Betriebssysteme

Dateisysteme

Praktikum 8

Fachhochschule Bielefeld
Campus Minden
Studiengang Informatik

Beteiligte Personen:

Name	Matrikelnummer
Karsten Michael Tymann	1047529
Mirko Weidemann Kreitz	1048290
Oxana Zhurakovskaya	130157
Yuliia Dobranska	1093568

---

15. Juni 2016

Inhaltsverzeichnis

1 Aufgabe 1

Bearbeiten Sie die folgenden Aufgaben und protokollieren Sie Ihr Vorgehen mit- hilfe der Vorlage. Implementieren Sie ein C-Programm, das folgende Anforderungen erfüllt:

- Eine Datei wird zum Lesen geöffnet; anschließend wird zuerst die zweite Hälfte und dann die erste Hälfte des Dateiinhalts auf dem Bildschirm ausgegeben.
- Danach wird der Inhalt der Datei in eine neue Datei kopiert, wobei der Dateiname der Quell- und der Zieldatei dem Programm als Argument übergeben werden kann.
- Die letzten 10 Zeichen der ursprünglichen Datei werden ab der 11. Stelle der neuen Datei kopiert. Das Dateiende der neuen Datei soll jetzt nach den verschobenen Daten sein (also nach dem 21. Zeichen).
- Der Inhalt der Datei soll auf dem Bildschirm ausgegeben werden.

1.1 Vorbereitung

Für Testzwecke erstellen wir eine Datei mit Text (gefüllt mit generiertem text) *filetocopy* und *testtarget*

1.2 Durchführung

- Damit wir Dateinamen als Argumente an unser Programm übergeben können, definieren wir in der main Function die Parameter `main(int argc, char *argv[])`. Der Parameter `argc` beinhaltet die Anzahl der übergebenen Argumente + 1 (Programm Name) und `argv` beinhaltet alle übergebenen Argumente. Ab Index 1 können wir auf die Argument Liste zugreifen. Als erstes Argument erwarten wir den Pfad zur Quelldatei und als zweites Argument erwarten wir den Pfad zur Zieldatei. Die darauffolgenden Argumente werden ignoriert.
- wir prüfen ob genug Argumente übergeben wurden, wenn nicht, dann bricht das Programm ab.
- wenn alle benötigten Argumente vorhanden sind führt das Programm fort.
- wir definieren zwei Variablen, wo wir die Pfade für Quell- und Zieldatei speichern.

  ```
  char * quellpath;
  char * zielpath;
  ```

  Zudem definieren wir Variablen die den Quell- respektive den ZielFiledescriptor halten. Ein File Descriptor ist ein Handler um mittels In-und Output Methoden mit dieser File umzugehen. In C ist eine Abbildung als Integer üblich. Dabei geben negative Werte an, dass eine Fehlerbedingung vorliegt.

  ```
  int filedescriptorQuelle;
  int filedescriptorZiel;
  ```

Diese werden in den Methoden `read`, `write`, `close` benutzt um die Datei mit der gearbeitet wird zu spezifizieren.

- Für beide Dateien definieren wir jeweils eine Variable Modus. Dies sorgt dafür dass die Quelle im Lesemodus geöffnet wird. Das Ziel soll sowohl im Lese- als auch im Schreibmodus geöffnet werden.

```
mode_t modeQuelle = S_IRUSR;
mode_t modeZiel = S_IRUSR | S_IWUSR;
```

- Für Beide Dateien definieren wir jeweils Variable, die Länge der Datei speichert

```
long long quellsize;
long long targetsize;
```

- Als nächsten Schritt überprüfen wir die Anzahl der übergebenen Argumente. Wurden nicht genügend Parameter übergeben, also nicht Quelle und Ziel, so soll das Programm abbrechen. Zuvor übermitteln wir mittels `perror` den Fehler über den `stderr` Output Stream.

```
if (argc <= 2){
        perror("Error: too few arguments");
        exit(EXIT_FAILURE);
}
```

- Nun ermitteln wir den absoluten Pfad anhand der Eingaben des Users. Konnte der Pfad nicht aufgelöst werden, also z.b. wenn die Datei nicht gefunden werden konnte, so müssen wir das Programm abbrechen. Dazu prüfen wir den Rückgabeparameter von `realpath`. Ist dieser NULL, so existiert die Datei nicht und wir geben den Fehler aus. Hierbei ist anzumerken dass der Eingabeparameter eine beliebige Fehleranmerkung ist. `perror` gibt nämlich den mit dem in `errno` hinterlegten Fehlercode aus.

```
if(realpath(argv[1], quellpath) == NULL){
        perror("Error argv[1]:");
        exit(EXIT_FAILURE);
}
```

- Der Ziel Pfad wird auf gleiche Weise aufgelöst wie Quellpfad.

```
if(realpath(argv[2], zielpath) == NULL){
        perror("Error argv[2]:");
        exit(EXIT_FAILURE);
}
```

- Als nächstes ermitteln wir die Dateigrößen für die Quell- und Zieldatei. Dies geschieht in dem wir eine in `<sys/stat.h>` definierte Struktur initialisieren `quellstat` und hier mit Hilfe der Funktion `stat` die Informationen über die Datei speichern. Anschließend speichern wir in der Variable `quellsize` die in `quellstat` enthaltene Information über die Dateigröße `st_size`. Auf gleiche Weise ermitteln wir auch die Größe des Zieldatei.

```
struct stat quellstat;
stat(quellpath, &quellstat);
quellsize = (long long) quellstat.st_size;

struct stat zielstat;
stat(zielpath, &zielstat);
targetsize = (long long) zielstat.st_size;
```

- Wir legen eine Variable `char bufQuelle[quellsize];` an um den Dateinhalt aus der Quelle hier zu speichern und weiter zu verwenden.
- Anschließend können wir beide Dateien mit der function `open` öffnen. Die function liefert einen filedescriptor (um den Filestream zu verwalten), wenn die Datei geöffnet werden konnte, sonst liefert sie den Wert -1. Das nutzen wir in der if abfrage um Fehler abzufangen und abzugeben. Der Filestream wird im Read-Only Modus geöffnet, mit den oben beschriebenen Zugriffsrechten.

```
if ((filedescriptorQuelle = open(quellpath,
 O_RDONLY, modeQuelle)) == -1) {
perror("Error by opening quellpath:");
exit(1);
}
```

- Das selbe wird für das Ziel getan. Allerdings öffnen wir hier die Datei im Lese- und Schreibmodus bzw. erzeugen die Datei wenn sie noch nicht existiert. Die Modi der Operationen sind bereits oben beschrieben worden.

```
if ((filedescriptorZiel = open(zielpath, O_RDWR
| O_CREAT, modeZiel))
        == -1) {
perror("Error by opening zielpath:");
exit(1);
}
```

- Wir definieren eine Funktion `backupFileContent`, die uns hilft die Dateiinhalte in bufQuelle zu speichern
  - Die Methode nimmt 3 Argumente an. Der Integer descriptor gibt den übergebenen File Descriptor an. Der char Pointer buffer wird mit den gelesenen Daten beschrieben. Der Integer size gibt die Länge an die aus dem File gelesen wird an. In unserem Fall ist dies immer die gesamte Länge des Files. Im Fehlerfall, also bei einem negativen Wert, wird ein Fehler ausgegeben. Nach dem Methodenaufruf wurde somit die übergebene Datei in ein char Array gelesen.

```
void backupFileContent(int descriptor, char *buffer, int size) {
size_t error = read(descriptor, buffer, size);
if (error == -1) {
perror("Inhalt konnte nicht gelesen werden bzw. Datei ist leer.");
```

```
}
}
```

- Nach dem Methodenaufruf wurde die bufQuelle mit dem Inhalt der gelesenen Datei gefüllt. `backupFileContent(filedescriptorQuelle, bufQuelle, quellsize);`
- Nun erfolgt die Ausgabe der Datei. Dabei soll zunächst die zweite Hälfte und dann die erste Hälfte ausgegeben werden. Hierzu definieren wir eine Funktion die diese Ausgabe übernimmt. `printHalfFile()`
  - Die Methode nimmt 2 Eingabeparameter. Der Integer filesize hält die Größe der auszugebenden Dateien. Der char Buffer hält den auszugebenden Text.

    Innerhalb der Methode wird zunächst die Hälfte der Größe ermittelt. Hierbei ist es möglich dass der Buffer einer ungerade Anzahl an Zeichen hat. In diesem Fall, damit wir kein Zeichen verlieren, müssen wir mittels `ceil` das Ergebnis aufrunden (aus <math.h>). Dieser Schritt wird nur für die zweite Hälfte getan. Die Hälfte, ohne runden, wird nun für die erste Hälfte ausgeführt. Mit diesen zwei Größen erstellen wir uns 2 char Arrays die den Text halten werden.

    Mittels `strncpy` speichern wir uns nun die zweite Hälfte aus der bufQuelle. Hierbei startet der Pointer, der Kopierschritt, an der mittleren Stelle (+ firsthalf) für die Länge von secondhalf (den Rest des Strings). Ziel ist bufpartone. Mittels `write` geben wir den Text nun aus. Die Konstante `OUT` repräsentiert den Wert 1, welcher den `stdout` File Stream darstellt. Auf diesen Stream möchten wir schreiben (dies sind die zu sehenden Ausgaben). bufpartone gibt den zu schreibenden String aus und secondhalf die Länge. Für den bufparttwo wird ähnliches getan, nur das beim Kopieren des Strings der Zeiger an der ersten Stelle anfängt und bis zur Mitte läuft. Auch dies wird dann ausgegeben.

    ```
    void printHalfFile(int filesize , char *bufQuelle){
    int secondhalf = ceil(filesize/2);
    int firsthalf = filesize/2;
    char bufpartone[secondhalf];
    char bufparttwo[firsthalf];

    strncpy(bufpartone, bufQuelle + firsthalf, secondhalf);
    write(OUT, bufpartone, secondhalf);
    strncpy(bufparttwo, bufQuelle, firsthalf);
    write(OUT, bufparttwo, firsthalf);
    }
    ```

- Als nächstes definieren wir eine Funktion `copyFile` um die 10 letzten Zeichen aus der Quelldatei zu lesen und in die Ziel Datei ab der 11. Stelle zu schreiben.
  - die Funktion nimmt als erstes Argument ein Filedescriptor für das Ziel Document, als zweiten Argument die Größe der ZielDatei, als dritten Argument die Größe des Quelldatei und als viertes Argument einen pointer auf ein chararray, das den Inhalt der Quelldatei beinhalten soll

– mit der Funktion `lseek` verschieben wir den cursor an die gewünschte Stelle. Dabei gibt die Zahl 10 die Stellen an um die wir den cursor verschieben möchten, 10 laut Aufgabe. Das `SEEK_SET` gibt an ab welcher Position wir den cursor verschieben möchten. `SEEK_SET` bezieht sich dabei auf den Dateibeginn. Wenn die Verschiebung nicht erfolgreich war, wird ein Wert -1 zurückgeliefert. Dies nutzen wir in der if-Abfrage und brechen die funktion ab wenn der cursor nicht verschoben werden konnte.

– Nachdem der cursor in die Zieldatei verschoben wurde legen wir ein chararray an,

  `char bufeleventoend[targetsize - 10];`,

  dass alle Zeichen des Ziels bis auf die ersten 10 halten soll.

– Mittels `backupFileContent` schreiben wir uns die Zeichen 11-Ende in das Array bufeleventoend. Da der cursor an Stelle 11 geschoben ist liest der Filedescriptor auch erst ab dieser Stelle. Die Länge können wir um 10 dekrementieren. Das Array hält nun alle Zeichen 10-Ende.

```
int copyFile(int zielDiscr,
int targetsize, int  quellsize,
char *bufQuelle) {

if (lseek(zielDiscr, 10, SEEK_SET) == -1){
perror("Couldnt set pointer ");
return -1;
}

char bufeleventoend[targetsize - 10];
backupFileContent(zielDiscr, bufeleventoend, targetsize - 10);
...
```

– Zunächst legen wir ein Array für die letzten 10 Zeichen der Quelle an. Dieses befüllen wir mittels `strncpy`. Dazu fangen wir das Kopieren an der 11-letzten Stelle für 10 Zeichen an. Das Array hält nun die letzten 10 Zeichen.

  Nun können wir den cursor im Ziel wieder auf Stelle 10 setzen. Ist dieser Schritt geglückt so befüllen wir die Datei mit den restlichen Inhalt.

  Mittels `write` wird nun ab der Stelle des Cursors zunächst das Array lastchars in die Datei des Zieldescriptors geschrieben. Anschließend schreiben wir die Zeichen aus bufeleventoend, die den Rest der Zeichen der Zieldatei halten wieder in die Zieldatei ein. Dieser Schritt ist nötig da der erste `write` Vorgang die alten Zeichen überschrieben hat.

  Nun ist die Datei wie gewünscht zusammengestellt worden.

```
...
/* In Zieldatei schreiben */
ssize_t bytes_written;
```

```
char lastchars[11];

strncpy(lastchars, bufQuelle + quellsize - 11, 10);

if (lseek(zielDiscr, 10, SEEK_SET) == -1){
perror("Couldnt set pointer ");
return -1;
}

bytes_written = write(zielDiscr, lastchars, 10);
bytes_written = write(zielDiscr, bufeleventoend,
sizeof(bufeleventoend));
}
```

- in `main` rufen wir die beschriebene Funktion `printHalfFile` auf.
- in `main` rufen wir die beschriebene Funktion `copyFile`
- Nachdem die Funktionen ihre Aufgaben erfüllt haben, müssen wir die geöffneten Dateien schließen. Dabei nutzen wir die function `close`, der wir einen filedescriptor übergeben. Die function liefert einen Wert = 0 zurück wenn die Datei erfolgreich geschlossen wurde. Sonst geben wir einen Fehler aus.

```
if ((close(filedescriptorQuelle)) == 0
 && close(filedescriptorZiel) == 0) {
  printf("Dateien wurden geschlossen!\n");
} else {
  perror("Datei konnte nicht geschlossen werden!\n");
}
```

1.3 Fazit