Systemnahe Software
WS 2006/2007

Andreas F. Borchert
Universität Ulm

1. Februar 2007

Hierarchie der Dateisysteme

- Jedes Dateisystem enthält eine Hierarchie der Verzeichnisse.
- Darüber hinaus gibt es auch eine Hierarchie der Dateisysteme.
- Es beginnt mit der Wurzel / und dem die Wurzel repräsentierenden Wurzel-Dateisystem. (Dies ist das erste Dateisystem, das verwendet wird und das auch das Betriebssystem oder zumindest wesentliche Teile davon enthält.)
- Weitere Dateisysteme können bei einem bereits existierenden Verzeichnis eingehängt werden.
- So entsteht eine globale Hierarchie, die sich über mehrere Dateisysteme erstreckt.

Hierarchie der Dateisysteme

```
doolin$ cd /
doolin$ df .
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s0    8263277 3705376 4475269    46%    /
doolin$ cd /var
doolin$ df .
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s7    8263277 2002000 6178645    25%    /var
doolin$
doolin$ cd /
doolin$ df -h .
Filesystem             size   used  avail capacity  Mounted on
/dev/dsk/c0t0d0s0      7.9G   3.5G   4.3G    46%    /
doolin$ cd var
doolin$ df -h .
Filesystem             size   used  avail capacity  Mounted on
/dev/dsk/c0t0d0s7      7.9G   1.9G   5.9G    25%    /var
doolin$ cd run
doolin$ df -h .
Filesystem             size   used  avail capacity  Mounted on
swap                   1.6G    48K   1.6G     1%    /var/run
doolin$
```

Historischer Aufbau eines Unix-Dateisystems

Boot–block	Super–block	Inode–Liste	Datenblöcke

- In den 70er Jahren (bis einschließlich UNIX Edition VII) hatte ein Unix-Dateisystem einen sehr einfachen Aufbau, bestehend aus
  - dem Boot-Block (reserviert für den Boot-Vorgang oder ohne Verwendung),
  - dem Super-Block (mit Verwaltungsinformationen für die gesamte Platte),
  - einem festdimensionierten Array von Inodes (Verwaltungsinformationen für einzelne Dateien) und
  - einem Array von Blöcken, die entweder für Dateiinhalte oder (im Falle sehr großer Dateien) für Verweise auf weitere Blöcke einer Datei verwendet werden.

Das UFS-Dateisystem

- Das heutige UFS (*UNIX file system*) geht zurück auf das von Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler und Robert S. Fabry Anfang der 80er Jahre entwickelte Berkeley Fast File System.
- Gegenüber dem historischen Aufbau enthält es einige wesentliche Veränderungen:
  - Die Verwaltungsinformationen einer Datei und der Dateiinhalt werden so auf der Platte abgelegt, dass sie möglichst schnell hintereinander gelesen werden können.
  - Dazu wird die Platte entsprechend ihrer Geometrie in Zylindergruppen aufgeteilt. Zusammenhängende Inodes und Datenblöcke liegen dann möglichst in der gleichen oder der benachbarten Zylindergruppe.
  - Die Blockgröße wurde vergrößert (von 1k auf 8k) und gleichzeitig wurden für kleine Dateien fragmentierte Blöcke eingeführt.
  - Damit der Verlust des Super-Blocks nicht katastrophal ist, gibt es zahlreiche Sicherungskopien des Super-Blocks an Orten, die sich durch die Geometrie ableiten lassen.
- Das unter Linux lange Zeit populäre ext2-Dateisystem hatte UFS als Vorbild.

Inode

- Eine Inode enthält sämtliche Verwaltungsinformationen, die zu einer Datei gehören.
- Jede Inode ist (innerhalb eines Dateisystems) eindeutig über die Inode-Nummer identifizierbar.
- Die Namen einer Datei sind *nicht* Bestandteil der Inode. Stattdessen bilden Verzeichnisse Namen in Inode-Nummern ab.
- U.a. finden sich folgende Informationen in einer Inode:
  - Eigentümer und Gruppe
  - Dateityp (etwa gewöhnliche Datei oder Verzeichnis oder einer der speziellen Dateien)
  - Zeitstempel: Letzter Lesezugriff, letzter Schreibzugriff und letzte Änderung der Inode.
  - Anzahl der Verweise aus Verzeichnissen
  - Länge der Datei in Bytes (bei gewöhnlichen Dateien und Verzeichnissen)
  - Blockadressen (bei gewöhnlichen Dateien und Verzeichnissen)

Auslesen eines Verzeichnisses

- In der Unix-Welt gibt es keine standardisierten Systemaufrufe, die ein Auslesen eines Verzeichnisses ermöglichen.
- Der Standard IEEE Std 1003.1 bietet jedoch die Funktionen *opendir*, *readdir* und *closedir* als portable Schnittstelle oberhalb der (nicht portablen) Systemaufrufe an.
- Alle anderen Funktionalitäten (Auslesen des öffentlichen Teils einer Inode, Wechseln des Verzeichnisses und sonstige Zugriffe auf Dateien) sind auch auf der Ebene der Systemaufrufe standardisiert.

Auslesen eines Verzeichnisses

dir.c

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
   char* cmdname = *argv++; --argc;
   char usage[] = "Usage: %s [directory]\n";
   if (argc > 1) {
      fprintf(stderr, usage, cmdname);
      exit(1);
   }
   char* dirname;
   if (argc > 0) {
      dirname = *argv;
   } else {
      dirname = ".";
   }
   /**** Auslesen von dirname ****/
}
```

Auslesen eines Verzeichnisses

dir.c

```
if (chdir(dirname) < 0) {
   perror(dirname);
   exit(1);
}
DIR* dir = opendir(".");
if (!dir) {
   perror(dirname);
   exit(1);
}
```

- Mit *chdir()* ist es möglich, das aktuelle Verzeichnis zu wechseln. Dies betrifft den aufrufenden Prozess (und wird später an neu erzeugte Prozesse weiter vererbt).
- *chdir()* wird hier verwendet, um im weiteren Verlauf den Zusammenbau zusammengesetzter Pfade aus dem Verzeichnisnamen und dem darin enthaltenen Dateinamen zu vermeiden.
- Nach dem Aufruf von *chdir()* ist das gewünschte (dann aktuelle) Verzeichnis unter dem Namen . erreichbar.

Auslesen eines Verzeichnisses

dir.c

```
struct dirent* entry;
while ((entry = readdir(dir))) {
   printf("%s: ", entry->d_name);
   struct stat statbuf;
   if (lstat(entry->d_name, &statbuf) < 0) {
      perror(entry->d_name); exit(1);
   }
   if (S_ISREG(statbuf.st_mode)) {
      printf("regular file with %lld bytes\n",
 (long long int)statbuf.st_size);
   } else if (S_ISDIR(statbuf.st_mode)) {
      puts("directory");
   } else if (S_ISLNK(statbuf.st_mode)) {
      char buf[1024];
      ssize_t len = readlink(entry->d_name, buf, sizeof buf);
      if (len < 0) {
         perror(entry->d_name); exit(1);
      }
      printf("symbolic link pointing to %.*s\n", len, buf);
   } else {
      puts("special");
   }
}
closedir(dir);
```

Auslesen eines Verzeichnisses

dir.c

```
struct dirent* entry;
while ((entry = readdir(dir))) {
   printf("%s: ", entry->d_name);
```

- *readdir* liefert einen Zeiger auf eine (statische) Struktur mit Informationen über die nächste Datei aus dem Verzeichnis.
- Die Struktur mag mehrere systemabhängige Komponenten haben. Relevant und portabel ist jedoch nur der Dateiname in dem Feld *d_name*

Auslesen einer Inode

dir.c

```
struct stat statbuf;
if (lstat(entry->d_name, &statbuf) < 0) {
   perror(entry->d_name); exit(1);
}
```

- Es gibt mehrere Systemaufrufe, die den öffentlichen Teil einer Inode auslesen können.
- Dazu gehört *lstat*, das einen Dateinamen erhält und dann in dem per Zeiger referenzierten Struktur die gewünschten Informationen aus der Inode ablegt.
- Im Unterschied zu *stat*, das genauso aufgerufen wird, folgt *lstat* nicht implizit symbolischen Links, so dass wir hier die Chance haben, diese als solche zu erkennen.

Das Feld *st_mode*

dir.c

```
      if (S_ISREG(statbuf.st_mode)) {
         printf("regular file with %lld bytes\n",
   (long long int)statbuf.st_size);
```

- Das Feld *st_mode* aus der von *lstat()* gefüllten Datenstruktur enthält in kombinierter Form mehrere Informationen über eine Datei:
  - den Dateityp,
  - die Zugriffsrechte (rwx) für Besitzer, Gruppe und den Rest der Welt und
  - eventuelle weitere besondere Attribute wie etwa das Setuid-Bit oder das Sticky-Bit.
- Damit der Zugriff weniger kompliziert ist, gibt es standardisierte Makros im Umgang mit *st_mode*. So liefert etwa *S_ISREG* den Wert **true**, falls es sich um eine gewöhnliche Datei handelt.

Auslesen eines symbolischen Links

dir.c

```
      } else if (S_ISLNK(statbuf.st_mode)) {
         char buf[1024];
         ssize_t len = readlink(entry->d_name, buf, sizeof buf);
         if (len < 0) {
            perror(entry->d_name); exit(1);
         }
         printf("symbolic link pointing to %.*s\n", len, buf);
```

- Wäre *stat()* an Stelle von *lstat()* verwendet worden, würde dieser Fall nie erreicht werden, da normalerweise symbolische Links implizit gefolgt wird.
- Mit *readlink()* kann der Link selbst ausgelesen werden.
- Das Ziel eines symbolischen Links muss nicht notwendigerweise existieren. Falls das Ziel nicht existiert, liefert *stat()* einen Fehler, während *lstat()* uns unabhängig von der Existenz das Ziel nennt.

Identitäten

- Bei Systemaufrufen sind, soweit sie von Privilegien und/oder einem Zugriffsschutz abhängig sind, folgende u.a. folgende vier Identitäten von Belang:

effektive Benutzernummer	*geteuid()*
effektive Gruppennummer	*getegid()*
reale Benutzernummer	*getuid()*
reale Gruppennummer	*getgid()*

- Normalerweise gleichen sich die effektiven und realen Nummern. Im Falle von Programmen mit dem s-bit werden die effektiven Identitätsnummern von dem Besitzer des Programmes übernommen, während die realen Nummern gleichbleiben.
- In Bezug auf Zugriffe im Dateisystem sind die effektiven Nummern von Belang.

Zugriffsschutz bei Dateien

- Zu jeder Inode gehören die elementaren Zugriffsrechte die Lese-, Schreib- und Ausführungsrechte angeben für den Besitzer, die Gruppe und den Rest der Welt.
- Wenn die effektive Benutzernummer die 0 ist, dann ist alles erlaubt (Super-User-Privilegien).
- Falls die effektive Benutzernummer mit der der Datei übereinstimmt, dann sind die Zugriffsrechte für den Besitzer relevant.
- Falls nur die effektive Gruppennummer mit der Gruppenzugehörigkeit der Datei übereinstimmt, dann sind die Zugriffsrechte für die Gruppe relevant.
- Andernfalls gelten die Zugriffsrechte für den Rest der Welt.

Zugriffsschutz bei Verzeichnissen

- Lese-, Schreib- und Ausführungsrechte haben bei Verzeichnissen besondere Bedeutungen.
- Das Leserecht gibt die Möglichkeit, das Verzeichnis mit *opendir* und *readdir* anzusehen, aber noch *nicht* das Recht, *stat* für eine darin enthaltene Datei aufzurufen.
- Das Ausführungsrecht lässt die Verwendung des Verzeichnisses in einem Pfad zu, der an einem Systemaufruf weitergereicht wird.
- Das Schreibrecht gewährt die Möglichkeit, Dateien in dem Verzeichnis zu entfernen (*unlink*), umzutaufen (*rename*) oder neu anzulegen. Das Ausführungsrecht ist aber eine Voraussetzung dafür.

Weitere Bits in *st_mode*

- Zusätzlich gibt es noch drei weitere Bits:

  Set-UID-Bit Bei einer Ausführung wird die effektive Benutzernummer (UID) gesetzt auf die Benutzernummer des Besitzers.

  Set-GID-Bit Entsprechend wird auch die effektive Gruppennummer (GID) gesetzt. Bei Verzeichnissen bedeutet dies, dass neu angelegte Dateien die Gruppe des Verzeichnisses erben.

  Sticky-Bit Programme mit dem Sticky-Bit bleiben im Speicher. Verzeichnisse mit diesem Bit schränken die Schreibrechte für fremde Dateien ein – nützlich für gemeinsam genutzte Verzeichnisse wie etwa */tmp*.

Das Ein- und Ausgabe-System

- Die Systemschnittstelle für Ein- und Ausgabe dient primär zwei Zielen:
  - Sie sollte möglichst gut abstrahieren und somit Anwendungen befreien von Hardware-Abhängigkeiten und bis zu einem gewissen Umfange auch von den Besonderheiten eines Dateisystems.
  - Sie sollte eine höchstmögliche Effizienz erlauben bis hin zum Verzicht auf jegliche zusätzliche Kopieraktionen zwischen dem Betriebssystem und dem Adressraum des Prozesses (*zero copy*).

Dateideskriptoren

- Dateideskriptoren sind ganzzahlige Werte aus dem Bereich $[0, N-1]$, wobei N typischerweise eine Zweierpotenz ist (etwa 512 oder 1024).
- Dateideskriptoren werden innerhalb des Betriebssystems als Indizes für Verwaltungstabellen verwendet.
- Dateideskriptoren referenzieren somit vom Betriebssystem verwaltete Objekte.
- Für jeden Prozess verwaltet das System eine eigene Tabelle. Entsprechend kann beispielsweise der Dateideskriptor 2 bei zwei Prozessen mit völlig unterschiedlichen Objekten verbunden sein.
- Die so referenzierten Objekte sind typischerweise Dateien, können aber auch Netzwerkverbindungen, Verbindungen zu anderen Prozessen, Geräte und Speicherbereiche sein.
- In C wird für Dateideskriptoren der Datentyp **int** verwendet.

Wieviel Dateideskriptoren gibt es?

openmax.c

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

int main() {
   long maxfds = sysconf(_SC_OPEN_MAX);
   printf("maximal number of open file descriptors: %ld\n", maxfds);
}
```

- Der Systemaufruf *sysconf* erlaubt die Abfrage zahlreicher Größen, von denen auch einige erst zur Laufzeit festliegen.
- Der Parameter *_SC_OPEN_MAX* liefert die maximale Zahl offener Dateien und die damit die Größe der systeminternen Tabelle der Objekte für diesen Prozess.

```
doolin$ gcc -Wall -std=c99 openmax.c
doolin$ a.out
maximal number of open file descriptors: 512
doolin$
```

Vererbung von Dateideskriptoren

- Wenn ein neuer Prozess erzeugt wird, dann wird die Tabelle mit den Dateideskriptoren kopiert.
- Entsprechend kann die Shell einige Dateideskriptoren für ein Programm, das sie startet, vorbereiten.
- Wenn nichts anderes spezifiziert wird, sind dies folgende Dateideskriptoren:

  0 Standard-Eingabe
  1 Standard-Ausgabe
  2 Standard-Fehlerausgabe

- Die Bourne-Shell und die von ihr abgeleiteten Shells erlauben das Öffnen und Schliessen beliebiger Dateideskriptoren. Folgendes Beispiel ruft a.out auf, wobei 0 geschlossen wird, 7 zum Schreiben geöffnet wird auf die Datei *out* und 10 zum Lesen für die Datei *in* eröffnet wird:

```
a.out 0<&- 7>out 10<in
```

Kopieren mit der stdio

scopy.c

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

int main(int argc, char* argv[]) {
   char* cmdname = argv[0];
   if (argc != 3) {
      fprintf(stderr, "Usage: %s infile outfile\n", cmdname);
      exit(1);
   }
   char* infile = argv[1]; char* outfile = argv[2];
   FILE* in = fopen(infile, "r"); if (!in) perror(infile), exit(1);
   FILE* out = fopen(outfile, "w"); if (!out) perror(outfile), exit(1);
   int ch;
   while ((ch = getc(in)) != EOF) {
      if (putc(ch, out) == EOF) perror(outfile), exit(1);
   }
   fclose(in);
   if (fclose(out) == EOF) perror(outfile), exit(1);
}
```

Kopieren mit Systemaufrufen

copy.c

```
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stralloc.h>
#include <string.h>
#include <unistd.h>

char* cmdname;
/* ... */

int main(int argc, char* argv[]) {
   cmdname = argv[0];
   if (argc != 3) {
      stralloc usage = {0};
      if (stralloc_copys(&usage, "Usage: ") &&
            stralloc_cats(&usage, cmdname) &&
            stralloc_cats(&usage, " infile outfile\n")) {
         write(2, usage.s, usage.len);
      }
      exit(1);
   }
   /* ... */
}
```

Kopieren mit Systemaufrufen

copy.c

```
char* infile = argv[1]; char* outfile = argv[2];

int infd = open(infile, O_RDONLY);
if (infd < 0) die(infile);
int outfd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
if (outfd < 0) die(outfile);
char buf[8192]; ssize_t nbytes;
while ((nbytes = read(infd, buf, sizeof buf)) > 0) {
   ssize_t count;
   for (ssize_t written = 0; written < nbytes; written += count) {
      count = write(outfd, buf + written, nbytes - written);
      if (count <= 0) die(outfile);
   }
}
if (nbytes < 0) die(infile);
close(infd);
if (close(outfd) < 0) die(outfile);
```

Das Eröffnen einer Datei

copy.c

```
int infd = open(infile, O_RDONLY);
if (infd < 0) die(infile);
int outfd = open(outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666);
if (outfd < 0) die(outfile);
```

- Mit dem Systemaufruf *open* kann eine Datei eröffnet werden. Im Erfolgsfalle wird ein (zuvor unbenutzter) Dateideskriptor zurückgeliefert.
- Der zweite Parameter gibt an, wie die Datei zu eröffnen ist. Hier können zahlreiche Werte mit einem bitweisen Oder verknüpft werden, wobei nicht jede Kombination sinnvoll ist. Eine Auswahl:

O_RDONLY	Nur zum Lesen eröffnen
O_WRONLY	Nur zum Schreiben eröffnen
O_RDWR	Zum Lesen und Schreiben eröffnen
O_CREAT	Datei neu anlegen, falls noch nicht existent
O_TRUNC	Datei auf Länge 0 bringen, falls existent

- Der optionale dritte Parameter wird nur hinzugefügt, falls bei dem zweiten Parameter *O_CREAT* mit angegeben wurde. Er legt die Zugriffsrechte fest. 0666 steht für rw-rw-rw.

Die Systemaufrufe read und write

copy.c

```
while ((nbytes = read(infd, buf, sizeof buf)) > 0) {
   ssize_t count;
   for (ssize_t written = 0; written < nbytes; written += count) {
      count = write(outfd, buf + written, nbytes - written);
      if (count <= 0) die(outfile);
   }
}
if (nbytes < 0) die(infile);
```

- Die Systemaufrufe *read* und *write* erhalten jeweils als Parameter, einen Dateideskriptor, einen Zeiger auf einen Puffer und eine Angabe, wieviel Bytes maximal zu transferieren sind.
- Grundsätzlich haben *read* und *write* die Freiheit, weniger Bytes zu übertragen als angegeben.
- Der Rückgabewert gibt die Zahl der übertragenen Bytes im Erfolgsfalle (immer positiv) oder ist gleich 0 (bei *read* steht dies für das Eingabeende) oder -1 bei Fehlern.

Das Schliessen eines Dateideskriptors

copy.c

```
close(infd);
if (close(outfd) < 0) die(outfile);
```

- Mit *close* können Dateideskriptoren geschlossen werden.
- Bei einem zuvor zum Schreiben geöffneten Dateideskriptor ist es sinnvoll, den Erfolg zu überprüfen, weil so noch am Ende aufgetretene Fehler erkannt werden können – auch wenn dies eher selten der Fall sein dürfte.

Kopieren mit Systemaufrufen

copy.c

```
void die(char* filename) {
   stralloc msg = {0};
   if (stralloc_copys(&msg, cmdname) &&
         stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, strerror(errno)) &&
         stralloc_cats(&msg, ": ") &&
         stralloc_cats(&msg, filename) &&
         stralloc_cats(&msg, "\n")) {
      write(2, msg.s, msg.len);
   }
   exit(1);
}
```

- *strerror* liefert die Fehlermeldung passend zu *errno*. Die bislang bekannte Funktion *perror* basiert auf *strerror*.

Kopieren mit mmap

mcopy.c

```
struct stat statbuf; if (fstat(infd, &statbuf) < 0) die(infile);
off_t nbytes = statbuf.st_size;
char* buf = (char*) mmap(0, nbytes, PROT_READ, MAP_SHARED, infd, 0);
if (buf == MAP_FAILED) die(infile);
ssize_t count;
for (ssize_t written = 0; written < nbytes; written += count) {
   count = write(outfd, buf + written, nbytes - written);
   if (count <= 0) die(outfile);
}
```

- Der Systemaufruf *mmap* (*memory map*) erlaubt es, den Inhalt des Puffer-Cache, der zu einer Datei gehört, direkt in den eigenen Adressraum zu legen.
- Auf diese Weise entfällt das Kopieren des Inhalts der zu kopierenden Datei in den Adressraum des Kopierprogramms.

Vergleich der Kopierprogramme

```
turing$ mkfile 10m 10m
turing$ time scopy 10m out && rm out

real    0m0.64s
user    0m0.59s
sys     0m0.05s
turing$ time copy 10m out && rm out

real    0m0.09s
user    0m0.01s
sys     0m0.08s
turing$ time mcopy 10m out && rm out

real    0m0.07s
user    0m0.00s
sys     0m0.07s
turing$
```