur für Dateiinformation */
   if(stat(argv[1],&infos) == -1)
      {
        printf("Datei bzw. Dateiverzeichnis nicht vorhanden\n");
        exit(1);
      }
                           /* Überprüfung des Dateityps */
   if((infos.st_mode & S_IFMT) == S_IFDIR)
      printf("Dateiverzeichnis : %s\n",argv[1]);
   if((infos.st_mode & S_IFMT) == S_IFREG)
      printf("normale Datei : %s\n",argv[1]);
                           /* ======================================
                              Änderung der Schutzbitbelegung unter Ver-
                              wendung der Konstanten S_ISUID und S_IEXEC */
   chmod(argv[1], (S_ISUID | S_IEXEC));
                           /* s-Bit und Ausführberechtigung für
                              Eigentümer */
                           /* ===================================
                              Detaillierte Information über eine Datei
                              ausgeben  */
   printf("benutzter Speicherplatz : %ld\n",infos.st_size);
   printf("letzte Änderung        : %s", gctime(&infos.st_mtime));
}
```

Dateien

/usr/include/sys/filsys
: Definitionen für die Implementierung des Dateisystems

/usr/include/types.h
: Typdefinitionen

/usr/include/stat.h
: Definition der Struktur stat

> > > > fstat, ls(Kommando)

Systemuhr stellen

int stime(sek_zg**)**
long *sek_zg;

Nur für den Systemverwalter!
Mit stime kann der Systemverwalter die Systemuhr stellen, wobei die Zeitangabe, auf die *sek_zg* zeigt, die Zeit seit dem 1.Januar 1970 00:00:00 (GMT) in Sekunden angibt.

Typ

Systemaufruf

Parameter

long *sek_zg
: Zeiger auf die Zeitangabe in Sekunden, mit der die Systemuhr gestellt werden soll.

Ergebnis

0	bei Erfolg
-1	Fehler, wenn der Prozeß nicht unter der Kennung des Systemverwalters läuft.

Fehlermeldung

Bei Ergebnis -1 steht in errno der Fehlercode:

EPERM : Hat anderen Eigentümer

Hinweis

time liefert die Uhrzeit, die durch stime eingestellt wurde.

> > > > time, ctime, date(Kommando)

Konkatenation von Zeichenreihen

char *strcat(s1,s2**)**
char *s1, *s2;

strcat hängt eine Kopie der Zeichenreihe *s2* ans Ende der Zeichenreihe *s1* und liefert einen Zeiger auf *s1* zurück.

Typ

C-Funktion

Parameter

← char *s1	Ergebniszeichenreihe, strcat schließt die Zeichenreihe mit '\0' ab.
char *s2	Zeichenreihe, die angehängt werden soll

Ergebnis

Zeiger auf die Ergebniszeichenreihe

Achtung

strcat überprüft nicht, ob *s1* für das Ergebnis lang genug ist; dafür müssen Sie selbst sorgen!

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

Programmstück, das den Dateiverzeichnis-Namen des aktuellen Aufrufers erzeugt:

```
char *meindv;
static char dv[20] = "/usr/";

    meindv = strcat(dv,getlogin());
```

>>>> strncat

Vergleich von zwei Zeichenreihen

int strcmp(s1,s2**)**
char *s1, *s2;

strcmp vergleicht die Zeichenreihen *s1* und *s2* in Bezug auf lexikographische Ordnung, z.B:

"Affenhaus" ist lexikographisch kleiner als "Affentheater"

Typ

C-Funktion

Parameter

char *s1	Zeichenreihe
char *s2	Zeichenreihe

Ergebnis

< 0	*s1* ist lexikographisch kleiner als *s2*
0	*s1* und *s2* sind lexikographisch gleich groß
> 0	*s1* ist lexikographisch größer als *s2*

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

Programm, das einen eingelesenen Namen in der *kartei* sucht:

```
char kartei[];

main(argc,argv)
int argc;
char **argv;
{
  int j,i=0;
  while(kartei[i] && (j = strcmp(argv[1],kartei[i++])));
  if (j == 0)
     printf("Der Kandidat ist schon bekannt!\n");
  else
     printf("Das muß ein Neuer sein, nix wie her damit!\n");
}
```

>>>> strncmp

Zeichenreihe kopieren

char *strcpy(s1,s2**)**
char *s1, *s2;

strcpy kopiert die Zeichenreihe *s2* auf die Zeichenreihe *s1*.

Typ

C-Funktion

Parameter

← char *s1	Ergebniszeichenreihe
char *s2	Zeichenreihe, die kopiert werden soll

Ergebnis

Zeiger auf die Ergebniszeichenreihe s1
s2 wird einschließlich dem Nullbyte '\0' kopiert.

Achtung

strcpy überprüft nicht, ob *s1* für das Ergebnis lang genug ist; dafür müssen Sie selbst sorgen!

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

```
              /* Folgendes Programm gibt die Inhalte
                 von s1 und s2 aus, ruft dann strcpy
                 auf und gibt nochmal beide Inhalte aus. */

main()
{
    char *strcpy();
    char *s1 = "Silvia hat es gut !";
    char *s2 = "Peter weniger !";
    printf("Inhalt s1: %s\nInhalt s2: %s\n",s1,s2);

              /* s2 nach s1 kopieren */
    strcpy(s1,s2);

    printf("Nach strcpy:\nInhalt s1: %s\nInhalt s2: %s\n",s1,s2);
}
```

> > > > strncpy

Temporäre Kopie einer Zeichenreihe

char *strdup(s)
char *s;

strdup kopiert die Zeichenreihe *s* in einen **statischen** Bereich, der beim nächsten Aufruf überschrieben wird. Den erforderlichen Bereich liefert malloc.

Typ

C-Funktion

Parameter

char *s	Zeichenreihe, die kopiert werden soll

Ergebnis

Zeiger auf den statischen Bereich, in dem die Kopie steht

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

Das Programm gibt die Zeichenreihe *s* zweimal aus, einmal direkt und einmal mit Hilfe von strdup:

```
main()
{
   char *strdup();
   char *s = "Diesen Satz gibt es jetzt 2-mal.";
   printf("%s\n%s\n",s,strdup(s));
}
```

Länge einer Zeichenreihe

int strlen(s)
char *s;

strlen bestimmt die Länge der Zeichenreihe *s*.

Typ

C-Funktion

Parameter

char *s	Zeichenreihe

Ergebnis

Länge von s
Das Nullbyte am Ende wird nicht mitgezählt!

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

Das Programm liest eine Zeichenreihe ein und berechnet ihren Speicherplatzbedarf. Wobei zu beachten ist, daß strlen das Nullbyte nicht mitzählt.

```
#include <stdio.h>
main()
{
    char s[BUFSIZ];
    printf("Geben Sie bitte Ihre Zeichenreihe ein.\n");
    scanf("%s",s);
    printf("Benoetigter Speicherplatz fuer diese Zeichenreihe: %d\n",strlen(s)+1);
}
```

> > > > strcat, strcpy, strcmp

Konkatenation von Zeichenreihen

char *strncat(s1,s2,n**)**
char *s1, *s2;
int n;

strncat hängt maximal *n* Zeichen der Zeichenreihe *s2* ans Ende der Zeichenreihe *s1* und liefert einen Zeiger auf *s1* zurück.

Typ

C-Funktion

Parameter

← char *s1	Ergebniszeichenreihe, strncat schließt die Zeichenreihe mit '\0' ab.
char *s2	Zeichenreihe, die angehängt werden soll
int n	positive ganze Zahl, die angibt, wieviele Zeichen maximal angehängt werden sollen.

Ergebnis

Zeiger auf die Ergebniszeichenreihe

Achtung

strncat überprüft nicht, ob *s1* für das Ergebnis lang genug ist; dafür müssen Sie selbst sorgen!

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

Programmstück, das Namen erzeugt, die höchstens 10 Zeichen haben:

```
char *meindv;
static char tz[10] = "Tanz/";

    meindv = strncat(tz,argv[1],5);
```

>>>> strcat

Vergleich von zwei Zeichenreihen

```
int strncmp(s1,s2,n)
char *s1, *s2;
int n;
```

strncmp vergleicht die Zeichenreihen *s1* und *s2* bis zur maximalen Länge *n* in Bezug auf lexikographische Ordnung, z.B liefert

strncmp("Sau","Saustall",3)

das Ergebnis 0, weil die beiden Argumente in den ersten drei Zeichen übereinstimmen.

Typ

C-Funktion

Parameter

char *s1	Zeichenreihe
char *s2	Zeichenreihe
int n	Länge bis zu der verglichen werden soll

Ergebnis

< 0	*s1* ist in den ersten *n* Zeichen lexikographisch kleiner als *s2*
0	*s1* und *s2* sind in den ersten *n* Zeichen lexikographisch gleich groß
> 0	*s1* ist in den ersten *n* Zeichen lexikographisch größer als *s2*.

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

In folgendem Rate-Programm wird strncmp dazu benutzt die lexikographische Ordnung zweier Zeichenreihen zu bestimmen.

```
#include <stdio.h>
main()
{
    int i,n,wert;
    char s[BUFSIZ],wort[BUFSIZ];
    printf("Bitte geben Sie das zu ratende Wort ein : ");
                            /* Shellkommando-Aufruf :
                               Bildschirmausgabe unterdrücken */
    system("stty -echo cbreak");
    scanf("%s",&wort);
                            /* Shellkommando-Aufruf :
                               Bildschirmausgabe normal */
    system("stty echo -cbreak");
    n = strlen(wort);
    printf("\nDas eingegebene Wort hat %d Buchstaben.\n",n);
    i = 0;
    do
    {
       i++;
       printf("Ihr Versuch : ");
       scanf("%s",&s);
                            /* wert wird das Ergebnis von
                               strncmp zugewiesen */
       if (strlen(s) > n)
          {
            printf("Ihre Eingabe ist zu lang!\n");
            continue;
          }
       wert = strncmp(s,wort,n);
       if (wert > 0)
           printf("%s ist lexikographisch groesser.\n",s);
        else {
                  if (wert < 0)
                      printf("%s ist lexikographisch kleiner.\n",s);
             }
    }
    while (wert != 0);
    printf("Richtig das Wort hies : %s\n",wort);
    printf("Sie haben %d Versuche gebraucht.\n",i);
}
```

>>>> strcmp

Zeichenreihe kopieren

char *strncpy(s1,s2,n**)**
char *s1, *s2;
int n;

strncpy kopiert die ersten *n* Zeichen der Zeichenreihe *s2* auf die Zeichenreihe *s1.*

Typ

C-Funktion

Parameter

← char *s1	Ergebniszeichenreihe
char *s2	Zeichenreihe, die kopiert werden soll
int n	Länge bis zu der kopiert werden soll

Ergebnis

Zeiger auf die Ergebniszeichenreihe s1
s2 wird auf die Länge *n* gebracht, d.h eventuell gekürzt oder mit Nullbytes aufgefüllt. Ist die ursprüngliche Länge von *s2* größer oder gleich *n,* ist die Ergebniszeichenreihe nicht mehr mit dem Nullbyte abgeschlossen!

Achtung

strncpy überprüft nicht, ob *s1* für das Ergebnis lang genug ist; dafür müssen Sie selbst sorgen!

Hinweis

Alle Funktionen, die Zeichenreihen bearbeiten, erwarten als Argumente immer Zeichenreihen die mit dem Nullbyte '\0' abgeschlossen sind.

Beispiel

Das Programm kopiert die Zeichenreihe *s2* in die Zeichenreihe *s1* bis zur eingelesenen Länge.

```
#include<stdio.h>
main()
{
    int n;
    char *strncpy();
    char *s2 = "Peter geht schwimmen !";
    char *s1 = "                      ";
    printf("der Satz lautet : %s\nWieviel Zeichen kopieren ? ",s2);
    scanf("%d",&n);
    printf("%s\n",strncpy(s1,s2,n));
}
```

>>>> strcpy

Eigenschaften einer seriellen Schnittstelle festlegen

#include <sgtty.h>

int stty(dk,sssp**)**
int dk;
struct sgttyb *sssp;

stty legt Eigenschaften der seriellen Schnittstelle, die über die Dateikennzahl *dk* angesprochen wird, fest. Eigenschaften sind in der Struktur, auf die *sssp* zeigt, erfaßt.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl für die serielle Schnittstelle

struct *sgttyb sssp
Zeiger auf die Struktur, in der die Eigenschaften der seriellen Schnittstelle erfaßt sind. Die Struktur sgttyb ist in <sgtty.h> wie folgt definiert:

```
struct sgttyb {
char    sg_ispeed;    /* Eingabegeschwindigkeit */
char    sg_ospeed;    /* Ausgabegeschwindigkeit */
char    sg_erase;     /* Korrekturzeichen ([CTRL] h) */
char    sg_kill;      /* Abbruchzeichen ([CTRL] x) */
int     sg_flags;     /* Merkmale */
};
```

Die einzelnen Bits der Komponente sg_flags sind wie folgt festgelegt:

```
#define TANDEM  01      /* automatische Flußkontrolle */
#define CBREAK  02      /* cbreak Modus: jedes Zeichen */
                        /* wird sofort weitergeleitet */
#define LCASE   04      /* Großbuchstaben in Klein- */
                        /* buchstaben umwandeln */
#define ECHO    010     /* Echo (voll duplex) */
#define CRMOD   020     /* CR auf LF setzen, CR und */
                        /* LF werden als CR-LF aus- */
                        /* gegeben */
#define RAW     040     /* raw Modus: keine Zeichen- */
                        /* bearbeitung (roh, 8-Bit) */
#define ODDP    0100    /* ungerade Parität */
#define EVENP   0200    /* gerade Parität */
#define ANYP    0300    /* keine Parität aber */
                        /* 7-Bit-Übertragung */
#define NLDELAY 001400 /* Verzögerung für NL */
#define TBDELAY 006000 /* Verzögerung für TAB */
#define XTABS   06000  /* Tabulator expandieren */
#define CRDELAY 030000 /* Verzögerung für CR */
#define VTDELAY 040000 /* Verzögerung für FF */
                        /* VT */
#define BSDELAY 0100000/* Verzögerung für BS */
#define ALLDELAY 0177400 /* Maske */
```

die Kürzel (CR,LF,...) haben die übliche Bedeutung.

Ergebnis

0	bei Erfolg
-1	Fehler, wenn die Dateikennzahl *dk* nicht die erforderliche serielle Schnittstelle bezeichnet.

Hinweis

- Zusammen mit gtty kann stty dazu benutzt werden, Bildschirmeigenschaften neu zu definieren.
- Ist die serielle Schnittstelle ein Bidschirm, dann sind die Aufrufe:

  stty(dk,sssp)
  und ioctl(dk,TIOCSETP,sssp)

  äquivalent.

Beispiel

siehe Beispiel bei gtty.

Dateien

/usr/include/sgtty.h
Definition der Struktur sgttyb

/dev/tty — Kontrolldatei

/dev/tty* — Gerätedatei für den jeweiligen Bildschirm

/dev/console
Gerätedatei für die Konsole

>>>> ioctl, gtty, stty(Kommando)

Bytes vertauschen

void swab(von,nach,anz**)**
char *von, *nach;
int anz;

swab kopiert *anz* Bytes aus dem Vektor *von* in den Vektor *nach*, wobei jeweils zwei benachbarte Bytes vertauscht werden.

Typ

C-Funktion

Parameter

char *von	Eingabevektor
← char *nach	Ergebnisvektor
int anz	Anzahl der Bytes, die kopiert werden sollen. *anz* sollte gerade und positiv sein.

Hinweis

Die Funktion ist nützlich, wenn Daten zwischen verschiedenen Maschinen ausgetauscht werden sollen.

Beispiel

Das Beispiel zeigt, wie swab die benachbarten Bytes vertauscht.

```
char a[] = "aHcs!h";
char b[] = "        ";

main()
{
   swab(&a,&b,10);
   printf("%s\n%s\n",a,b);
}
```

Superblock aktualisieren

void sync()

sync schreibt alle Daten, die das Dateisystem betreffen, aus den Systempuffern auf Platte. Dazu gehören veränderte Superblöcke und Indexeinträge sowie noch nicht ausgegebene gepufferte Ausgabe.

Typ

Systemaufruf

Parameter

keine

Hinweis

- Der Begriff "Systempuffer" bezeichnet Puffer, die für den Benutzer nicht zugänglich sind und lediglich vom System benutzt werden.
  Sie sind also nicht zu verwechseln mit den Puffern, die bei den Standardein/ausgabe-Funktionen verwendet werden!
- Bei Rückkehr von sync kann die Ausgabe eventuell noch nicht beendet sein.
- sync sollte bei allen Programmen verwendet werden, die das Dateisystem überprüfen, wie /etc/fsck, /etc/dcheck, /etc/ncheck, /etc/df usw.
- sync sollte auf alle Fälle ausgeführt werden bevor das System angehalten wird und beim Systemstart.

> > > > sync(Kommando), /etc/update(Kommando)

Shell Kommando ausführen

int system(kommz**)**
char *kommz;

system startet einen zweiten Prozeß, der das shell-Kommando ausführt, das in *kommz* steht. Der aufrufende Prozeß wartet bis die Shell fertig ist und erhält als Ergebnis des system Aufrufs den Endestatus des ausgeführten Kommandos.

Typ

C-Funktion

Parameter

char *kommz
Zeichenreihe, die als Shell-Kommandozeile interpretiert wird.

Ergebnis

Endestatus des ausgeführten Kommandos
bei Erfolg

ungleich 0 das Kommando konnte nicht ausgeführt werden

Achtung

Denken Sie daran, vor Aufruf von system Ihren Puffer zu entleeren (fflush), da es sonst zu Unstimmigkeiten zwischen der bereits vorhandenen Ein/Ausgabe Ihres Programms und der des Kommandos kommen kann!

Beispiel

Programmstück, das das aktuelle Datum vor der normalen Ausgabe ausgibt:

```
main()
{
  system("datum");
   .
   .
   .
}
```

>>>> popen, exec, wait

Tangens

```
#include <math.h>

double tan(x)
double x;
```

tan berechnet im zulässigen Gleitkommaintervall die trigonometrische Funktion Tangens.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl, die den Winkel im Bogenmaß angibt

Ergebnis

tan(x)	falls *x* eine zulässige Gleitkommazahl ist
HUGE	sonst

Hinweis

Wenn Sie in Ihrem Programm tan verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Das folgende Programmm gibt den Tangens einer eingelesenen Zahl aus.

```
#include <math.h>
main ()
{
    double x;
    printf("Geben Sie bitte eine Zahl ein : ");
    scanf("%lf",&x);
    printf("Der Tangens von %g ist %g \n",x,tan(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

>>>> sin, cos, tanh, atan

Tangens Hyperbolicus

#include <math.h>

double tanh(x)
double x;

tanh berechnet im zulässigen Gleitkommaintervall die Funktion Tangens Hyperbolicus.

Typ

C-Funktion

Parameter

double x	Gleitkommazahl

Ergebnis

tanh(x)	falls *x* eine zulässige Gleitkommazahl ist
HUGE	sonst

Hinweis

Wenn Sie in Ihrem Programm tanh verwenden, müssen Sie den Übersetzer mit cc progname -lm aufrufen.

Beispiel

Das folgende Programm gibt den Tangenshyperbolicus einer eingelesenen Zahl aus.

```
#include <math.h>
main ()
{
     double x;
     printf("Geben Sie bitte eine Zahl ein : ");
     scanf("%lf",&x);
     printf("Der Tangenshyperbolicus von %g ist %g \n",x,tanh(x));
}
```

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > sin, cos, tan, atan

Aktuelle Position des Lese/Schreibzeigers

long tell(dk)
int dk;

tell bestimmt die aktuelle Position des Lese/Schreibzeigers für die Datei mit Dateikennzahl *dk*.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl der Datei, deren Lese/Schreibposition bestimmt werden soll

Ergebnis

Anzahl der Bytes, die der Lese/Schreibzeiger vom Dateianfang entfernt ist

	bei Erfolg
–1	Fehler, falls – *dk* eine ungültige Dateikennzahl ist oder – *dk* eine Pipe bezeichnet oder – *dk* eine Gerätedatei bezeichnet

Hinweis

Die Aufrufe tell(dk) und lseek(dk, 0L, 1) sind äquivalent.

Beispiel

siehe Beispiel bei lseek.

> > > > lseek, fseek, ftell

Programmierhilfe für bildschirmorientierte Anwendungen

termcap besteht aus einer Datei /etc/termcap, in der Beschreibungen von verschiedenen Bildschirmtypen abgelegt sind und einer Bibliothek von C-Funktionen, die den Zugriff auf diese Daten und deren Verarbeitung ermöglichen.
termcap ist eine Programmierhilfe, mit der bildschirmorientierte Anwendungen wie Emulatoren, Editoren, Menusysteme, usw. unabhängig von dem jeweiligen Bildschirm programmiert werden können.
Diese Konzeption macht ein Programm unabhängig von dem jeweiligen Typ der Bedieneinheit und dadurch leichter portierbar.

UMGEBUNGSVARIABLEN

Wenn ein Programm startet, steht ihm vom aufrufenden Programm ein Vektor (char **environ) zur Verfügung, in dem gewisse "Umgebungsvariablen" definiert sind (siehe exec). Sie können die vorbesetzten Variablen mit dem Kommando printenv oder der C-Funktion getenv erfragen.
Zu den Umgebungsvariablen gehören die Variablen TERM und TERMCAP, die von den termcap-Funktionen benutzt werden:

TERM: Typ des Bildschirms, mit dem termcap-Anwendungen arbeiten sollen, z.B. 97801, wird automatisch von der Shell gesetzt, wenn ein Benutzer login macht.

TERMCAP: Quelle, in der die Bildschirmbeschreibung steht:

- beginnt der Inhalt der Variablen mit '/' wird er als Pfadname für die termcap-Datei interpretiert und die Datei eröffnet,
- stimmt der Inhalt von TERM mit dem Anfang des Inhalts dieser Variablen überein, wird die nachfolgende Zeichenreihe als termcap-Beschreibung interpretiert,
- ist der Inhalt nicht definiert, wird standardmäßig aus /etc/termcap gelesen.

DIE DATEI /etc/termcap

Die Datei /etc/termcap enthält Einträge für verschiedene Bildschirmtypen.

Was ist ein EINTRAG?

Ein Eintrag ist gekennzeichnet durch eine Folge von Namen und beschreibt die einzelnen Funktionen und Möglichkeiten des Bildschirms in Form von Feldern, formal:

<kennzeichnung>:<feld>...

Ein Eintrag für einen Bildschirm darf aus höchsten 1024 Zeichen bestehen. Geht ein Eintrag über mehrere Zeilen muß das Zeilenende jeweils durch '\' entwertet werden.

Beispiel

Als Beispiel eines solchen Eintrages sehen Sie hier den Standard-Eintrag aus /etc/termcap für den Siemens SINIX-Pc:

```
standard|97801:\
        :co#80:li#24:am:bs:bt=\E[Z:cm=\E[%i%d;%dH:nd=\E[C:up=\E[A:\
        :ce=[E[OK:cd=\E[OJ:cl=\E[H\E[2J:dl=\E[M:al=\E[L:sr=\E[T:sf=\E[S:\
        :ae=[E[2m:as=\E[m:so=\E[7m:se=\E[m:ti=\E[1;24r\E[m^O\E)w:\
        :ic=[E[@:dc=\E[P:us=\E[4m:ue=\E[m:ta=^I:cs=\E[%i%d;%dr:\
        :ku=[E[A:kd=\E[B:kr=\E[C:kl=\E[D:kh=\E[H:\
        :k0=[E[@:k1=\E[P:k2=\Eo:k3=\Ep:k4=\E[L:k5=\E[M:\
        :k6=\E\072:k7=\E9:k8=\E[T:k9=\E[S:l0=\E>:l1=\Em:l2=^D:\
        :F1=\E :F2=\E;:F3=\E":F4=\E#:F5=\E$:F6=\E%:F7=\E&:F8=\E':F9=\E<:Fa=\E=:\
        :P1=\E@:P2=\EA:P3=\EB:P4=\EC:P5=\ED:P6=\EF:P7=\EG:P8=\EH:P9=\EI:Pa=\EJ:\
        :Pb=\EK:Pc=\EL:Pd=\EM:Pe=\EN:Pf=\EO:Pg=\EP:Ph=\EO:Pi=\E_:Pj=\Ed:Pk=\ET:\
        :y0=^NB^O:y1=^NC^O:y2=^ND^O:y3=^NE^O:y4=^NA^O:y5=^N@^O:y6=\E[2;7m:\
        :y7=\E[m:ya=\012:yb=\177:yc=\015:yd=^R:ye=^X:yf=^H:Pl=\Eg:GS=^N:\
        :GE=^O:GV=«:GH=A:
```

Wie wird ein Eintrag gekennzeichnet?

Jeder Eintrag beginnt mit mehreren Namen, getrennt durch '|'.
In SINIX bezeichnet der erste Name den gesamten Eintrag (im Beispiel: standard) im Hinblick auf seine häufigste Anwendung.
Der zweite Name ist die Typbezeichnung des Bildschirms (im Beispiel: 97801) Es können noch weitere Namen angegeben werden.

Was sind FELDER?

Ein Feld beschreibt eine Funktion oder Möglichkeit des Bildschirms.
Jedes Feld beginnt mit einem zwei Zeichen langen Feldnamen.
Ein Feldeintrag wird mit ':' abgeschlossen.

Es gibt drei Arten von Feldern:

Typ 1 : Boolsche Felder
<feldname>
sie zeigen das Vorhandensein bestimmter Funktionen an, z.B. bs bedeutet, daß auf dem Bildschirm durch das Backspace-Zeichen (in C '\b') eine Position nach links gegangen wird

Typ 2 : Numerische Felder
<feldname> # <zahl>
sie geben die Größe eines Bereichs an, z.B. li # 24 bedeutet, daß der Bildschirm 24 Zeilen hat

Typ 3 : Zuweisungsfelder
<feldname> = [wartezeit] <steuerzeichenfolge>
sie definieren Steuerzeichenfolgen für die Bildschirmansteuerung, und für spezielle Tastenfunktionen, z.B. dl = \E[M bedeutet, daß \E[M die Steuerzeichenfolge für "Lösche Zeile" ist.

Wartezeiten

Bei der Verwendung von Steuerzeichenfolgen kann nach dem '=' optional eine Wartezeit in Millisekunden angegeben werden, die durch Aussenden von Füllzeichen im Anschluß an die Zeichenfolge erreicht wird.
Eine Wartezeit-Angabe hat die Form:
<zahl>, z.B. 30 oder
<zahl>.<zahl>*, z.B. 4.5* für Anzahl der Millisekunden/betroffener Einheit.

Die SINIX-PC's benötigen keine Wartezeiten nach komplexen Funktionen.

Was Sie über Steuerzeichenfolgen wissen müssen

Wenn in einer Steuerzeichenfolge nicht druckbare Zeichen vorkommen, müssen diese kodiert werden.

Nicht abdruckbare Zeichen

Einige nicht druckbare Zeichen haben einen symbolischen Code:

Code	Bedeutung
\E	Escape-Zeichen
^*x*	Control-*x*
\n	Neue Zeile
\r	Wagenrücklauf
\t	Tabulator
\b	Backspace
\f	Seitenvorschub

Wegen ihrer besonderen Bedeutung im termcap-Eintrag, müssen die Zeichen ':' (Trenner), '^' (Control), '\' (Kodierung von Sonderzeichen) und '\0' wie folgt kodiert werden:

Code	Bedeutung
\^	^
\\	\
\072	:
\200	binäre Null

Außer den oben aufgeführten Spezialkodierungen, kann jedes Zeichen durch seinen ASCII-Oktalcode angegeben werden, in der Form
\ddd, d Oktalziffer

Steuerzeichenfolgen können auf der Tastatur druch gleichzeitiges Drücken der Taste [CTRL] und einer weiteren Taste erzeugt werden. Sie können ein Steuerzeichen dementsprechend auch durch ^*x* darstellen, z.B. ^D für die Taste [END].

Formate für Cursor-Positionierung

Steuerzeichenfolgen für die absolute Cursorpositionierung müssen in der Regel mit aktuellen Werten für Zeile und Spalte versorgt werden, bevor sie an den Bildschirm geschickt werden.
Im termcap-Eintrag stehen anstelle der aktuellen Werte Formatparameter, die später in einer Anwendung mit Hilfe der Funktion tgoto ersetzt werden können. Die Formatangabe ist ähnlich wie das Format in der Standardausgabefunktion printf. Im Einzelnen gibt es folgende Möglichkeiten:

Format	Bedeutung
%%	für %
%d	wie in printf (ganze Zahl)
%2	wie %2d in printf (zweistellige ganze Zahl)
%3	wie %3d in printf
%.	wie %c in printf (ein Zeichen)
%+*x*	addiert *x* zum aktuellen Wert, dann wie %.

Die folgenden Formate können zusätzlich zu den o.g. eingetragen werden, um die Umwandlung zu steuern, bewirken aber keine Ausgabe:

Format	Bedeutung
%>xy	falls der aktuelle Wert größer als *x*, addiere *y*
%r	vertausche die Reihenfolge von Zeilen- und Spaltenangabe
%i	erhöhe den aktuellen Zeilen- und Spaltenwert um 1
%n	exclusiv-ODER vom aktuellen Zeilen- und Spaltenwert mit 0140
%B	BCD-Ausgabe (16*(*x*/10)) + (*x*%10)
%D	BCD-Umgekehrte Codierung (*x* - 2*(*x*%16)

Beispiel

Das parametrisierte Zuweisungsfeld für die absolute Cursorpositionierung lautet im Eintrag für den SINIX-Pc (s.o.):

cm=\E[%i%d;%dH

Der Aufruf

tgoto("cm",4,5)

bewirkt, daß der Cursor in die 5.Zeile auf die 6.Spalte positioniert wird.

Mögliche Felder für einen termcap-Eintrag

Name	Typ	Bedeutung
ae	3	Ausschalten des alternativen Zeichensatzes
al	3	Einfügen einer Leerzeile über der Cursor-position
am	1	Cursor springt beim Erreichen des Zeilenendes auf nächste Zeile
as	3	Einschalten des alternativen Zeichensatzes
bc	3	Backspace, wenn nicht durch ^ H realisiert
BE	3	Steuerzeichenfolge für Klingel
bs	1	Backspace ist durch ^H realisiert
BS	3	Code der Backspace-Taste
bt	3	Rückwärts-Tabulator-Zeichen
bw	1	Backspace bei Zeilenwechsel (von der 1.Spalte in Zeile n zur letzten Spalte in Zeile n-1)
CC	3	Befehlszeichen für einen einzurichtenden Bildschirm
cd	3	Löschen ab Cursorposition bis Bildschirmende
ce	3	Löschen ab Cursorposition bis Zeilenende
CF	3	Cursor unsichtbar
ch	3	wie cm (Cursor bewegen) aber nur horizontal
CL	3	Code der Zeichen-nach-links-Taste
cl	3	Löschen Bildschirm
cm	3	Cursor positionieren
CN	3	Code der CANCEL-Taste
co	2	Anzahl Spalten pro Zeile
CO	3	Cursor sichtbar
CR	3	Code der Zeichen-nach-rechts-Taste
cr	3	Wagenrücklauf (Voreinstellung: ^M)
cs	3	Einstellen Scroll-Bereich (für VT100)
cv	3	wie cm (Cursor positionieren), aber nur vertikal
CW	3	Code der CHANGE WINDOW-Taste
da	1	Der Bildschirmspeicher enthält mehr als einen Bildschirm, der Inhalt wird beim Scrollen nach unten ausgegeben
db	1	wie da, aber Text wird beim Scrollen nach oben ausgegeben
dB	2	Angabe in Millisekunden für Backspace-Verzögerung
dC	2	Angabe in Millisekunden für Wagenrücklauf-Verzögerung
dc	3	Löschen Zeichen
dF	2	Angabe in Millisekunden für Seitenvorschub-Verzögerung

Name	Typ	Bedeutung
DK	3	Code der Cursor-nach-unten-Taste, falls kein Feld kd
DL	3	Code der Taste [DEL]
dl	3	Löschen Zeile
dm	3	Einschalten Lösch-Modus
dN	2	Angabe in Millisekunden für Zeilenvorschub-Verzögerung
do	3	Cursor eine Zeile nach unten
dT	2	Angabe in Millisekunden für Tabulator-Verzögerung
ed	3	Ausschalten Lösch-Modus
EE	3	Ausschalten des Editier-Modus
EG	2	Anzahl Zeichen für ES und EE
ei	3	Ausschalten des Einfüge-Modus; (falls ic angegeben, :ei=:)
EN	3	Code der Taste [END]
eo	3	Überschreibungen (siehe os) können mit Leerzeichen gelöscht werden
ES	3	Einschalten Editier-Modus
ff	3	Steuerzeichenfolge für Seitenvorschub bei Hardcopy-Geräten (Voreinstellung ^L)
G1	3	rechte obere Ecke eines Quadrates
G2	3	linke obere Ecke eines Qudrates
G3	3	linke untere Ecke eines Qudrates
G4	3	rechte untere Ecke eines Qudrates
GD	3	Pfeilspitzen-Zeichen nach unten
GE	3	Ausschalten Graphik-Modus
GG	2	Anzahl Zeichen für GS und GE
GH	3	horizontaler Strich
GS	3	Einschalten Graphik-Modus
GU	3	Pfeilspitzen-Zeichen nach oben
GV	3	vertikaler Strich
hc	1	Hardcopy-Gerät
hd	3	halbe Zeile nach unten
HM	3	Code der HOME-Taste
ho	3	Cursor an Bildschirmanfang, falls kein cm Feld
HP	3	Code der Taste [HELP]
hu	3	halbe Zeile nach oben
hz	3	Ersatzzeichenfolge für Bildschirme, die keine Tilde ausgeben können
ic	3	Einfügen Zeichen
if	3	Dateiname für das Feld is
im	1	Einschalten Einfüge-Modus; (falls ic, dann :im:)

Name	Typ	Bedeutung
in	1	Im Einfüge-Modus werden echte Leerzeichen weitergeschoben, NIL-Zeichen werden überschrieben
ip	3	Einfügen von Füllzeichen nach Zeichen-Einfügen
is	3	Zeichenfolge zur Initialisierung eines Bildschirmes
k0-k9	3	Code der Funktionstasten 0-9
kb	3	Code der Backspace-Taste
kd	3	Code der Cursor-nach-unten-Taste
ke	3	Ausschalten Tastenübertragungs-Modus
KF	3	Ausschalten Tastenklick
kh	3	Code der Cursor-an-Bildschirmanfang-Taste
kl	3	Code der Cursor-nach-links-Taste
kn	2	Anzahl "anderer" Tasten (siehe ko)
KO	3	Einschalten Tastenklick
ko	3	Funktionstasten, die bereits durch Steuerzeichenfolge beschrieben wurden
kr	3	Code der Cursor-nach-rechts-Taste
ks	3	Einschalten Tastenübertragungs-Modus
ku	3	Code der Cursor-nach-oben-Taste
l0-l9	3	Bezeichnung der Funktionstasten, wenn sie nicht f0-f9 heißen (Tastencode siehe k0-k9)
LD	3	Code der Zeichen-Lösch-Taste
LD	3	Code der Zeilen-Einfüge-Taste
li	2	Anzahl Zeilen am Bildschirm
LK	3	Code der Cursor-nach-links-Taste, wenn kein Feld kl
ll	3	letzte Zeile, erste Spalte, wenn kein Feld cm
ma	3	nur für Editor vi (Version 2), Cursor-Tasten
mi	1	Sichern der restlichen Zeichen im Einfüge-Modus
ml	3	Text über dem Cursor wird im Bildschirmspeicher "gemerkt" (der betroffene Speicherbereich wird gesperrt)
MN	3	Code der Minus-Taste
MP	3	Zeichenfolge für Multiplan-Initialisietung
MR	3	Rücksetz-Zeichenfolge für Multiplan
mu	3	Lösen der Speicher-Sperre (siehe ml)
nc	1	Wagenrücklaufn ist ersetzt durch Wagenrücklauf plus Zeilenvorschub
nd	3	Cursor-nach-rechts ohne überschreiben
nl	3	Code für Neue Zeile Voreinstellung \n
ns	1	CRT-Bildschirm, der nicht scrollen kann
NU	3	Code der NEXT-UNLOCK-CELL-Taste
os	1	mehrere Zeichen können übereinander-geschrieben werden

Name	Typ	Bedeutung
pc	3	Füllzeichen
PD	3	Code der Seite-nach-unten-Taste
PL	3	Code der Seite-nach-links-Taste
PR	3	Code der Seite-nach-rechts-Taste
PS	3	Code der Plus-Taste
pt	1	Tabulator ist hardwaremäßig realisiert
PU	3	Code der Seite-nach-oben-Taste
RC	3	Code der RECALC-Taste
RF	3	Code der TOGGLE REFERENCE-Taste
RK	3	Code der Cursor-nach-rechts-Taste, wenn kein Feld kr
RT	3	Code der Return-Taste
sb	3	Rückwärts scrollen
se	3	Ausschalten Invers-Modus
sf	3	Scrollen nach oben
sg	2	Anzahl Leerzeichen für so und se
so	3	Einschalten Invers-Modus
ta	3	Tabulator
TB	3	Code der Tabulator-Taste
tc	3	ähnlicher Bildschirm; muß letztes Feld sein
te	3	Ausschalten Cursorpositionierungs-Modus
ti	3	Einschalten Cursorpositionierungs-Modus
uc	3	Zeichenfolge zum Unterstreichen eines Zeichens mit Cursor 1 Stelle nach rechts
ue	3	Ausschalten Unterstreichungs-Modus
ug	2	Anzahl Leerzeichen für us und ue
UK	3	Code der Cursor-nach-oben-Taste, wenn kein Feld ku
ul	1	Bildschirm kann Zeichen unterstreichen
up	3	Cursor in der gleichen Spalte eine Zeile nach oben
us	3	Einschalten Unterstreichungs-Modus
vb	3	sichtbares Klingel-Zeichen
ve	3	nur für den Editor ex; ausschalten des visual-Modus
vs	3	nur für den Editor ex; einschalten des visual-Modus
WL	3	Code der Wort-nach-links-Taste
WR	3	Code der Wort-nach-rechts-Taste
xb	1	auf f1 ist escape, auf f2 ist ^C
xn	1	nur falls am, Neue Zeile wird bei automatischem Zeilenwechsel ignoriert
xr	1	Return funktioniert wie cerrnn
xt	1	Tabulatoren mit Leerzeichen überschreiben

Eintrag aus /etc/termcap suchen

int tgetent(bp,name**)**
char *bp;
char *name;

Die Funktion tgetent sucht den Eintrag für die serielle Schnittstelle *name* und schreibt ihn, falls gefunden, in den Speicherbereich, auf den *bp* zeigt. Beim Suchen wird die Umgebungsvariable TERMCAP überprüft. Wenn diese mit einem '/' beginnt, wird ihr Inhalt als Pfadnamen interpretiert und die Datei mit diesem Namen als Termcap-Datei geöffnet.
Wenn der Inhalt der Variable TERM (z.Bsp.:97801) gleich dem Anfang der Variable TERMCAP ist, wird der Inhalt von TERMCAP als Termcap-Eintrag angesehen.
Wenn TERMCAP leer ist oder einen ungültigen Inhalt hat, liest tgetent standardmäßig aus der Datei /etc/termcap.

Typ

C-Funktion

Parameter

← char *bp	Zeiger auf Speicherbereich, in den der termcap-Eintrag eingelesen wird. Der Bereich sollte mindestens 1024 Bytes groß sein. tgetent speichert die Adresse dieses Bereichs, sodaß bei späteren Aufrufen von anderen termcap-Funktionen die Adresse nicht mehr angegeben werden muß. Nicht druckbare Zeichen werden erst bei diesen späteren Aufrufen umgesetzt.
char *name	Name des einzulesenden Eintrags.

Ergebnis

1	Eintrag konnte nach *bp* gelesen werden und alles verlief ohne Fehler.
0	Angegebene Datei existiert zwar, aber es existiert dort kein Eintrag für *name*. Deshalb wird auch hier nichts eingelesen.
-1	Es wurde nichts getan, da die termcap-Datei nicht geöffnet werden kann.

Hinweis

Wenn Sie termcap-Funktionen verwenden, müssen Sie die Bibliothek /usr/lib/libtermcap.a dazubinden, etwa durch:

cc progname -ltermcap

Beispiel

```
#include <stdio.h>

main()
{
  int ret;
  char tcbuffer[1024];

  if (( ret = tgetent(tcbuffer,getenv("TERM")))<= 0)
        { printf("termcap: Fehler beim Einlesen, Rueckgabewert:%d\n",ret);
          exit(1);   }
  else
        { printf("Es wurde eingelesen:\n");
          printf("%s\n",tcbuffer);   }
 }
```

Dateien

/etc/termcap
Termcap-Datei

/usr/lib/libtermcap
Bibliothek der Termcap-Funktionen

> > > > tgetnum, tgetstr, tgetflag, tputs, tgoto

Boolsches Feld im Eintrag suchen

int tgetflag(id)
char *id;

tgetflag sucht das Feld mit Namen *id* vom Typ 1 (Boolsches Feld) in dem Eintrag, der von tgetent eingelesen wurde. tgetflag gibt mit 1 oder 0 an, ob es das gesuchte Feld gefunden hat oder nicht.

Typ

C-Funktion

Parameter

char *id	Name eines Feldes

Ergebnis

1	Wenn das Feld im Eintrag vorhanden ist.
0	Wenn das Feld nicht im Eintrag vorhanden ist.

Hinweis

- Wenn Sie termcap-Funktionen verwenden, müssen Sie die Bibliothek /usr/lib/libtermcap.a dazubinden, etwa durch:

  cc progname -ltermcap

- Die Funktion tgetflag kann nur ohne Fehler aufgerufen werden, wenn vorher die Funktion tgetent aufgerufen wurde, damit tgetflag einen Eintrag zur Verfügung hat.

Beispiel

```
#include <stdio.h>

main()
{
  char buff[1024];

        tgetent(buff,getenv("TERM"));

        if ( tgetflag("bs") == 1 )
                printf("Terminal kennt Backspacezeichen\n");
        else
            printf("Terminal kennt kein Backspacezeichen\n");

}
```

Dateien

/etc/termcap
: Termcap-Datei

/usr/lib/libtermcap
: Bibliothek der Termcap-Funktionen

> > > > tgetent, tgetnum, tgetstr, tputs, tgoto

Numerisches Feld im Eintrag suchen

int tgetnum(id)
char *id;

tgetnum sucht das Feld mit Namen *id* vom Typ 2 (numerisches Feld) in dem Eintrag, der von tgetent eingelesen wurde. Wenn tgetnum das Feld findet, gibt es den entsprechenden Wert aus.

Typ

C-Funktion

Parameter

char *id	Zeiger auf Zeichenreihe, die die Bezeichnung eines Feldes enthält.

Ergebnis

Wert von *id* als Integer
Bei Erfolg.

-1 Falls das Feld nicht existiert oder leer ist.

Hinweis

- Wenn Sie termcap-Funktionen verwenden, müssen Sie die Bibliothek /usr/lib/libtermcap.a dazubinden, etwa durch:

  cc progname -ltermcap

- Die Funktion tgetnum kann nur aufgerufen werden, wenn vorher die Funktion tgetent aufgerufen wurde, damit tgetnum einen Eintrag zur Verfügung hat.

Beispiel

```
#include <stdio.h>

main()
{
   int lines;
   char tcbuffer[1024];

   tgetent(tcbuffer,getenv("TERM"));

   if ((lines = tgetnum("li")) == -1)
        { printf("termcap: Zeilenzahl nicht angegeben\n");
          exit(1);      }
   else
        { printf("Zeilenanzahl ist: %d\n",lines); }
}
```

Dateien

/etc/termcap
: Termcap-Datei

/usr/lib/libtermcap
: Bibliothek der Termcap-Funktionen

> > > > tgetent, tgetstr, tgetflag, tputs, tgoto

Steuerzeichenfolge im Eintrag suchen

char *tgetstr(id, area**)**
char *id;
char **area;

tgetstr sucht das Feld mit Namen *id* vom Typ 3 (Zuweisungsfeld) in dem Eintrag, der von tgetent eingelesen wird. Wenn tgetstr das Feld findet, wird der Inhalt an der Adresse abgelegt, die über *area* erreicht wird.

Typ

C-Funktion

Parameter

char *id — Zeiger auf Zeichenreihe, die die Bezeichnung eines Feldes enthält.

← char **area

Speicherbereich, in dem der Inhalt des gefundenen Feldes abgelegt wird.
Hier wird bewußt kein einfacher Zeiger benutzt (siehe Hinweis).

Ergebnis

Adresse des Speicherbereichs, an dem die gefundene Zeichenreihe abgelegt wurde.
Bei Erfolg

Nullzeiger — Wenn das Feld nicht existiert oder leer ist.

Hinweis

- Wenn Sie termcap-Funktionen verwenden, müssen Sie die Bibliothek /usr/lib/libtermcap.a dazubinden, etwa durch:

  cc progname -ltermcap

- Die Funktion tgetstr kann nur ohne Fehler aufgerufen werden, wenn vorher die Funktion tgetent aufgerufen wurde, damit tgetstr einen Eintrag zur Verfügung hat.

- Mit *area* erhält tgetstr nicht direkt die Adresse des Speicherbereichs für das Ergebnis, sondern einen Zeiger auf diese Adresse. Das liegt daran, daß tgetstr nach Ablegen eines gefundenen Feldes den Zeiger um die Länge des eingetragenen Inhalts erhöht. Der Zeiger zeigt dann also wieder auf den nächsten freien Speicherplatz. So kann man tgetstr mehrfach mit demselben *area* aufrufen, ohne das Ergebnis eines vorangegangenen Aufrufs zu überschreiben.

Beispiel

```
#include <stdio.h>
char *tgetstr();

main()
{
     char buf[1024];
     char buff[1024];
     char *bp, *zei;

      tgetent(buff,getenv("TERM"));

               /* Stellt Eintrag fuer tgetstr zur
                  Verfuegung */

      bp = buf;
      zei = tgetstr("cm",&bp);
      printf("Der Zeiger nach dem ersten Aufruf ist: %u\n",zei);

        if ( zei == NULL )
           {
            printf("Gesuchte Faehigkeit unmoeglich\n");
            exit(1);
           }
       else
          {
           printf("Der Inhalt ist: %s\n",zei);
          }
               /* Gibt Inhalt des gesuchten Eintrags aus.
                  Hier 'cm' gesucht */

               /* Nächste Steuerzeichenfolge */
        zei=tgetstr("bt",&bp);
        printf("Der Zeiger nach dem zweiten Aufruf ist:  %u\n",zei);
}
```

Dateien

/etc/termcap
: Termcap-Datei

/usr/lib/libtermcap
: Bibliothek der Termcap-Funktionen

> > > > tgetent, tgetnum, tgetflag, tputs, tgoto

Steuerzeichenfolge mit aktuellen Werten versorgen

extern char *UP;
extern char *BC;

char *tgoto(cm,spalte,zeile**)**
char *cm;
int spalte, zeile;

tgoto versorgt die parametrisierte Steuerzeichenfolge *cm* zur Cursor-Bewegung mit den aktuellen Werten *spalte* und *zeile*. tgoto benutzt die externen Variablen UP und BC, falls die Ausgabe von '\n', '^D' und '\000' verhindert werden soll, weil diese gesondert behandelt werden.
Das aufrufende Programm muß dazu den externen Variablen UP und BC die entsprechenden Steuerzeichenfolgen für up und bc aus der termcap-Datei zuweisen.

Typ

C-Funktion

Parameter

char *cm	parametrisierte Steuerzeichenfolge für die Cursor-Bewegung, z.B. cm=\E[%i%d;%dH
int spalte	aktueller Wert für die Spalten-Positionierung
int zeile	aktueller Wert für die Zeilen-Positionierung

Ergebnis

Zeiger auf die aktualisierte Steuerzeichenfolge	bei Erfolg
"00PS"	falls tgoto eine Formatangabe (%...) nicht interpretieren konnte

Achtung

Bevor Sie tgoto aufrufen, sollten Sie den Modus "Tabulator expandieren" (XTABS-Bits in der Struktur sggtyb löschen, siehe gtty, stty) ausschalten, da tgoto, falls möglich, ein Tabulatorzeichen zur Positionierung verwendet.

Hinweis

Wenn Sie termcap-Funktionen verwenden, müssen Sie die Bibliothek /usr/lib/libtermcap.a dazubinden, etwa durch

cc progname -ltermcap.

Beispiel

siehe Beispiel bei tputs

Externe Größen

char *UP
: externe Variable, der vom aufrufenden Programm die Zeichenfolge für up (Cursor in der gleichen Spalte eine Zeile nach oben) zugewiesen werden muß.

extern char *BC
: externe Variable, der vom aufrufenden Programm die Zeichenfolge für bc (Backspace) zugewiesen werden muß.

Dateien

/etc/termcap
: Termcap-Datei, enthält insbesondere die Einträge für cm, up und bc

/usr/lib/libtermcap.a
: Bibliothek der termcap-Funktionen

> > > > tgetent, tgetnum, tgetflag, tgetstr, tputs

Ausgabe von Steuerzeichenfolgen

extern char PC;
extern short ospeed;

int tputs(cp, anz, ausg**)**
char *cp;
int anz;
int (*ausg**) ()**;

tputs gibt die Steuerzeichenfolge cp aus. Dabei wertet tputs Wartezeitangaben am Anfang von Steuerzeichenfolgen aus und übergibt die nötige Anzahl von Füllzeichen.

Typ

C-Funktion

Parameter

char *cp — Steuerzeichenfolge, die ausgegeben werden soll, z.B. das Ergebnis von tgoto (siehe Beispiel)

int anz — Anzahl der Zeilen, die von der Operation betroffen sind

int (*ausg)() — Funktion zur zeichenweise Ausgabe, die Ausgabe von tputs wird zeichenweise an *ausg* übergeben.

Hinweis

- Zur Berechnung der Anzahl von Füllzeichen, die erforderlich ist, um die gewünschte Wartezeit zu erreichen, muß in der externen Variablen ospeed die Übertragungsgeschwindigkeit zum angeschlossenen Bildschirm eingetragen sein. Sie erhalten diesen Wert aus der Struktur sgttyb, die durch die Funktion gtty aufgefüllt wird (siehe gtty, stty).
- Wenn Sie tputs verwenden, müssen Sie die Bibliothek /usr/lib/libtermcap.a dazubinden, durch cc progname -ltermcap.

Beispiel

```
#include <stdio.h>
#include <sgtty.h>

 extern short ospeed;
 extern char *UP;
 extern char *BC;

 char *tgoto(), *tgetstr();
 char *cm;

 outc(c)
 char c;
 {
    putchar(c);
 }

 main()
 {
        struct sgttyb ttymodes;

        char puff[1024];
        char puf[1024];
        char *bp, *zei;

        tgetent(puff,getenv("TERM"));

        bp = puf;
              /* Versorung der externen Variablen */
              /* Übertragungsgeschwindigkeit */
        gtty(1, &ttymodes);
        ospeed = ttymodes.sg_ospeed;
              /* UP und BC besetzen */
        UP = tgetstr("up",&bp);
        if (!tgetflag("bs"))
            BC = tgetstr("bc",&bp);

              /* parametrisierte Steuerzeichenfolge
                 für Cursor-Bewegung */
        zei = tgetstr("cm",&bp);

              /* Ausgabe der aktualisierten
                 Steuerzeichenfolge */
        tputs(tgoto(zei,35,3),3,outc);
 }
```

Externe Größen

extern char PC
: externe Variable, in der das Füllzeichen abgespeichert wird, falls es ungleich '\0' ist.

extern short ospeed
: externe Variable, in der die Übertragungsgeschwindigkeit des Bildschirmes eingetragen wird (siehe gtty, stty)

Dateien

/etc/termcap
: Termcap-Datei

/usr/lib/libtermcap.a
: Bibliothek der Termcap-Funktionen

> > > > tgetent, tgetflag, tgetnum, tgetstr, tgoto

Aktuelle Zeitangabe

long time(sek _ zg**)**
long *sek _ zg;

time liefert die aktuelle Zeit (GMT) seit dem 1.Januar 1970 00:00:00 in Sekunden.

Typ

Systemaufruf

Parameter

← long *sek _ zg
Zeiger, auf das von time gelieferte Ergebnis
Sie haben folgende Möglichkeiten:

0L — der Parameter *sek _ zg* hat keine Bedeutung

ungleich Nullzeiger
das Ergebnis von time wird zusätzlich in den Bereich gespeichert, auf den *sek _ zg* zeigt

Ergebnis

Zeit in Sekunden seit dem 1.1.1970 00:00:00
die Zeit, die time liefert, kann mit stime eingestellt werden.

Beispiel

Aktuelle Zeit (GMT) und Ortszeit (MEZ):

```
#include <stdio.h>

char *asctime();
struct tm *gmtime();
char *ctime();
long time();
long a;

main()
{
  a=time(0L);
  printf("Sekunden : %ld\n",a);
  printf("Ortszeit : %s",ctime(&a));
  printf("GMT      : %s",asctime(gmtime(&a)));
}
```

> > > > ftime, stime, date(Kommando)

Laufzeit eines Prozesses

#include <sys/times.h>

int times(zeiten)
struct tbuffer *zeiten;

times mißt in 1/{HZ} Sekunden die Laufzeiten des aufrufenden Prozesses und aller beendeter Sohnprozesse, auf die der Prozeß gewartet hat (wait).

Typ

Systemaufruf

Parameter

← struct tbuffer *zeiten

Zeiger auf eine Struktur, in die times sein Ergebnis ablegt. Die Struktur *tbuffer* ist in <sys/times.h> wie folgt definiert:

```
struct tbuffer {
        long proc_user_time;
        long proc_system_time;
        long child_user_time;
        long child_system_time;
};
```

Ergebnis

Zeit seit Systemstart in 1/{HZ} Sekunden
bei Erfolg

-1 bei Fehler

Hinweis

Die Komponenten der Struktur tbuffer geben folgende Prozeßzeiten an (in 1/{HZ} Sekunden):

```
proc_user_time    : Benutzerzeit
proc_system_time  : Systemzeit
chid_user_time    : Summe der Benutzerzeiten von allen Sohnprozes-
sen
child_system_time: Summe der Systemzeiten von allen Sohnprozessen
```

Beispiel

Laufzeiten des aktuellen Prozesses auflisten:

```
#include <stdio.h>
#include <sys/times.h>

struct tbuffer buffer;

main()
{
   times(&buffer);
   printf("Benutzerzeit : %ld\n",buffer.proc_user_time);
   printf("Systemzeit : %ld\n",buffer.proc_system_time);
}
```

Dateien

/usr/include/sys/times.h
Definition der Struktur tbuffer

> > > > time(Kommando)

Laufzeit eines Prozesses

#include <sys/times.h>
#include <sys/types.h>

int times(zeiten**)**
struct tms *zeiten;

times mißt in 1/{HZ} Sekunden die Laufzeiten des aufrufenden Prozesses und aller beendeter Sohnprozesse, auf die der Prozeß gewartet hat (wait).

Typ

Systemaufruf

Parameter

← struct tms *zeiten

Zeiger auf eine Struktur, in die times sein Ergebnis ablegt. Die Struktur tms ist in <sys/times.h> wie folgt definiert:

```
struct tms {
        time-t tms_utime;
        time_t tms_stime;
        time_t tms_cutime;
        time_t tms_cstime;
};
```

Ergebnis

Zeit seit Systemstart in 1/{HZ} Sekunden
bei Erfolg

-1 bei Fehler

Hinweis

Die Komponenten der Struktur tms geben folgende Prozeßzeiten an (in 1/{HZ} Sekunden):

```
tms_utime  : Benutzerzeit
tms_stime  : Systemzeit
tms_cutime : Summe der Benutzerzeiten von allen Sohnprozessen
tms_cstime : Summe der Systemzeiten von allen Sohnprozessen
```

Der Typ time_t ist in <sys/types.h> definiert.

Beispiel

Laufzeiten des aktuellen Prozesses auflisten:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>

struct tms buffer;

main()
{
   times(&buffer);
   printf("Benutzerzeit : %ld\n",buffer.tms_utime);
   printf("Systemzeit : %ld\n",buffer.tms_stime);
}
```

Dateien

/usr/include/sys/times.h
: Definition der Struktur tms

/usr/include/sys/types.h
: Definition des Typs time_t

/usr/include/sys/param.h
: Definition der Zeitauflösung {HZ}

> > > > time(Kommando), exec, fork, time, wait

Name der Zeitzone

char *timezone(zone,sz**)**
int zone, sz;

timezone liefert den Namen der Zeitzone, die Sie in *zone* durch Minuten westlich von Greenwich angeben.

Typ

C-Funktion

Parameter

int zone	Minuten westlich von Greenwich
int sz	Flag für Sommerzeit (nicht unterstützt)

immer 0

Ergebnis

Name der Zeitzone, die *zone* Minuten westlich von Greenwich liegt

Achtung

timezone schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

Wenn timezone für eine Abweichung *zone* keinen entsprechenden Eintrag in seiner Namentabelle findet, liefert es die Ausgabe:

GMT+zone

Beispiel

Aktuelles Datum für Mitteleuropäische Zeit (MEZ):

```
#include <stdio.h>

long time();
char *ctime();
long clock;
char *timezone();

main()
{
 clock = time(0L);
 printf("%s : %s\n",timezone(-60,0),ctime(&clock));
}
```

> > > > localtime, ftime

Zeichen in ASCII-Zeichen umwandeln

#include <ctype.h>

int toascii(c)
char c;

toascii wandelt das Zeichen *c* in ein ASCII-Zeichen um.

Typ

Makro

Parameter

char c	Zeichen

Ergebnis

c & 0177	ASCII-Zeichen

Beispiel

Das folgende Programm liest einen ganzzahligen Wert ein. Der Wert wird dann - ins Oktalsystem übertragen - einmal unverändert und einmal nach dem toascii Aufruf ausgegeben.

```
#include <stdio.h>
#include <ctype.h>
main()
{
    int wert;
    scanf("%d",&wert);
    printf("Vorher in oktaler Schreibweise : %o \n",wert);
    printf("Nachher in oktaler Schreibweise : %o \n",toascii(wert));
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenbehandlung

> > > > toupper, tolower

Großbuchstabe in Kleinbuchstabe umwandeln

#include <ctype.h>

int tolower(c)
char c;

tolower wandelt den Großbuchstaben *c* in den entsprechenden Kleinbuchstaben um.

Typ

Makro

Parameter

char c	Großbuchstabe

Ergebnis

c - 'A' + 'a'
der entsprechende Kleinbuchstabe zu *c*

Achtung

In CES-Version 1.0B wird nicht abgeprüft, ob *c* ein Großbuchstabe ist.

Beispiel

Das folgende Programm liest eine Zeichenreihe ein, und wandelt die Zeichen zunächst in Kleinbuchstaben und dann in Großbuchstaben um.

```
#include <ctype.h>
main()
{
    int i;
    char s[81];
    printf("Bitte geben Sie eine Zeichenreihe (max.80 Zeichen) ein \n");
    scanf("%s",&s);
    printf("Und jetzt alles in Kleinbuchstaben \n");
    i = 0;
    while( s[i] != '\0' )
         if (isupper(s[i]))
              printf("%c",tolower(s[i++]));
         else printf("%c",s[i++]);
    printf("\n Und in Großbuchstaben \n");
    i = 0;
    while( s[i] != '\0' )
         if (islower(s[i]))
              printf("%c",toupper(s[i++]));
         else printf("%c",s[i++]);
    printf("\n");
}
```

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenbehandlung

> > > > toupper, toascii

Kleinbuchstabe in Großbuchstabe umwandeln

#include <ctype.h>

int toupper(c)
char c;

toupper wandelt den Kleinbuchstaben *c* in den entsprechenden Großbuchstaben um.

Typ

Makro

Parameter

char c	Kleinbuchstabe

Ergebnis

c - 'a' + 'A'
der entsprechende Großbuchstabe zu *c*

Achtung

In CES-Version 1.0B wird nicht abgeprüft, ob *c* ein Kleinbuchstabe ist!

Beispiel

siehe Beispiel bei tolower

Dateien

/usr/include/ctype.h
Definition der Makros zur Zeichenbehandlung

>>>> tolower, toascii

Name einer Gerätedatei

char *ttyname(dk**)**
int dk;

ttyname liefert den vollen Pfadnamen der Gerätedatei mit Dateikennzahl *dk*.

Typ

C-Funktion

Parameter

int dk	Dateikennzahl

Ergebnis

voller Pfadname der Gerätedatei
falls *dk* einer Gerätedatei zugeordnet ist

Nullzeiger	falls *dk* keiner Gerätedatei im Dateiverzeichnis /dev zugeordnet ist

Achtung

ttyname schreibt sein Ergebnis in einen statischen Datenbereich, der bei jedem Aufruf überschrieben wird!

Hinweis

ttyname funktioniert für jede Gerätedatei im Dateiverzeichnis /dev und nicht nur für Gerätedateien, die einer Datensichtstation zugeordnet sind!

Beispiel

Folgendes Programm gibt den Namen der Datensichtstation aus, die dem Prozeß zugeordnet ist, überprüft, ob Standardeingabe und Fehlerausgabe auf den Bildschirm gelegt sind und gibt an, in welcher Zeile der Datei /etc/ttys die Datensichtstation des Prozesses eingetragen ist:

```
#include <stdio.h>
char *ttyname();

main()
{
  printf("Dateiname für Standardausgabe : %s\n",ttyname(1));
  if(isatty(0) == 1)
    printf("Standaradeingabe mit Bildschirm verbunden\n");
  if(isatty(2) == 1)
    printf("Standardfehlerausgabe mit Bildschirm verbunden\n");
 printf("Zeilennummer in /etc/ttys : %d\n",ttyslot());
}
```

Wenn Sie die Standardausgabe umlenken (z.B. exec 1>aus) bevor Sie obiges Programm aufrufen, liefert ttyname(1) den Nullzeiger und die Ausgabe wird in die Datei *aus* geschrieben.

Dateien

/dev/*	Gerätedateien
/etc/ttys	Liste der angeschlossenen Datensichtstationen

>>>> ttyname, ttyslot, isatty, ioctl

Eintrag in /etc/ttys bestimmen

int ttyslot()

ttyslot sucht in /etc/ttys den Eintrag der Datensichtstation, die dem aufrufenden Prozeß zugeordnet ist.

Typ

C-Funktion

Parameter

keine

Ergebnis

Zeilennummer des Datensichtstationeintrags in /etc/ttys
falls dem Prozeß eine Datensichtstation zugeordnet ist

0 falls ttyslot

- keine zugeordnete Datensichtstation bestimmen kann oder
- auf /etc/ttys nicht zugreifen kann

Beispiel

siehe Beispiel bei ttyname

Dateien

/etc/ttys	Liste der angeschlossenen Datensichtstationen

> > > > ttyname, isatty, ioctl

Kontrollfunktionen auf einen Benutzerprozeß

long ulimit(akt,arg**)**
int akt;
long arg;

ulimit gibt Ihnen die Möglichkeit, mit Hilfe einer in *akt* angegebnen Aktion gewisse Prozeßschranken abzufragen oder neu zu setzen.

Typ

Systemaufruf

Parameter

int akt long arg	Sie geben hier eine Konstante an, mit der die gewünschte Aktion ausgesucht wird. *arg* ist im Fall 2 aktueller Parameter für die Aktion. Es gibt folgende Möglichkeiten:
1	die maximale Dateigröße für einen Prozeß wird in Blöken von 512 Byte zurückgegeben. Ein Sohnprozeß übernimmt diese Größe vom Vaterprozeß.
2	die maximale Dateigröße für einen Prozeß wird auf *arg* gesetzt. Nur der Systemverwalter darf den Wert vergrößern!
3	der maximale Wert für den "break" wird zurückgegeben.

Ergebnis

Bei Erfolg ist das Ergebnis eine ganze Zahl ungleich -1 und hängt von der gewählten Aktion ab:

maximale Dateigröße in Blöcken zu 512 Bytes
falls die Aktion 1 gewählt wurde

ungleich -1
falls die Aktion 2 gewählt wurde

maximale Adresse für den "break"
falls die Aktion 3 gewählt wurde

-1 Fehler, wenn

- *akt* keine gültige Aktion ist oder
  arg ein unzulässiges Argument für die ausgewählte Aktion ist oder
- die Dateigröße erhöht werden sollte, der Prozeß aber nicht unter der Kennung des Systemverwalter läuft.

Fehlermeldung

Bei Rückkehr mit Fehler wird in errno ein entsprechender Fehlercode abgelegt:

EINVAL : Unzulässiges Argument
EPERM : Hat anderen Eigentümer

Hinweis

Die Schranke für die Dateigröße gilt nur für Schreiboperationen auf normale Dateien:

- Sie dürfen Dateien beliebiger Größe lesen.
- Sie dürfen auf Magnetbänder, Disketten, usw. beliebiger Größe schreiben.

> > > > sbrk, write

Schutzbitmaske setzen

int umask(ok_zahl**)**
int ok_zahl;

umask setzt die Schutzbit-Maske, die bei creat und mknod zur Definition der Zugriffsrechte einer Datei verwendet wird. Als Ergebnis liefert umask die vorige Maskenbelegung.

Typ

Systemaufruf

Parameter

int ok_zahl

Sie geben hier eine dreistellige Oktalzahl an, die Schutzbitmaske des Prozesses (oder kurz: Prozeßmaske).
Das Komplement der Binärdarstellung von *ok_zahl* wird bei creat und mknod zur Festlegung der Zugriffsrechte einer neuen Datei verwendet:
Eine 1 in der Maske dient dazu, die entsprechende Stelle im *modus*-Parameter von creat, open oder mknod auszublenden. Die Angabe einer Oktalziffer ungleich 0 bedeutet daher, daß die entsprechende Zugriffsart nicht gestattet ist. Im einzelnen können Sie die Bedeutung der Oktalziffern aus folgender Tabelle herleiten:

Oktalzahl		Bedeutung
000		keine Beschränkung
400	Eigentümer:	nicht lesen
200		nicht schreiben
100		nicht ausführen bzw.durchsuchen
040	Gruppe:	nicht lesen
020		nicht schreiben
010		nicht ausführen bzw.durchsuchen
004	Andere:	nicht lesen
002		nicht schreiben
001		nicht ausführen bzw.durchsuchen
002		Standard: keine Schreiberlaubnis für Andere

Ergebnis

Belegung der Schutzbitmaske vor Aufruf von umask.

Hinweis

Ein Sohnprozeß übernimmt die Schutzbitmaske seines Vaterprozesses.

Beispiel

Leseerlaubnis für Eigentümer,Gruppe und Andere setzen:

```
main()
{
  int dk;
              /* Schutzbitmaske setzen */
  umask(0223);

              /*                     oktal      binär
                  Schutzbitmaske     0223       010 010 011

                  Komplement         0554       101 101 100
                & modus              0666       110 110 110
                ________________________________________________
                  Zugriffsrechte     0444       100 100 100   */

dk = creat("neu",0666);
}
```

> > > > umask(Kommando), chmod, mknod

Dateisystem abhängen

int umount(gerät**)**
char *gerät;

Nur für den Systemverwalter!
umount hängt ein Dateisystem ab, das zuvor mit mount in den aktuellen Dateibaum eingehängt wurde und stellt die ursprünglichen Verhältnisse wieder her.

Typ

Systemaufruf

Parameter

char *gerät
: Dateiname der Gerätedatei, die dem Datenträger zugeordnet ist, auf dem sich das Dateisystem befindet, z.B.

/dev/fl2
: Dateisystem ist auf Diskette

Ergebnis

0
: das Dateisystem wurde abhängt:

- noch anstehende Ein/Ausgabeoperationen wurden vorher korrekt abgeschlossen
- das Dateisystem wurde als in Ordnung markiert
- ein durch den mount Aufruf überdeckter Teilbaum wird wieder verfügbar

-1 Fehler, wenn

- die Gerätedatei *gerät* nicht existiert oder
- das entsprechende Gerät kein Dateisystem enthält, das eingehängt wurde oder
- Dateien in dem eingehängten Dateisystem noch benützt werden oder
- der Prozeß nicht unter der Kennung des Systemverwalters läuft oder
- *gerät* kein blockorientiertes Gerät ist oder
- eine Komponente des Pfadnamens kein Dateiverzeichnis ist

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

ENXIO :	Gerät oder Adresse unbekannt
EINVAL :	Unzulässiges Argument
EBUSY :	Gerät oder Dateiverzeichnis noch nicht frei
EPERM :	Hat anderen Eigentümer
ENOTBLK :	Nur bei blockorientierten Geräten möglich
ENOTDIR :	Kein Dateiverzeichnis

Beispiel

siehe Beispiel bei mount

Dateien

/etc/mtab Tabelle über alle eingehängten Dateisysteme

> > > > mount, /etc/umount, /etc/mount

Name des aktuellen SINIX Systems

#include <sys/utsname.h>

int uname(name**)**
struct utsname *name;

uname liefert den Namen des aktuellen SINIX Systems in einer Struktur utsname.

Typ

Systemaufruf

Parameter

← struct utsname *name

Zeiger auf eine Struktur, in die uname sein Ergebnis schreibt. Die Struktur ist in <sys/utsname.h> wie folgt definiert:

```
struct utsname {
        char sysname [15];
        char nodename [15];
        char release [15];
        char version [15];
        char sysid[15];
        char language[15];
};
```

Ergebnis

nicht negativ

uname hat den Systemnamen in einer Struktur utsname abgelegt

-1 Fehler, falls *name* kein gültiger Zeiger ist

Fehlermeldung

Kehrt uname mit Fehler zurück, besetzt es errno mit dem Fehlercode:

EFAULT : Unzulässige Adresse

Beispiel

Name Ihres SINIX Systems ausgeben.

```
#include <stdio.h>
#include <sys/utsname.h>
main()
{
    int h;
    struct utsname *name,utn;
    name = &utn;
    h = uname(name);
    if( h != -1)
      {
         printf("Systemname : %s\n",name->sysname);
         printf("Nodename   : %s\n",name->nodename);
         printf("Release    : %s\n",name->release);
         printf("Version    : %s\n",name->sysid);
         printf("Language   : %s\n",name->language);
      }
    else printf("Fehler");
}
```

Dateien

/usr/include/sys/utsname.h
Definition der Struktur utsname

Zeichen zurückstellen

```
#include <stdio.h>

int ungetc(c,dz)
int c;
FILE *dz;
```

ungetc schreibt das Zeichen *c* in den Puffer zurück, der der Datei mit Dateizeiger *dz* zugeordnet ist. Die nächste Leseoperation, die zeichenweise von dieser Datei einliest (getc), liefert dann nochmals *c*.

Typ

C-Funktion (s)

Parameter

int c	Zeichen, das zurückgestellt werden soll
EOF	bei Eingabe von EOF macht ungetc nichts und liefert EOF zurück
FILE *dz	Dateizeiger für die Datei, in deren Puffer das Zeichen zurückgeschrieben werden soll

Ergebnis

c	bei Erfolg liefert ungetc das zurückgestellte Zeichen
EOF	Fehler, wenn ungetc das Zeichen nicht zurückschreiben kann.

Hinweis

- Außer bei stdin muß wenigstens ein Zeichen vor dem ersten ungetc Aufruf aus der Datei gelesen worden sein.
- Der Datei muß ein Ein/Ausgabe-Puffer zugeordnet sein (fopen, setbuf).
- EOF kann nicht zurückgestellt werden.
- Es kann höchstens ein Zeichen zurückgestellt werden.
- Ein fseek Aufruf löscht ein zurückgestelltes Zeichen.

Beispiel

```
#include <stdio.h>

FILE *fp;
int c;

main()
{
  fp = fopen("datei","r");
              /* es werden mehrere Zeichen von stdin einge-
                 lesen, aber nur das erste Zeichen wird in
                 den Puffer zurückgeschrieben */
  while((c=getchar()) != EOF)
        ungetc(c,fp);
  while((c=getc(fp)) != EOF)
        putc(c,stdout);
  close(fp);
}
```

Dateien

/usr/include/stdio.h
Definitionen für Standardein/ausgabe

> > > > getc, setbuf, fseek

Verweis auf eine Datei löschen

int unlink(pfad**)**
char *pfad;

unlink entfernt den Dateiverzeichniseintrag, der in *pfad* angegeben ist. Wenn alle Verweise (Dateiverzeichniseinträge) auf eine Datei gelöscht sind, wird der Platz, den die Datei belegt hatte, freigegeben, sobald kein Prozeß mehr mit dieser Datei arbeitet.
Nur der Systemverwalter darf den Verweis auf ein Dateiverzeichnis löschen!

Typ

Systemaufruf

Parameter

char *pfad
: Pfadname, der den Eintrag benennt, der gelöscht werden soll

Ergebnis

0 unlink hat den Verweis entfernt

-1 Fehler, wenn

- eine Komponente des Pfadnamens kein Dateiverzeichnis ist oder
- die in *pfad* angegebene Datei nicht existiert oder
- ein Dateiverzeichnis auf dem Pfad nicht durchsucht werden darf oder
  in dem Dateiverzeichnis, in dem der Eintrag steht, nicht geschrieben werden darf oder
- *pfad* ein Dateiverzeichnis bezeichnet und der Prozeß nicht unter der Kennung des Systemverwalters läuft oder
- *pfad* die Wurzel eines eingehängten Dateisystems bezeichnet oder
- der zu löschende Eintrag der letzte Verweis auf eine Datei ist, die nur Programmtext enthält, der gerade ausgeführt wird.

Fehlermeldung

Bei Fehler steht in errno ein entsprechender Fehlercode:

ENOTDIR : Kein Dateiverzeichnis
ENOENT : Datei oder Dateiverzeichnis unbekannt
EACCES : Zugriff untersagt
EPERM : Hat anderen Eigentümer
EBUSY : Gerät oder Dateiverzeichnis noch nicht frei
ETXBSY : Programm wird gerade ausgeführt

Beispiel

siehe Beispiel bei fread

> > > > close, link, open, rm(Kommando)

Zeiten einer Datei neu setzen

#include <sys/types.h>

int utime(d_name,zeit**)**
char *d_name;
time_t zeit[2];

Nur für Dateieigentümer oder Systemverwalter!
utime setzt die Zeiten der Datei *d_name* neu. Für jede Datei werden drei Zeiten geführt, die Sie mit dem Systemaufruf stat abfragen können:

- Zeit des letzten Zugriffs
- Zeit der letzten Änderung
- Zeit der letzten Änderung des Indexeintrages

Die neuen Zeiten für letzten Zugriff und letzte Änderung geben Sie in *zeit[0]* und *zeit[1]* an. Die Änderungszeit des Indexeintrags wird auf die aktuelle Zeit gesetzt.

Typ

Systemaufruf

Parameter

char *d_name
: Dateiname der Datei, deren Zeiten geändert werden sollen

time_t zeit[2]
: Vektor mit den zwei neuen Zeitangaben in Sekunden. Der integer Typ time_t ist in <sys/types.h> definiert.

Ergebnis

0 die Zeiten wurden erfolgreich geändert

-1 Fehler, wenn

- die Datei *d_name* nicht existiert oder
- eine Komponente auf dem Pfad zu *d_name* kein Dateiverzeichnis ist oder
- ein Dateiverzeichnis auf dem Pfad zu *d_name* nicht durchsucht werden darf oder
- die effektive Benutzernummer ist weder die Kennung des Systemverwalters noch des Dateieigentümers

Fehlermeldung

Bei Ergebnis -1 wird in errno ein entsprechender Fehlercode abgelegt:

ENOENT : Datei oder Dateiverzeichnis unbekannt
ENTODIR : Kein Dateiverzeichnis
EACCES : Zugriff untersagt
EPERM : Hat anderen Eigentümer

Beispiel

```
#include <sys/types.h>
#include <sys/stat.h>
time_t timep[2];
struct stat dinf;
char *ctime();

main()
{
  timep[0] = 1001;
  timep[1] = 2001;
  utime("datei",timep);
  stat("datei",&dinf);
  printf("%ld : %ld : %ld\n",dinf.st_atime,dinf.st_mtime,dinf.st_ctime);
  printf("%s",ctime(&dinf.st_ctime));
}
```

Dateien

/usr/include/sys/types.h
Definition des Typs time_t

> > > > stat, ls(Kommando)

Auf Prozeßbeendigung warten

int wait(stat_zg**)**
int *stat_zg;

wait hält den aufrufenden Prozeß an bis ein direkter Sohnprozeß normal terminiert, durch ein Signal abgebrochen wird oder während einem Trace-Vorgang an einem Haltepunkt stoppt.
Falls der Sohnprozeß allerdings bereits vor dem wait Aufruf im Vater beendet oder gestoppt ist, kehrt wait sofort zurück, d.h. der Prozeß wird dann nicht mehr angehalten.

Typ

Systemaufruf

Parameter

← int *stat_zg

Zeiger auf eine ganze Zahl

Nullzeiger — der Parameter *stat_zg* hat keine Bedeutung

ungleich Nullzeiger

die beiden niedrigeren Bytes der ganzen Zahl, auf die *stat_zg* zeigt, informieren darüber, wie und warum der Sohnprozeß zu Ende ging:

	exit-Status	Abbruchursache
15	7	0

Bild 2-11 Aufteilung der Information

Die Bits sind im Einzelnen wie folgt definiert:

- im höheren Byte steht der exit-Status des Sohnprozesses, falls dieser normal terminiert hat (siehe exit)
- im niedrigeren Byte steht Information über die Ursache der Beendigung, wenn der Sohnprozeß abgebrochen oder gestoppt wurde:

0000 :	normale Terminierung
signr :	Signalnummer des Signals, das den Abbruch des Sohnprozesses verursachte (siehe signal)
0177 :	Sohnprozeß wurde gestoppt, ist aber noch nicht fertig und kann wieder gestartet werden (siehe ptrace)
0200 :	dieses Bit wird zusätzlich gesetzt, wenn ein Speicherabzug abgelegt wurde

Ergebnis

Prozeßnummer des Sohnprozesses
bei Erfolg

-1 wait wurde durch ein Signal unterbrochen (siehe signal)

-1 Fehler, wenn keine Sohnprozesse existieren

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EINTR :	Systemaufruf wurde unterbrochen
ECHILD :	Keine Kindprozesse

Hinweis

- Durch einen wait Aufruf erfährt der Vaterprozeß nur vom Ende eines Sohnprozesses. Will er auf die Beendigung mehrerer Sohnprozesse warten, muß er für jeden dieser Prozesse wait aufrufen.
- Wenn der Vaterprozeß terminiert ohne auf seine noch lebenden Söhne zu warten, wird der Initialisierungsprozeß (Prozeßnummer 1) zum Vater.

Beispiel

Nach Beendigung des Sohnprozesses wird im Vaterprozeß abgeprüft, ob der Sohnprozeß normal terminiert hat oder abgebrochen wurde:

```
#include <stdio.h>

int konst = 0;

main()
{
   int status,sohn;
   switch(sohn = fork())
         {
          case -1 : exit(1);
          case  0 : printf("%d\n",2/konst);
               /* Division durch 0 ==> Programm bricht ab */
                   exit(128);
         }
              /* Vater wartet auf Beendigung des Sohnprozesses */
   while(wait(&status) != sohn)
         ;
               /* Exit-status 128 (im hoeheren Byte)
                  besagt normales Ende */e
   if(((status >> 8) & 255) == 128)
      printf("korrektes ende\n");
   else
      {
        printf("Sohnprozeß fehlerhaft beendet\n");
               /* Der Grund für das fehlerhafte Ende
                  steht im niedrigeren Byte */
        printf("Ursache : %d\n", (status & 255));
               /* Wenn das Bit 0200 (= 128 dezimal) gesetzt ist,
                  wurde Speicherabzug angelegt */
        if (status > 128)
           {
            printf("Speicherabzug wurde erzeugt\n");
            printf("Ausloesendes Signal : %d\n",(status - 128));
           }
      }
}
```

Wenn Sie das Programm laufen lassen, wird der Sohnprozeß wegen der unzulässigen Division durch 0 abgebrochen. Der Vaterprozeß bringt dann die entsprechende Meldung, wobei das Signal SIGILL 4 : unzulässiger Befehl den Abbruch verursachte.

Wenn Sie in obigem Programm z.B. konst = 1; setzten, endet der Sohnprozeß normal und liefert den exit-Status 128.

> > > > exit, fork, signal, pause

Elementare Schreiboperation

int write(dk,puf,anz**)**
int dk;
char *puf;
int anz;

write ist die elementare Schreiboperation.
write schreibt bis zu *anz* zusammenhängende Bytes aus dem Bereich, auf den *puf* zeigt, in die Datei mit Dateikennzahl *dk*.

Typ

Systemaufruf

Parameter

int dk	Dateikennzahl für die Ausgabedatei
char *puf	Zeiger auf den Bereich, in dem die Daten stehen, die in die Datei geschrieben werden sollen.
int anz	Anzahl der Bytes, die in die Datei geschrieben werden sollen. Es ist nicht sichergestellt, daß write tatsächlich *anz* Bytes schreibt!

Ergebnis

Anzahl der tatsächlich geschriebenen Bytes
bei Erfolg

–1 Fehler, wenn

- *dk* keine gültige Dateikennzahl ist oder
- auf die Ausgabedatei nicht geschrieben werden kann, weil sie nicht existiert oder keine Schreiberlaubnis besteht oder
- der Bereich, in dem die Daten stehen, nicht korrekt angegeben ist oder
- ein physikalischer Ein/Ausgabefehler vorliegt, oder
- das maximale Dateilimit überschritten würde (siehe ulimit) oder
- während dem write Aufruf ein Signal abgefangen wurde (siehe signal)

Fehlermeldung

Bei Ergebnis -1 steht in errno ein entsprechender Fehlercode:

EBADF :	Unzulässige Dateinummer
EACCES :	Zugriff untersagt
EFAULT :	Unzulässige Adresse
EIO :	Ein/Ausgabe Fehler
EFBIG :	Datei zu groß
EINTR :	Systemaufruf wurde unterbrochen

Hinweis

- Sie sollten nach jedem write Aufruf die Anzahl der tatsächlich geschriebenen Bytes überprüfen. Stimmt das Ergebnis nicht mit der Angabe in *anz* überein, liegt i.a. ein Fehler vor.
- Um sicherzugehen, daß Ihre Angabe in *anz* die Größe des Puffers nicht überschreitet, können Sie die Funktion sizeof verwenden.
- Die zwei häufigsten Angaben für *anz* sind:

  1 : für zeichenweise Ausgabe (ungepuffert)
  BUFSIZ : für gepufferte Ausgabe. Die Zahl BUFSIZ entspricht der physikalischen Blockgröße der meisten blockorienten Geräte.
- Wenn das Bit O_APPEND im System-Dateistatus-Byte gesetzt ist, wird der Lese/Schreibzeiger vor dem Schreiben auf Dateiende gesetzt.
- Das Bit O_NDELAY im System-Dateistatus-Byte (siehe fcntl, open) wirkt sich bei einem write auf eine Pipe oder ein zeichenorientiertes Gerät wie folgt aus:

  O_NDELAY gesetzt:
  Ist kein Platz zum Schreiben vorhanden, kehrt write sofort mit Ergebnis 0 zurück.

  O_NDELAY nicht gesetzt:
  write blockiert solange, bis Platz zum Schreiben vorhanden ist.

Beispiel

siehe Beispiel bei read.

Dateien

/usr/include/stdio.h
Definition von BUFSIZ

> > > > read, open, creat, pipe

Besselfunktionen der zweiten Art

#include < math.h >

double y0(x**)**
double x;

double y1(x**)**
double x;

double yn(n,x**)**
int n;
double x;

Die Funktionen y0, y1 und yn berechnen die Besselfunktionen der zweiten Art für reelle Argumente x und ganzzahlige Ordnungen n.

Typ

C-Funktion

Parameter

double x	reelles Argument
int n	ganzzahlige Ordnung

Ergebnis

Bessel Funktion für x

Dateien

/usr/include/math.h
Deklaration mathematischer Funktionen

> > > > j0, j1, jn

A Anhang

Inhalt

Version 1.0B A-2
Version 1.0C A-8
Version 2.0 . . . . A-13
ASCII-Tabelle A-17

Version 1.0B

HARDWARE

Rechner: PC-X
Prozessor: Intel 80186 16 Bit

C-KONSTANTEN

Konstante	Bedeutung	Wert
BUFSIZ	Standard-Blockgröße	512
EOF	Ende der Eingabe	-1
NULL	Nullzeiger	0
HUGE	Wert bei Überlauf	nicht definiert

DEFINITIONSBEREICHE DER ELEMENTAREN C-DATENTYPEN

Datentyp	Defintionsbereich	Länge in Bit
int, short	[-{INT}, +{INT}-1]	16
long	[-{LONGINT}, +{LONGINT}-1]	32
char	[{CHAR_MIN}, {CHAR_MAX}]	8
float	[-{FLOAT}, +{FLOAT}]	32
double	[-{DOUBLE}, +{DOUBLE}]	64

Variablen vom Typ Zeiger (<typ> *p) haben 16 Bit Länge.

SYSTEM-KONSTANTEN (alphabetisch)

Konstante	Bedeutung	Wert
{ARG_MAX}	maximale Länge der Argumentliste bei exec	5K
{BPROZ_MAX}	Maximale Anzahl von Prozessen pro Benutzer	15
{BRK_ZAHL}	Rundungszahl zum Berechnen der end-Adresse bei brk	2K
{CHAR_MAX}	obere Intervallgrenze für den Datentyp char	255
{CHAR_MIN}	untere Intervallgrenze für den Datentyp char	0
{DOUBLE}	Intervallgrenze für den den Datentyp double	10^{38}
{HZ}	Zeitauflösung	20HZ = 50ms
{FLOAT}	Intervallgrenze für den Datentyp float	10^{38}
{INT}	Intervallgrenze für den Datentyp int	2^{15}
{LINK_MAX}	Maximale Anzahl von Verweisen auf eine Datei	1000
{LONGINT}	Intervallgrenze für den Datentyp long	2^{31}
{LOCK_MAX}	Maximale Anzahl von Sperren auf eine Datei	100
{MOUNT_MAX}	Maximale Anzahl eingehängter Dateisysteme	8
{NAME_MAX}	Anzahl signifikanter Zeichen, die der Übersetzer erkennt	8
{PDAT_MAX}	Maximale Anzahl offener Dateien pro Prozeß	35

Konstante	Bedeutung	Wert
{PIPE_MAX}	Maximale Anzahl von Bytes, die bei pipe gepuffert werden	4096
{PUF_MAX}	Maximale Anzahl von Puffern im System	38
{SDAT_MAX}	Maximale Anzahl offener Dateien pro System	100
{SHORTINT}	Intervallgrenze für den Datentyp short	2^{15}
{SPROZ_MAX}	Maximale Anzahl von Prozessen pro System	80

FUNKTIONSUMFANG

In der Version 1.0B sind folgende Funktionen nicht vorhanden:

fcntl
getpgrp
getppid
setbuffer
setlinebuf
setpgrp
ulimit

FUNKTIONSUNTERSCHIEDE

Folgenden Funktionen sind in Version 1.0B in eingeschränkter Form vorhanden:

Funktion	Einschränkung
kill	weniger Möglichkeiten bei der Auswahl der betroffenen Prozesse nur 16 Signale sind definiert
open	nur 2 Parameter mit dem 2. Parameter kann lediglich die Zugriffsart (lesen, schreiben oder beides) bestimmt werden keine symbolischen Konstanten aus <sys/fcntl.h>
printf	Formatangabe '+' funktioniert nicht
signal	nur 16 Signale sind definiert
tolower, toupper	vor der Umwandlung wird nicht überprüft, ob das Argument ein Groß- bzw. Kleinbuchstabe ist

include-DATEIEN

Wie bereits in der Einführung erwähnt wurde, sind einige include-Dateien vor allem im Dateiverzeichnis /usr/include/sys äußerst maschinenabhängig. Sie sollten daher vor Gebrauch einer solchen Datei in ihrem System den Inhalt der Datei überprüfen. Die wichtigsten im Buch erwähnten include-Dateien, die Unterschiede in den Versionen aufzeigen, sind hier nochmals zusammengestellt:

<sys/fcntl.h>	nicht vorhanden
<sys/types.h>	Typdefinitionen für verschiedene integer-Datentypen
<sys/times.h>	Name der Struktur hier: tbuffer
<sys/timeb.h>	wird von ftime verwendet
<sys/stat.h>	Definition der Dateistatus-Bits
<a.out.h>	wird von nlist gebraucht
<stdio.h>	Definitionen für Standardein/ausgabe
<signal.h>	Definition der Signale
<param.h>	implementierungsabhängige Konstanten

Was Sie bei Version 1.0B berücksichtigen müssen

1) Für die Gleitkomma-Arithmetik muß in der Version 1.0B immer die Bibliothek: /lib/libffp.a dazugebunden werden, z.B. beim Aufruf des Übersetzers durch cc progname.c -lffp.
   Sollten Sie mehrere Bibliotheken benötigen, müssen Sie die Reihenfolge der Angaben berücksichtigen.
   Folgende Funktionen aus CES sind davon betroffen:

```
acos, asin, atan, atof, cabs, ceil, cos,
cosh, ecvt, exp, fabs, floor, frexp, gcvt,
hypot, ldexp, log, log10, modf, pow, sin,
sinh, sqrt, tan, tanh, y0, y1, yn, j0, j1, jn
bei fprintf, printf, sprintf und fscanf,
scanf, sscanf, falls Gleitkomma-Umwandlung gewünscht ist.
```

2) Das Fehlen von fcntl wirkt sich wie folgt aus:

   - eine Dateikennzahl kann nur mittels dup oder dup2 verdoppelt werden.
   - das sbe-Bit im Dateistatus-Byte kann nicht explizit verändert werden. Überall, wo es erwähnt ist (creat, dup, dup2, exec, open), gilt daher immer die Standardeinstellung (0).
   - das System-Dateistatus-Byte kann nicht explizit verändert werden.

3) Der eingeschränkte Systemaufruf open hat zur Folge, daß

   - Dateien nur mittels creat oder mknod eingerichtet werden können,
   - das System-Dateistatus-Byte nicht explizit gesetzt werden kann.

4) Das Fehlen von setprp, getpgrp und getppid wirkt sich wie folgt aus:

   - es können keine neuen Prozeßgruppen eröffnet werden,
   - die Prozeßgruppennummer und die Prozeßnummer des Vaters können nicht abgefragt werden.

5) Es gibt keine Möglichkeit "zeilenweise Pufferung" einzustellen.

6) In der Version 1.0B ist der Ergebnistyp "void" noch nicht implementiert. Bei allen Funktionen, bei denen dieser Datentyp als Ergebnistyp angegeben ist, können Sie stattdessen den Typ "int" einsetzen.

7) Der Preprozessor und der Übersetzer verarbeiten 8 signifikante Zeichen (einschließlich '_').

Version 1.0C

HARDWARE

Rechner: PC-MX
Prozessor: Intel 8086 16 Bit

C-KONSTANTEN

Konstante	Bedeutung	Wert
BUFSIZ	Standard-Blockgröße	1024
EOF	Ende der Eingabe	-1
NULL	Nullzeiger	0
HUGE	Wert bei Überlauf	nicht definiert

DEFINITIONSBEREICHE DER ELEMENTAREN C-DATENTYPEN

Datentyp	Defintionsbereich	Länge in Bit
int, short	[-{INT}, +{INT}-1]	16
long	[-{LONGINT}, +{LONGINT}-1]	32
char	[{CHAR_MIN}, {CHAR_MAX}]	8
float	[-{FLOAT}, +{FLOAT}]	32
double	[-{DOUBLE}, +{DOUBLE}]	64

Variablen vom Typ Zeiger (<typ> *p) haben 16 Bit Länge.

SYSTEM-KONSTANTEN (alphabetisch)

Konstante	Bedeutung	Wert
{ARG_MAX}	maximale Länge der Argumentliste bei exec	5K
{BPROZ_MAX}	Maximale Anzahl von Prozessen pro Benutzer	15
{BRK_ZAHL}	Rundungszahl zum Berechnen der end-Adresse bei brk	2K
{CHAR_MAX}	obere Intervallgrenze für den Datentyp char	255
{CHAR_MIN}	untere Intervallgrenze für den Datentyp char	0
{DOUBLE}	Intervallgrenze für den den Datentyp double	10^{38}
{HZ}	Zeitauflösung	20HZ = 50ms
{FLOAT}	Intervallgrenze für den Datentyp float	10^{38}
{INT}	Intervallgrenze für den Datentyp int	2^{15}
{LINK_MAX}	Maximale Anzahl von Verweisen auf eine Datei	1000
{LONGINT}	Intervallgrenze für den Datentyp long	2^{31}
{LOCK_MAX}	Maximale Anzahl von Sperren auf eine Datei	100
{MOUNT_MAX}	Maximale Anzahl eingehängter Dateisysteme	8
{NAME_MAX}	Anzahl signifikanter Zeichen, die der Übersetzer erkennt	8
{PDAT_MAX}	Maximale Anzahl offener Dateien pro Prozeß	35

Konstante	Bedeutung	Wert
{PIPE_MAX}	Maximale Anzahl von Bytes, die in einer Pipe gepuffert werden	4096
{PUF_MAX}	Maximale Anzahl von Puffern im System	40
{SDAT_MAX}	Maximale Anzahl offener Dateien pro System	100
{SHORTINT}	Intervallgrenze für den Datentyp short	2^{15}
{SPROZ_MAX}	Maximale Anzahl von Prozessen pro System	80

FUNKTIONSUMFANG

In der Version 1.0C sind folgende Funktionen nicht vorhanden:

setbuffer
setlinebuf

FUNKTIONSUNTERSCHIEDE

Folgenden Funktionen sind in Version 1.0C in eingeschränkter Form vorhanden:

Funktion	Einschränkung
kill	nur 19 Signale sind definiert
printf	Formatangabe '+' funktioniert nicht
signal	nur 19 Signale sind definiert
tolower, toupper	vor der Umwandlung wird nicht überprüft, ob das Argument ein Groß- bzw. Kleinbuchstabe ist

include-DATEIEN

Wie bereits in der Einführung erwähnt wurde, sind einige include-Dateien vor allem im Dateiverzeichnis /usr/include/sys äußerst maschinenabhängig. Sie sollten daher vor Gebrauch einer solchen Datei in ihrem System den Inhalt der Datei überprüfen. Die wichtigsten im Buch erwähnten include-Dateien, die Unterschiede in den Versionen aufzeigen, sind hier nochmals zusammengestellt:

<sys/types.h>	Typdefinitionen für verschiedene integer-Datentypen
<sys/times.h>	Name der Struktur hier: tms
<sys/timeb.h>	wird von ftime verwendet
<sys/stat.h>	Definition der Dateistatus-Bits
<a.out.h>	wird von nlist gebraucht
<stdio.h>	Definitionen für Standardein/ausgabe
<signal.h>	Definition der Signale
<param.h>	Definition von implementierungsabhängigen Konstanten

Was Sie bei Version 1.0C berücksichtigen müssen

1) Für die Gleitkomma-Arithmetik muß in der Version 1.0C immer die Bibliothek: /lib/libffp.a dazugebunden werden, z.B. beim Aufruf des Übersetzers durch cc progname.c -lffp.
   Sollten Sie mehrere Bibliotheken benötigen, müssen Sie die Reihenfolge der Angaben berücksichtigen.
   Folgende Funktionen aus CES sind davon betroffen:

```
acos, asin, atan, atof, cabs, ceil, cos,
cosh, ecvt, exp, fabs, floor, frexp, gcvt,
hypot, ldexp, log, log10, modf, pow, sin,
sinh, sqrt, tan, tanh, y0, y1, yn, j0, j1, jn
bei fprintf, printf, sprintf und fscanf,
scanf, sscanf, falls Gleitkomma-Umwandlung gewünscht ist.
```

2) Es gibt keine Möglichkeit "zeilenweise Pufferung" einzustellen.

3) In der Version 1.0C ist der Ergebnistyp "void" noch nicht implementiert. Bei allen Funktionen, bei denen dieser Datentyp als Ergebnistyp angegeben ist, können Sie stattdessen den Typ "int" einsetzen.

4) Der Preprozessor und der Übersetzer verarbeiten 8 signifikante Zeichen (einschließlich '_').

Version 2.0

HARDWARE

Rechner: PC-MX2, PC-MX4
Prozessor: NS32016 32 Bit

C-KONSTANTEN

Konstante	Bedeutung	Wert
BUFSIZ	Standard-Blockgröße	1024
EOF	Ende der Eingabe	-1
NULL	Nullzeiger	0
HUGE	Wert für Überlauf	0

DEFINITIONSBEREICHE DER ELEMENTAREN C-DATENTYPEN

Datentyp	Defintionsbereich	Länge in Bit
int, long	[-{INT}, +{INT}-1]	32
short	[-{SHORTINT}, +{SHORTINT}-1]	16
char	[{CHAR_MIN}, {CHAR_MAX}]	8
float	[-{FLOAT}, +{FLOAT}]	32
double	[-{DOUBLE}, +{DOUBLE}]	64

Variablen vom Typ Zeiger (<typ> *p) haben 32 Bit Länge.

SYSTEM-KONSTANTEN (alphabetisch)

Konstante	Bedeutung	Wert
{ARG_MAX}	maximale Länge der Argumentliste bei exec	10K
{BPROZ_MAX}	Maximale Anzahl von Prozessen pro Benutzer	20
{BRK_ZAHL}	Rundungszahl zum Berechnen der end-Adresse bei brk	??
{CHAR_MAX}	obere Intervallgrenze für den Datentyp char	255
{CHAR_MIN}	untere Intervallgrenze für den Datentyp char	0
{DOUBLE}	Intervallgrenze für den den Datentyp double	1.8e308
{HZ}	Zeitauflösung	50HZ = 20ms
{FLOAT}	Intervallgrenze für den Datentyp float	3.4e38
{INT}	Intervallgrenze für den Datentyp int	2^{31}
{LINK_MAX}	Maximale Anzahl von Verweisen auf eine Datei	1000
{LOCK_MAX}	Maximale Anzahl von Sperren auf eine Datei	100
{LONGINT}	Intervallgrenze für den Datentyp long	2^{31}
{MOUNT_MAX}	Maximale Anzahl eingehängter Dateisysteme	8
{NAME_MAX}	Anzahl signifikanter Zeichen, die der Übersetzer erkennt	16
{PDAT_MAX}	Maximale Anzahl offener Dateien pro Prozeß	35

Konstante	Bedeutung	Wert
{PIPE_MAX}	Maximale Anzahl von Bytes, die in einer Pipe gepuffert werden	4096
{PUF_MAX}	Maximale Anzahl von Puffern im System	uninteres-sant
{SDAT_MAX}	Maximale Anzahl offener Dateien pro System	106
{SHORTINT}	Intervallgrenze für den Datentyp short	2^{15}
{SPROZ_MAX}	Maximale Anzahl von Prozessen pro System	82

FUNKTIONSUMFANG

In der Version 2.0 sind alle Funktionen vorhanden.

include-DATEIEN

Wie bereits in der Einführung erwähnt wurde, sind einige include-Dateien vor allem im Dateiverzeichnis /usr/include/sys äußerst maschinenabhängig. Sie sollten daher vor Gebrauch einer solchen Datei in ihrem System den Inhalt der Datei überprüfen. Die wichtigsten im Buch erwähnten include-Dateien, die Unterschiede in den Versionen aufzeigen, sind hier nochmals zusammengestellt:

```
<sys/types.h>    Typdefinitionen für verschiedene integer-Datentypen
<sys/times.h>    Name der Struktur hier: tms
<sys/timeb.h>    wird von ftime verwendet
<sys/stat.h>     Definition der Dateistatus-Bits

<a.out.h>        wird von nlist gebraucht
<nlist.h>
<stdio.h>        Definitionen für Standardein/ausgabe
<signal.h>       Definition der Signale
<param.h>        Definition von implementierungsabhängigen Konstanten
```

Was Sie bei Version 2.0 berücksichtigen müssen

1) Der Preprozessor verarbeitet 32 signifikante Zeichen, der Übersetzer verarbeitet 16 signifikante Zeichen (einschließlich '_').

2) Der Adreßoperator vor Feldnamen ist nicht erlaubt!

3) Die veralteten Operatoren:
=+, =-, =>>, =<<
sind nicht mehr erlaubt.

4) Variablen vom Typ float und double werden im IEEE-Format abgespeichert.

ASCII-Tabelle

dezi-mal	oktal	hexa-dez.		Bedeutung	Control
0	00	00	NUL	Null, keine Operation	@
1	01	01	SOH	Start of Heading Vorspannanfang	A
2	02	02	STX	Start of Text Textanfang	B
3	03	03	ETX	End of Text Textende	C
4	04	04	EOT	End of Transmission Taste END Übertragungsende	D
5	05	05	ENQ	Enquiry Stationsanruf	E
6	06	06	ACK	Acknowledge Bestätigung	F
7	07	07	BEL	Bell Klingel	G
8	10	08	BS	Backspace Korrekturtaste	H
9	11	09	HT	Horizontal Tabulation Tabulatorzeichen	I
10	12	0A	LF	Line Feed Zeilenvorschub, neue Zeile	J
11	13	0B	VT	Vertical Tabulation	K
12	14	0C	FF	Form Feed Formularvorschub	L
13	15	0D	CR	Carriage Return Wagenrücklauf	M
14	16	0E	SO	Shift Out Umschalten Zeichensatz	N
15	17	0F	SI	Shift In Zurückschalten Zeichensatz	O
16	20	10	DLE	Data Link Escape Austritt aus der Datenverbindung	P
17	21	11	DC1	Device Control 1 Gerätesteuerung 1, Ausgabe fortsetzen	Q
18	22	12	DC2	Device Control 2	R
19	23	13	DC3	Device Control 3 Ausgabe anhalten	S
20	24	14	DC4	Device Control 4	T
21	25	15	NAK	Negative Acknowledge Fehlermeldung	U
22	26	16	SYN	Synchronous Idle Synchronisierung	V
23	27	17	ETB	End of Transm. Block Datenblockende	W
24	30	18	CAN	Cancel ungültig, Zeilenlöscher	X
25	31	19	EM	End of Medium Datenträgerende, quit (Signal3)	Y
26	32	1A	SUB	Substitute Character Zeichen ersetzen	Z

dezi-mal	oktal	hexa-dez.		Bedeutung		Control
27	33	1B	ESC	Escape	Rücksprung	
28	34	1C	FS	File Separator	Dateitrennung	\
29	35	1D	GS	Group Separator	Gruppentrennung	]
30	36	1E	RS	Record Separator	Satztrennung	^
31	37	1F	US	Unit Separator	Einheitentrennung	_ oder DEL
32	40	20	SP	SPACE	Leerzeichen	
33	41	21	!			
34	42	22	"			
35	43	23	#	Nummernzeichen		
36	44	24	$	oder nationales Währungssymbol		
37	45	25	%			
38	46	26	&			
39	47	27	'			
40	50	28	(			
41	51	29	)			
42	52	2A	*	(Stern gilt oft als Multiplikations-zeichen)		
43	53	2B	+			
44	54	2C	'			
45	55	2D	–			
46	56	2E	.			
47	57	2F	/	(dient als Divisionszeichen)		
48	60	30	0			
49	61	31	1			
50	62	32	2			
51	63	33	3			
52	64	34	4			
53	65	35	5			
54	66	36	6			
55	67	37	7			
56	70	38	8			
57	71	39	9			
58	72	3A	:			
59	73	3B	;			
60	74	3C	<			
61	75	3D	=			
62	76	3E	>			
63	77	3F	?			
64	100	40	@	(kaufmännisches "at" oder $)		
65	101	41	A			
66	102	42	B			
67	103	43	C			
68	104	44	D			
69	105	45	E			
70	106	46	F			
71	107	47	G			
72	110	48	H			
73	111	49	I			
74	112	4A	J			
75	113	4B	K			
76	114	4C	L			
77	115	4D	M			

dezi-mal	oktal	hexa-dez.		Bedeutung	Control
78	116	4E	N		
79	117	4F	O		
80	120	50	P		
81	121	51	Q		
82	122	52	R		
83	123	53	S		
84	124	54	T		
85	125	55	U		
86	126	56	V		
87	127	57	W		
88	130	58	X		
89	131	59	Y		
90	132	5A	Z		
91	133	5B	[	oder Ä	
92	134	5C	\	Gegenschrägstrich oder Ö	
93	135	5D	]	oder Ü	
94	136	5E	^	oder ↑	
95	137	5F	_	Unterstrich oder ←	
96	140	60	`		
97	141	61	a		
98	142	62	b		
99	143	63	c		
100	144	64	d		
101	145	65	e		
102	146	66	f		
103	147	67	g		
104	150	68	h		
105	151	69	i		
106	152	6A	j		
107	153	6B	k		
108	154	6C	l		
109	155	6D	m		
110	156	6E	n		
111	157	6F	o		
112	160	70	p		
113	161	71	q		
114	162	72	r		
115	163	73	s		
116	164	74	t		
117	165	75	u		
118	166	76	v		
119	167	77	w		
120	170	78	x		
121	171	79	y		
122	172	7A	z		
123	173	7B	{	oder ä	
124	174	7C	|	oder ö	
125	175	7D	}	oder ü	
126	176	7E	~	oder ß	
127	177	7F	DEL	Delete Löschzeichen, Interrupt (Signal2)	

Fachwörter

Fachwörter deutsch - englisch

abfangen (Signal)	catch
Ablauf verfolgen	trace
Adresse	address
Adreßraum	address space
aktuelles Dateiverzeichnis	working directory
aktuelles Umfeld	current environment
Alarmuhr	alarm clock
Anfrage	request
Anzeiger	flag
aufrufen	call
ausführen, durchsuchen	execute
Ausführberechtigung	execute permission
Ausführung	execution
Ausgabe	output
aushängen (Dateisystem)	umount
austauschen	swap
Benutzernummer	user ID (UID)
Benutzerzeit	user time
Bereinigung	cleanup
Bibliothek	library
Bildschirm	screen
Bildschirm, Datensichtstation	terminal
Bildschirmgruppe	tty group
blockorientiertes Gerät	block special device
break	break
Bruchteil	fractional part
Byte = 8 Bit	byte
Datei	file
Datei mit FILE-Struktur	stream
Dateikennzahl	file descriptor
Dateiname	file name
Dateistatus	file status
Dateisystem	file system
Dateiverzeichnis	directory
Dateizeiger	file pointer
Datensegment	data area

Datensegment, nicht initialisierte Daten	bss
dezimal (10)	decimal
Diskette	floppy disk
dual (2)	dual
dynamisch	dynamic
effektive Benutzernummer	effective UID
effektive Gruppennummer	effective GID
Eigentümer	owner
Eingabe	input
einhängen (Dateisystem)	mount
Endestatus	exit status
Ergebnis	result
Fehler	error
Fehlercode	error code
Fehlersuche	debugging
Feld, Vektor	array
Festplatte	disk
FILE-Struktur	FILE structure
Format	format
formatierte Ein/Ausgabe	formated I/O
freigeben	free
Gegenschrägstrich	backslash
gepufferte Ein/Ausgabe	buffered I/O
Gerät, externer Datenträger	device
Gerätedatei	special file
Gleitkommazahl	floating point number
Greenwich Meantime	GMT
Gruppe	group
Gruppenname	groupname
Gruppennummer	group ID (GID)
Haltepunkt	breakpoint
hexadezimal (16)	hexadecimal
Home-Dateiverzeichnis	home directory
include-Datei	include file
Indexeintrag	inode
Indexnummer	inumber
initialisieren	initialize

Keller	stack (LIFO)
Konsole	console
Kontrollzeichen	contol character
Korrekturtaste	back space key
Laufzeit	elapsed time
Leerzeichen	blank
Leseerlaubnis	read permission
lesen	read
magic number	magic number
major Nummer (Gerätetyp)	major number
Maske	mask
mehrfach benutzbar	sharable
minor Nummer (Gerätenummer)	minor number
Mitteleuropäische Zeit	MEZ
Modus	mode
Muster	pattern
Neue Zeile	new line
Nullbyte	null byte (\0)
Nullzeiger	null pointer
oktal (8)	octal
Pfad	path
Pfadname	pathname
Pipe, Einwegkanal	pipe
Priorität	priority
Prozeß	process
Prozeß wieder aufsetzen	resume execution
Prozeß-Dateistatus-Byte	file descriptor flag
Prozeßbeendigung	process termination
Prozeßgruppe	process group
Prozeßgruppenchef	process group leader
Prozeßgruppennummer	process grouppid
Prozeßmaske	process's mode mask
Prozeßnummer	process ID (PID)
Prozeßnummer des Vaters	parent process ID (PPID)
Prozeßspezifische Daten im System	USER area
Puffer	buffer
Quellcode	source code

reale Benutzernummer	real UID
reale Gruppennummer	real GID
reale Prozeßnummer	real PID
regulärer Ausdruck	regular expression
Root-Dateiverzeichnis	root directory
Rückkehr	return
s-Bit(Eigentümer,Gruppe)	setuid-Bit (s-Bit)
sbe-Bit (schließe bei exec)	close-on-exec Byte
Schlüssel	key
schreiben	write
Schreiberlaubnis	write permission
Schutzbit	protection bit
serielle Schnittstelle	asynchron communication
Shell	shell
Signal	signal
Sohnprozeß	child process
Sonderzeichen	special character
Speicherabzug	core
Speicherplatz	memory
sperren	lock
Standard-Fehlerausgabe	standarderror
Standardausgabe	standard output
Standardeingabe	standard input
statisch	static
statischer Datenbereich	static data area
steuern	schedule
Struktur	structure
Swapbereich	swap area
Symboltabelle	symbol table
System	system
System-Dateistatus-Byte	file flag
Systembenutzer	user
Systemkern	kernel
Systemprozeß	special process
Systemzeit	system time
Sytemverwalter	super user
t-Bit	sticky-Bit
Textsegment	text image
Trace Flag	trace flag
Treiber	driver

Überlauf	overflow
Übersetzer	compiler
Umgebung	environment
Umgebungsvariable	environmental variables
Umleitung	redirection
Unterbrechung	interrupt
Vaterprozeß	parent process
Verkettung	concatenation
Verweis	link
Vorzeichen	sign
Warteschlange	queue (FIFO)
Zeichen	character
zeichenorientiertes Gerät	character special device
Zeichenreihe	string
Zeiger	pointer
Zeitzone	timezone
Zentraleinheit	CPU
Ziffer	digit
Zufallszahl	random number
Zugriff	access
Zugriffsart	access mode
Zugriffsberechtigung	access permission
Zustand	state
Zwischenraum	space

Fachwörter englisch - deutsch

access	Zugriff
access mode	Zugriffsart
access permission	Zugriffsberechtigung
address	Adresse
address space	Adreßraum
alarm clock	Alarmuhr
array	Feld, Vektor
asynchron communication	serielle Schnittstelle
back space key	Korrekturtaste
backslash	Gegenschrägstrich
blank	Leerzeichen
block special device	blockorientiertes Gerät
break	break
breakpoint	Haltepunkt
bss	Datensegment, nicht initialisierte Daten
buffer	Puffer
buffered I/O	gepufferte Ein/Ausgabe
byte	Byte = 8 Bit
call	aufrufen
catch	abfangen (Signal)
character	Zeichen
character special device	zeichenorientiertes Gerät
child process	Sohnprozeß
cleanup	Bereinigung
close-on-exec Byte	sbe-Bit (schließe bei exec)
compiler	Übersetzer
concatenation	Verkettung
console	Konsole
contol character	Kontrollzeichen
core	Speicherabzug
CPU	Zentraleinheit
current environment	aktuelles Umfeld
data area	Datensegment
debugging	Fehlersuche
decimal	dezimal (10)
device	Gerät, externer Datenträger
digit	Ziffer
directory	Dateiverzeichnis
disk	Festplatte

driver	Treiber
dual	dual (2)
dynamic	dynamisch
effective GID	effektive Gruppennummer
effective UID	effektive Benutzernummer
elapsed time	Laufzeit
environment	Umgebung
environmental variables	Umgebungsvariable
error	Fehler
error code	Fehlercode
execute	ausfuehren, durchsuchen
execute permission	Ausführberechtigung
execution	Ausführung
exit status	Endestatus
file	Datei
file descriptor	Dateikennzahl
file descriptor flag	Prozeß-Dateistatus-Byte
file flag	System-Dateistatus-Byte
file name	Dateiname
file pointer	Dateizeiger
file status	Dateistatus
FILE structure	FILE-Struktur
file system	Dateisystem
flag	Anzeiger
floating point number	Gleitkommazahl
floppy disk	Diskette
format	Format
formated I/O	formatierte Ein/Ausgabe
fractional part	Bruchteil
free	freigeben
GMT	Greenwich Meantime
group	Gruppe
group ID (GID)	Gruppennummer
groupname	Gruppenname
hexadecimal	hexadezimal (16)
home directory	Home-Dateiverzeichnis
include file	include-Datei
initialize	initialisieren
inode	Indexeintrag
input	Eingabe
interrupt	Unterbrechung

inumber	Indexnummer
kernel	Systemkern
key	schlüssel
library	Bibliothek
link	Verweis
lock	sperren
magic number	magic number
major number	major Nummer (Gerätetyp)
mask	Maske
memory	Speicherplatz
MEZ	Mitteleuropäische Zeit
minor number	minor Nummer (Gerätenummer)
mode	Modus
mount	einhängen (Dateisystem)
new line	Neue Zeile
null byte (\0)	Nullbyte
null pointer	Nullzeiger
octal	oktal (8)
output	Ausgabe
overflow	Überlauf
owner	Eigentümer
parent process	Vaterprozeß
parent process ID (PPID)	Prozeßnummer des Vaters
path	Pfad
pathname	Pfadname
pattern	Muster
pipe	Pipe, Einwegkanal
pointer	Zeiger
priority	Priorität
process	Prozeß
process group	Prozeßgruppe
process group leader	Prozeßgruppenchef
process grouppid	Prozeßgruppennummer
process ID (PID)	Prozeßnummer
process termination	Prozeßbeendigung
process's mode mask	Prozeßmaske
protection bit	Schutzbit
queue (FIFO)	Warteschlange

random number	Zufallszahl
read	lesen
read permission	Leseerlaubnis
real GID	reale Gruppennummer
real PID	reale Prozeßnummer
real UID	reale Benutzernummer
redirection	Umleitung
regular expression	regulärer Ausdruck
request	Anfrage
result	Ergebnis
resume execution	Prozeß wieder aufsetzen
return	Rückkehr
root directory	Root-Dateiverzeichnis
schedule	steuern
screen	Bildschirm
setuid-Bit (s-Bit)	s-Bit(Eigentümer,Gruppe)
sharable	mehrfach benutzbar
shell	Shell
sign	Vorzeichen
signal	Signal
source code	Quellcode
space	Zwischenraum
special character	Sonderzeichen
special file	Gerätedatei
special process	Systemprozeß
stack (LIFO)	Keller
standard input	Standardeingabe
standard output	Standardausgabe
standarderror	Standard-Fehlerausgabe
state	Zustand
static	statisch
static data area	statischer Datenbereich
sticky-Bit	t-Bit
stream	Datei mit FILE-Struktur und Dateizeiger
string	Zeichenreihe
structure	Struktur
super user	Sytemverwalter

swap	austauschen
swap area	Swapbereich
symbol table	Symboltabelle
system	System
system time	Systemzeit
terminal	Bildschirm, Datensichtstation
text image	Textsegment
timezone	Zeitzone
trace	Ablauf verfolgen
trace flag	Trace Flag
tty group	Bildschirmgruppe
umount	aushängen (Dateisystem)
user	Systembenutzer
USER area	Prozeßspezifische Daten im System
user ID (UID)	Benutzernummer
user time	Benutzerzeit
working directory	aktuelles Dateiverzeichnis
write	schreiben
write permission	Schreiberlaubnis

Literatur

/1/ CES - Buch 1
Grundlagen und Kommandos
Bestellnummer:

/2/ Betriebssytem SINIX, Buch1
Betstellnummer: U1901-J-Z95-1

/3/ UNIX Time-sharing-System, UNIX Programmer's Manual Vol. 1-3
Seventh Edition, Murray Hill: Bell Telephone Laboratories 1979

/4/ X/OPEN Portability Guide
Elsevier Science Publishing company New York, 1985

/5/ The Bell System Technical Journal
Volume 57, No. 6, Part 2
American Telephone and Telegraph Company, 1978

/6/ Jürgen Gulbins
UNIX
Springer-Verlag Berlin, 1984

/7/ M.Banahan, A. Rutter
UNIX- lernen, verstehen, anwenden
Hanser Verlag München, 1984
Deutsche Ausgabe von Prof Dr.A.T.Schreiner, T.Mandry

/8/ B.W. Kernighan, D.M. Ritchie
Programmieren in C
Hanser Verlag München, 1983
deutsche Ausgabe von Prof Dr. A. T.Schreiner, Dr. Ernst Janich

/9/ Jack Purdom
Einführung in C
Markt&Technik- Verlag München, 1983
Deutsche Übersetzung Peter Lüke

/10/ Kaare Christian
The UNIX operating system
John Wiley&Sons, 1983

/11/ W.Kernighan, R.Pike
The UNIX Programming environment
Prentice-Hall, 1984