Grundlagen der Betriebssysteme - Labor

Versuch: Kommandozeile

Dieser Versuch ist eine Einführung in die Kommandozeile (engl. "Shell"). Vorab wird rekapituliert, was man über Dateien und Verzeichnisbäume hoffentlich schon weiß bzw. spätestens jetzt lernen sollte. In der ersten Aufgabe 0 (Informatiker zählen auch schon mal ab 0) werden einige Befehle vorgestellt und durch Übungen vertieft. In Aufgabe 1 folgt danach die Programmierung einer eigenen, minimalen Kommandoshell, um das eigentlich relativ einfache Grundprinzip zu erarbeiten. In Aufgabe 2 wird diese eigene Shell ein bisschen erweitert. Zuletzt soll in Aufgabe 3 ein Programm erstellt werden, dass geeignet ist von einer Kommandozeile aufgerufen zu werden, und somit als Musterfall geeignet ist, wie man die Kommandosprache einer Shell erweitern kann.

Intermezzo: Bäume, Verzeichnisse, Dateien, Textdateien

Bei modernen Betriebssystemen muss ein Anwender (fast) nicht mehr Wissen, was ein Prozessor ist oder wofür der Hauptspeicher gut ist. Ebenso muss man nicht mit Sektoren, Blöcken oder Clustern

auf Massenspeichern arbeiten. Geblieben ist bis heute, dass man mit Dateien und Verzeichnissen auf Massenspeichern umgehen können sollte. In diesem Skript wird vorausgesetzt, dass grundlegendes Wissen darüber vorhanden ist.

Dennoch wird hier kurz dargelegt, was für den Versuch bzw. im Praktikum bekannt sein sollte. Dabei geht es evtl. ein bisschen tiefer als es für den reinen Endanwender nötig ist: es ist das Wissen, dass man als Student der Informatik oder eines nahen Studienganges haben sollte.

Überblick

Auf jedem Medium (Festplatte, SSD, USB-Stick, ...) wird ein Verzeichnisbaum verwaltet. Das Wurzelverzeichnis heißt unter Unix "/", unter Windows "\". Darin können Verzeichnisse mit vom Nutzer gewählten Namen angelegt werden, und in diesen wieder, so dass ein Verzeichnisbaum vorliegt. In jedem Verzeichnis, nicht nur in den Blättern des Baums, können Dateien mit ebenfalls vom Nutzer gewählten Namen liegen. Um ein Baumelement zu benennen nutzt man folgende Syntax:

```
/home/ms123/dingsda
```

Das erste Zeichen ist der Name des Wurzelverzeichnisses: "/". Danach folgen Namen von jeweils darin liegenden Verzeichnissen, getrennt durch "/", die ab jetzt, also überall außer ganz am Anfang, als Namenstrenner dienen. Windows verwendet als Namentrenner auch wieder "\" statt "/". Das Beispiel liest man also so, dass im Wurzelverzeichnis "/" ein Unterverzeichnis "home" ist, darin ein Unterverzeichnis "ms123" und darin etwas, das "dingsda" heißt. Der Syntax kann man nicht ansehen, ob dingsda ein Verzeichnis oder eine Datei ist. Sicher ist aber, dass nur der letzte Name ein Dateiname sein kann.

Eine leicht abweichende Syntax spart viel Tipparbeit:

```
dingsda/irgendwas
```

Der fehlende "/" am Anfang besagt, dass es sich um einen sogenannten relativen Pfad handelt, im Gegensatz zu Pfaden, die mit "/" beginnen, die absolute Pfade genannt werden. Relative Pfade werden von einer im System liegenden Variablen vorne ergänzt zu einem absoluten Pfad. Wäre diese Systemvariable beispielsweise "/a/b", dann würde die genannte Pfadangabe letztlich zu "/a/b/dingsda/irgendws" expandiert. Die genannte Systemvariable existiert einmal pro Prozeß, enthält immer einen absoluten Pfad, und ist die, die auf der Kommandozeile mit dem Kommando "cd" (s.u.) modifiziert werden kann.

Bäume

"Pro Medium existiert ein Baum" wurde oben postuliert. Dies ist nicht falsch, verdient aber ein bisschen Nacharbeit:

- Viele Medien werden per Software unterteilt in sogenannte "Partitionen". Jede Partition wird wie ein eigenes Medium behandelt, enthält mithin einen eigenen Baum. Dadurch kann eine SSD im System wie mehrere Festplatten aussehen.
- Unter Windows erhalten die Bäume Namen, genauer sogenannte Laufwerksbuchstaben, welche dem Dateinamen mit Doppelpunkt vorangestellt werden, also z.B.
  c:\users\ms123
- Unter Unix wird hingegen der Baum eines Mediums in den Hauptbaum "eingeklinkt", indem ein bestehendes Verzeichnis überlagert wird vom Wurzelverzeichnis eines anderen Mediums. Dadurch kann ein Name wie z.B.:
  /a/b/c/d
  irgendwo bei a (oder b, c, d) das Medium wechseln, ohne dass davon jemand etwas bemerkt oder Wissen müßte. So könnte das Verzeichnis /a/b des Hauptbaums überlagert sein, so dass der Inhalt von /a/b selbst nicht mehr erreichbar ist. Stattdessen wird unter /a/b das Wurzelverzeichnis eines anderen Mediums sichtbar, mit seinen Dateien und Unterverzeichnissen.
- Netzwerklaufwerke simulieren über das Netzwerk Verzeichnisbäume, als ob sie auf Medien vorhanden wären, die direkt im Computer sind. In den Pools der SGI findet man als Nutzer sein Home-Laufwerk unter H:, unter Linux unter /home/<username>. Beide Pfade finden das Heimatverzeichnis eines Nutzers auf demselben Server, so dass alle Nutzerdateien unter beiden Betriebssystemen identisch sichtbar sind.
- Vielerlei andere Mechanismen existieren um ebenfalls scheinbar lokale Verzeichnisbäume anzubieten. Die Cloud, unter Windows sogenannte Bibliotheken (Libraries) um nur zwei zu nennen.

Verzeichnisse

Verzeichnisse können von den Nutzern frei benutzt werden. Lediglich Konventionen regeln hier die Nutzung. Unter Linux gibt es Beispielsweise schon immer "/etc" für Konfigurationsdateien und das Verzeichnis "/var/log" für Protokolldateien. Auf allen Unix-Maschinen der SGI werden Sie Ihr Heimatverzeichnis unter "/home/<accountname>" finden.
Windows benutzt beispielsweise "c:\windows" und "c:\program files". Nutzt man die Datei-Ansicht des Explorers, so bekommt man links den Verzeichnisbaum, rechts den Inhalt eines Verzeichnisses angezeigt.

Dateien

Bäume, Verzeichnisse sind Verwaltungsstrukturen des Betriebssystems und auch für den Endanwender nützlich um seine Dateien logisch zu gruppieren. Aber erst in diesen Dateien liegen die reinen Nutzerdaten.

Eine Datei ist heutzutage ein Container für eine Byte-Reihe, also eine geordnete Sequenz von Bytes. Eine Datei liegt unter einem (Datei-)Namen in einem Verzeichnis.

Dateien haben meist Endungen, die auf den Typ des Dateininhalts hindeuten: ".jpg" ist wohl eine Bilddatei. Allerdings sollte man immer im Hinterkopf behalten, dass es keinen Mechnismus gibt, der diese sinnvolle Zuordnung erzwingt. Das es überhaupt klappt liegt wohl daran, dass niemand sich selbst gerne reinlegt, indem er "falsche" Dateiendungen wählt.

Unter Windows versucht das Betriebssystem die Endungen mit Programmen zu verknüpfen, die hoffentlich mit dem Dateininhalt etwas anfangen können. Unter Unix ist es zwar auch üblich typisiserende Dateiendungen zu benutzen, jedoch ist es hier weitgehend den Anwendungen überlassen damit zu arbeiten. Grafische Oberflächen tun dies unter Unix ähnlich wie Windows.

Textdateien

Dateiformate kann man inzwischen reichlich z.B. in Wikipedia dokumentiert finden. An dieser Stelle sollen Text-Dateien vertieft werden.

Textdateien sind ganz einfach Dateien, deren Inhalt als Text gelesen werden kann. Die Zeichenkodierung ist meist ASCII oder eine der Kodierungen, die ASCII als Untergruppe enthalten. Der kommende Standard ist hierbei wohl UTF-8.

Im ASCII-Satz sind als druckbare Zeichen alle 26 Buchstaben des Alphabets als Klein- und als Großbuchstaben enthalten, alle Ziffern und eine ganze Menge Sonderzeichen. Dazu kommen 33 Steuerzeichen, die nicht direkt einem Zeichen auf Papier entsprechen, sondern spezielle Funktionen haben.

Von diesen speziellen Funktionen werden heute fast nur noch die Folgenden verwendet:

Zeilenende-Zeichen. Diese werden leider in drei verschiedenen Formen angetroffen:

Windows benutzt die zwei Zeichen CR,LF (Kodierungen 13 und 10) hintereinander um ein Zeilenende zu markieren.

Unix nutzt nur das Zeichen LF dafür.

MacIntoshs benutzen nur das Zeichen CR.

Das Tabulator-Zeichen, Kodierung 9 steht meist für einige Leerzeichen. Leider ist nicht festgelegt, für wie viele.

Da der ASCII-Satz nur sieben Bit benötigt wurden viele Erweiterungen erfunden, um einerseits das achte Bit eines Bytes zu nutzen, andererseits Mängel zu beheben. Beispielsweise benötigt man im Deutschen die Umlaute "Ä" bis "ü" und das "ß". Alle Kodierungen, die ASCII auf 256 Zeichen erweitern, haben aber den Nachteil, dass sie nicht vollständig sind, d.h., immer noch nicht jedes beliebige Zeichen kodieren können.

Dafür wurde Unicode erfunden, das sich in der Form UTF-8 langsam durchzusetzen scheint. UTF-8 nutzt einerseits ASCII, und andererseits die Bytes mit dem höchstwertigen Bit gesetzt, um mehrere Bytes für ein Zeichen zu kodieren. UTF-8 Sequenzen reichen von einem Byte (ASCII) bis vier Byte Länge und können den gesamten Unicode abbilden. Unicode enthält sehr viele Zeichen, und wurde mit dem Ziel geschaffen alle Zeichen aller Sprachen aufzunehmen, incl. z.B. von ca. 50000 chinesischen Schriftzeichen. Als Programmierer sollte man bedenken, dass mit UTF-8 bzw. Unicode ein Zeichen nicht mehr in ein Byte passt, sondern bis zu 32 Bit Platz benötigt.

Textdateien werden für vieles verwendet:

- Einfach nur als Text, wobei zur Formattierung nicht viel mehr als Zeilenenden zur Verfügung stehen (vgl. oben).
- Als Quelltext für eine Programmiersprache. Praktisch jede Programmiersprache baut auf Textdateien auf.
- Quelltext für formatierten Text, z.B. RTF, HTML, TeX/LaTeX. Zur "hübschen" Anzeige benötigt man ein Hilfsprogramm (hier: Word, Web-Browser, TeX/LaTeX Compiler).
- Tabellen als "Comma Separated Values", kurz CSV. Hiermit werden Datenbanken migriert, Daten zwischen Programmen ausgetauscht, und vieles mehr. Zur Formattierung der Tabelle werden waagerecht Zeilenenden benutzt, für die Spalten dem Namen CSV nach Kommas. Allerdings wird statt dem Komma oft ein beliebiges anderes Zeichen benutzt, behält aber die Bezeichnung "CSV" bei.
- Datencontainer für komplexere Daten: XML, JSON. Diese Formate enthalten Daten, die zwar strukturiert sind, aber dennoch wohl nur von Programmen eines Projekts ausgetauscht werden. Immerhin kann man als Mensch meist halbwegs erschließen, welche Daten vorliegen und auch deren Strruktur.

Eine Mischform von obiger Einteilung ist PDF (bzw. auch die Vor-Form Postscript). Dies sind Programmquelltexte in einer Programmiersprache, die vor allem zum Textsatz geeignet ist. Allerdings sind in solchen Dateien oft auch noch Fonts oder Bilder als Binärdateien eingbettet, so dass es oft keine reinen Textdateien mehr sind.

5

Übrigens ist das Format ".doc" von Word keine Textdatei. Doc-Dateien enthalten binäre Daten, deren Form nur Word versteht, und evtl. einige andere Programme, die sich Mühe geben dieses Format ebenfalls lesen zu können.

Ein wichtiger Grund auch heute noch Textdateien zu nutzen ist, dass der Inhalt von einem Menschen gelesen, und oft auch direkt leidlich verstanden werden kann. Deshalb gibt es auch Programme, die "Editor" genannt werden, mit denen man als Mensch solche Dateien nicht nur anschauen sondern sogar ändern kann. Einige davon sind z.B. Notepad, VI, Emacs, Nano, Notepad++, ... Auch viele IDEs enthalten einen Editor zum Bearbeiten von Quelltexten.

Aufgabe 0: Mit einer Shell arbeiten

In diesem Versuchsteil soll zu Übungszwecken mit einer Shell gearbeitet werden. Zwar gibt es nicht "die" Shell, sondern eine ganze Reihe, jedoch hat sich in der Unix-Welt ein gewisser Standard durchgesetzt, an den sich alle Shells halten. In dieser Aufgabe werden die Unterschiede von Unix-Shells nicht thematisiert.

Zuerst werden einige Kommandos für die Shell vorgestellt, um mit Verzeichnissen und Dateien zu arbeiten, mit anschließender Übung in Aufgabe 0.1.

Der darauf folgende Abschnitt 0.2 stellt Aufgaben, die über die Arbeit mit Dateien hinausgehen. Das Ziel ist hier, zu zeigen, dass man mit der Kommandozeile vielerlei Aufgaben erledigen kann, quasi ein "schnuppern", warum die Kommandozeile wohl mächtiger als jede Mausoberfläche ist.

Der letzte Abschnitt konzentriert sich darauf, wie man mit der Kommandozeile Programme entwickelt. Dies ist gleichzeitig die Grundlage, wie man weitere Kommandos "erschaffen" kann. Nutzen Sie die Chance zu lernen, was eine IDE vor dem Programmierer verbirgt.

Erste Befehle der Kommandozeile

- **login**
  "login" ist ein Kommando, das man immer wieder benutzt, ohne es explizit aufzurufen: es läuft beim einloggen ab. Wenn man sich dafür interessiert, was es tut: "man login" hilft ("man" kommt weiter unten).
- **logout, exit**
  Zum Beenden der Shell. Sollte man immer tun, wenn man geht, damit niemand anderes Zugriff auf die eigenen Dateien bekommt. Auch schlimmeres geht, alles mit der "Unterschrift" desjenigen, der noch eingeloggt ist. Der Unterschied zwischen logout und exit ist, dass logout ein eigenständiges Programm ist, dass weiß, wie es sich selbst und die shell beendet. exit ist hingegen die Anweisung direkt an die shell, sich selbst zu beenden.

- **man <Kommando>**
  Damit kann man zu praktisch jedem Kommando Hilfe erhalten. Am Anfang sind die man-pages (Computer-Jargon für die angezeigten Hilfe-Seiten) etwas sperrig, aber das liegt daran, dass sie als Referenz und nicht als Tutorial gedacht sind. Es wird alles in kürzester Form erklärt. Tipp: am Anfang alles lesen, und alles ignorieren, was man nicht versteht. Mit der Zeit "springt" man zu der Stelle, die einen Interessiert, und über die Jahre versteht man immer mehr. (Ja, das dauert wirklich Jahre.)
- Viele Kommandos kennen heutzutage auch die Option "--help". Es ergibt typisch noch weniger Hilfetext als die man-page des Kommandos. Wichtigster Vorteil von "--help" ist, dass es auch dann funktioniert, wenn die man pages nicht installiert wurden.
- Um ein Kommando für eine neue Aufgabe zu finden empfiehlt es sich jemanden zu fragen, der mehr Erfahrung hat. Nicht ganz so gut: jede Internet-Suchmaschine. Beides hilft auch, wenn eine man-page unverständlich bleibt. (Fragen Sie die Tutoren!)
- **echo ...**
  echo gibt alles auf der Kommandozeile Mitgegebene wieder aus. "echo Hello, World" gibt "Hello, World" aus. Das Kommando wird hier erwähnt, weil man damit die shell gut erforschen kann, und es deshalb unten immer wieder benutzt wird.

Mit Verzeichnissen & Dateien arbeiten

- **ls bzw. ll <name>**
  Zum Anzeigen von Informationen über <name>. Ist <name> eine Datei wird genau zu dieser Datei Information ausgegeben, bei Verzeichnissen alle Namen, die darin vorkommen. Läßt man <name> weg, so werden alle Namen im aktuellen Verzeichnis angezeigt.
  "ll" ist Standard für "ls -l", also ls mit der Option -l: es zeigt mehr Informationen als nur den Namen an.
- **mkdir <neuer_verzeichnisname>**
  legt ein neues Verzeichnis an. Es gibt auch die Umkehrung "rmdir", aber die braucht man nicht, weil "rm -r <Verzeichnisname>" das auch kann, und vor allem auch dann, wenn das Verzeichnis vorher noch nicht leer ist.
- **cd <verzeichnisname>**
  Setzt die Systemvariable, die vor relativen Pfadnamen ergänzt wird. "cd" akzeptiert absolute Pfade, aber auch relative: dann wird der bisherige Inhalt der Systemvariablen ergänzt. Die Systemvariable wird meist im Prompt der Kommandozeile mit angezeigt, weshalb das Kommando "pwd" zum Anzeigen fast unbekannt ist.

- **cp <von> <nach>**
  cp ist das Kurzwort für copy. Kopieren einer Datei von → nach. Mit der Option -r gehen auch Verzeichnisbäume.
- **mv <von> <nach>**
  mv ist das Kurzwort für move. Verschieben einer Datei oder eines Verzeichnisses (samt Inhalt) von → nach.

Da mv auch von einem Medium zum Anderen verschieben kann denkt man sich am besten, dass es eine Kopie erzeugt (als "nach") und dann das Original ("von") löscht. Diese Vorstellung stimmt immer, jedoch wird "mv" oft Vereinfachungen finden und nutzen.
Es gibt auch "rename", aber das wird praktisch nie benutzt.

- **rm <name>**
  rm ist das Kurzwort für remove. Löscht die Datei name. Wenn name ein Verzeichnis ist, dann muss man die Option "-r" mit angeben: dann wird das Verzeichnis mit seinem gesamten Inhalt gelöscht. Am Anfang empfiehlt sich die Option "-i" (auch mit -r zusammen): "rm -i -r <name>" fragt vor jeder einzelnen Löschoperation nach!

Inhalt von Dateien anschauen oder bearbeiten

Alle Kommandos hier funktionieren mit allen Dateitypen, ergeben aber meist nur Sinn, wenn man sie auf Textdateien anwendet!

Außerdem taucht der Begriff "sdtout" auf. Im Moment einfach lesen als: die Ausgabe des Programms. Mehr Details werden später im Versuch Thema sein.

- **cat <namen>**
  cat ist das Kurzwort für concatenation. Kopiert alle namen, also auch mehrere, auf "stdout". Dadurch sind alle Dateiinhalte auf stdout ein Datenstrom, quasi eine Datei. Daher der Name des Kommando concatenation: (dt.) verketten.
  Oft wird es nur benutzt, um nur einen Datei auf stdout zu kopieren: "cat datei" zeigt den Inhalt von datei auf dem Bildschirm an, solange stdout nicht umgelenkt ist. Es soll langjährige Kommandozeilenbenutzer geben, die nicht (mehr) Wissen, dass cat eigentlich zum Verketten da ist.
- **less <name>**
  Um eine Datei auf dem Bildschirm anzuschauen empfiehlt sich eher less als cat. less zeigt dan Anfang der Datei an, dann kann man vorwärts und rückwärts Blättern, suchen und noch einiges mehr. less nimmt auch Daten von einer Pipe entgegen, also von stdout.
- **nano, vi, emacs, ...**
  Will man einen Datei nicht nur anschauen sondern ändern empfiehlt sich ein Editor. Das sind

Programme, die eine Datei einlesen, interaktiv vom Benutzer modifizieren, und auf Anweisung wieder zurückschreiben (und dabei den vorigen Zustand der Datei zerstören). Anfangs empfiehlt sich etwas einfaches, wie nano mit Fähigkeiten etwa wie notepad unter Windows. Braucht man mehr sollte man irgendwann die Hürde nehmen und einen leistungsfähigeren Editor lernen. vi ist recht sperrig zu bedienen, aber praktisch unter jedem Unix verfügbar. Emacs ist freundlicher zu Einsteigern, aber oft nicht verfügbar. Es gibt aber noch beliebig viele andere Editoren.

- **sed, ...**
  sed ist das Kurzwort für stream editor. Er sei hier stellvertretend erwähnt für eine Klasse von Editoren, die eher nicht interaktiv gedacht sind, sondern zum Einsatz in Skripten.

Nicht mehr ganz so essentiell, aber zur professionellen Nutzung der Kommandozeile wichtig:

- **grep <Suchstring> <namen>**
  Durchsucht alle genannten Dateien (<namen>) nach dem Suchstring und gibt die Trefferzeilen aus. Aber: der Suchstring ist eine sogenannte "regular expression". Diese werden hier nicht vertieft. Immerhin: Buchstaben und Ziffern stehen für sich selbst in regulären Ausdrücken. "grep hello meine_datei.txt" findet alle Vorkommen von "hello" in der Datei.
- **sort <namen>**
  Sortiert alle Zeilen aller angegebenen Dateien und gibt das Ergebnis auf stdout aus. "sort a b c > d" liest also die drei Dateien a, b und c, sortiert diese alphabetisch zusammen und die Ausgabe wird von der shell in die Datei d gelenkt.
- **diff <name1> <name2>**
  Vergleicht zwei Dateien und gibt die Unterschiede aus. Leider ist die "normale" Ausgabe als Eingabe für sed gedacht und für Menschen nicht einfach lesbar. Es gibt aber nette Optionen, wie z.B. "-y", womit die zwei Dateien nebeneinander aufgelistet werden.

Netzwerk

Alle Kommandos in diesem Abschnitt benötigen beim entfernten Rechner ein bereits ständig laufendes Programm, dass die Verbindung annimmt. Unter Linux ist dies häufig das Programm "sshd". Nur der Besitzer des Rechners muss dies installieren und starten. Der Anwender muss nichts davon Wissen bzw. Hoffen, dass dieses Programm läuft.

- **ssh <username@host>**
  Einloggen auf einem entfernten Host (Rechner) mit dem angegebenen Usernamen. ssh steht für (s)ecure (sh)ell. Man sitzt jetzt quasi vor einem anderen Rechner, bis man sich wieder ausloggt.
- **scp <von> <nach>**
  Ein bisschen wie cp, aber zum Kopieren von Dateien oder Verzeichnisbäumen zu einem anderen

Rechner über eine verschlüsselte Verbindung. Für entfernte Dateien muss man vor den Dateinamen <username>@<rechnername>: schreiben (mit dem Doppelpunkt!).

mehr Kommandos

Gibt es. Massenhaft. Nach 50 Jahren werden immer noch neue Kommandos "erfunden" und zum Standard, während andere in Vergessenheit geraten. Man lernt hier ein Leben lang.

Ein Blick auf die Syntax von Shells

Die Shell kennt das Wildcard (dt: Joker) "*". Wird es benutzt versucht die Shell dieses Wildcard zu expandieren in auch mehrere Datei- und Verzeichnisnamen. (Wenn es keinen Treffer gibt bleibt der Stern erhalten.) "echo *" gibt alle Dateien des aktuellen Verzeichnisses aus, weil die Shell den Stern in alle Namen des aktuellen Verzeichnisses expandiert hat. echo weiß gar nicht, dass es Dateinamen ausgibt.

Alle Kommandos unter Unix akzeptieren deshalb statt nur einem Dateinamen auch eine Liste von Dateinamen. So kann man mit rm beliebig viele Dateien auf einmal löschen. Vorsicht bei Kommandos wie cp und mv, die genau zwei Dateinamen erwarten: die Shell expandiert Wildcards ohne Rücksicht auf Verluste! "mv abc*" würde, wenn "abc*" zu "abcdef abcxyz" expandiert, offensichtlich zu "mv abcdef abcxyz". Das bewirkt dann, dass abcdef umbenannt word zu abcxyz, während das alte abcxyz Platz machen muss, und gelöscht wird. Wenn man das Verhalten kennt, kann man es aber sogar ausnutzen.

Was tun, wenn der * nicht expandiert werden soll? Dafür kann jede shell sogenanntes 'Quoting': wenn man etwas in einfache Hochkomma setzt, dann weiss die shell, dass sie die Zeichen darin in Ruhe lassen soll. "echo '*'" gibt genau den Stern wieder aus.

Wenn man weiter in die Shell einsteigt: es gibt mehr als nur den * als Wildcard, und es gibt auch noch mehr Mechanismen zum Quoting. Z.B. bewirkt ein \ vor einem Zeichen ebenfalls, dass die Shell es in Ruhe läßt. Es gibt aber auch noch shell-Variablen, Schleifen, einen Kommando-alias-Mechanismus, ...

Ein sehr nützlicher Tipp sei noch der sogenannte recall-Buffer: mit den Pfeil-hoch und Pfeil-runter Tasten kann man Kommandos zurüchholen, um sie erneut auszuführen, gegebenfalls nachdem man sie modifziert hat.

Aufgabe 0.1: Dateien verwalten

Geben Sie folgendes Kommando ein:

```
/import/grpdrvs/ti_prak/gdbs_test   365
```

Evtl. stellt das Kommando Fragen, wenn es Probleme hat. Aber typisch wird es in Ihrem Home-

Verzeichnis unter gdbs_test einen kleinen Baum aus Verzeichnissen und Dateien anlegen. Die Aufgabe ist es, den Baum in Ordnung zu bringen! Das Skript kann man auch nochmals aufrufen um den Anfangszustand wieder herzustellen, allerdings sollte man dazu den Verzeichnisbaum gdbs_test vorher mit cd verlassen.

Den Baum anschauen kann man mit "ls", "ll" oder "ll -R". In Dateien hineinschauen kann man mit "cat <Dateiname>". Mit den daraus gewonnen Informationen sollte man nun passende Kommandos finden, die meist auf das Kommando "mv", selten auf "rm" oder "rm -r" oder "rmdir" hinauslaufen. Keinesfalls sollen Sie den Inhalt von Dateien ändern. vi, nano, ... werden nicht benötigt.

Achtung: die Aufgaben sind insofern ein bisschen gemein, dass manchmal der Zielort für eine Datei nicht frei ist. "mv" überschreibt ohne Rückfrage Zieldateien, und kann dadurch Dateien löschen. Ein weiteres kleines "Problem" könnte sein, dass relative Angaben vorkommen können, wie "bewege diese Datei ein Verzeichnis nach oben", die man natürlich nur einmal ausführen soll.

Aufgabe 0.2: ein paar kleine Aufgaben

Die Aufgaben in diesem Kapitel sollten Sie sich alle anschauen, und dann wenigstens zwei erfolgreich bearbeiten, aber gerne alle probieren.

Die Aufgaben zeigen ein bisschen die Vielseitigkeit der Kommandozeile, aber auch den Grund, warum die Kommandozeile wohl eher unbeliebt ist: neue Kommandos wehren sich ehe man sie nutzen mag. Die man-pages helfen zwar, aber auch diese muss man lesen lernen. Tun Sie dies bei den Aufgaben hier!

Aufgabe: Uhrzeit und Datum in eine Datei schreiben

Schreiben Sie mittels "date" (und evtl. "echo") in eine neu erzeugte Datei eine Zeile Text "Heute ist der dd.mm.yyyy", wobei dd, mm und yyyy durch das aktuelle Datum ersetzt sein sollen. Die Ausgabe eines Kommandos leitet man mittels "> name" in eine Datei um.

Ergänzen Sie die Datei anschließend mit einem weiteren "date" Kommando um die aktuelle Uhrzeit im Format "Es ist hh:mm Uhr", wobei hh und mm die aktuelle Uhrzeit sein sollen. Die Ausgabe eines Kommandos kann man mittels ">> name" an das Ende einer Datei anhängen.

Und noch ein Hinweis: Das Kommando "time" hat mit der Uhrzeit nichts zu tun. (Es dient der Messung, wie lange ein Programm läuft.)

Aufgabe: Baum lokal und über das Netzwerk kopieren

Duplizieren Sie mittels "cp" einen (beliebigen) Verzeichnisbaum in Ihrem Heimat-Verzeichnis!

Kopieren Sie dann einen beliebigen Verzeichnisbaum ihres Home-Verzeichnisses über das Netzwerk mittels des Kommandos "scp" zu einem anderen Rechner. Außer tip01 bis tip16 gehen hier auch wsl50 oder wsl51 des Linux-Pools.

Wichtig: geben Sie dem Zielverzeichnis dabei einen anderen Namen! Da alle genannten Rechner dasselbe Home-Laufwerk importieren wird der Baum zwar über das Netzwerk kopiert, landet aber doch wieder in ihrem eigenen Home-Verzeichnis.

Löschen Sie die zwei Kopien wieder! (Die belegen nur unnötig Platz auf der Festplatte und belegen auch Ihr Platten-Quota (== Limitierung, wieviel Platz Sie belegen dürfen.)

Aufgabe: Alle ".java" Dateien finden & nicht löschen*

Wie kann man mit "find" alle Dateien mit dem Namennsmuster "*.java" in einem Verzeichnisbaum finden? Wenn das klappt, wie kann man das Kommando so erweitern, dass die gefundenen Dateien gelöscht würden? Bitte nicht -delete vom find benutzen, weil man den nicht stoppen kann. Benutzen sie "-exec echo rm -i ...". Dadurch bekommen sie, wegen "echo", eine Liste von rm-Kommandos auf den Bildschirm, die nicht ausgeführt werden. Ohne "echo" würden sie allerdings wirklich ausgeführt. Passen Sie also auf, dass Sie nicht wirklich etwas löschen!

Aufgabe: Leere Verzeichnisse finden

Geben Sie eine Kommandozeile an, die alle leeren Verzeichnisse im Baum Ihres Home-Verzeichnisses auflistet! (Damit das Ergebnis nicht leer ist, legen sie zwei, drei leere Verzeichnissse vorher an, nicht alle im obersten Verzeichnis!) Tipp: "find" ist auch hier das Kommando der Wahl.

Aufgabe: Größten Platzfresser finden

Verketten Sie "du" und "sort" so mit einer Pipe ("|"), dass eine Liste aller ihrer Dateien und Verzeichnis ihres Wurzelverzeichnisses entsteht, in der die Platzbelegung als Zahl vorne steht, dahinter der Datei oder Verzeichnisname, und nach Platznutzung so sortiert, dass die großen unten stehen!

Aufgabe 0.3: Hello World

Heutzutage werden Programme oft mit sogenannten Integrated Development Environments, kurz IDEs entwickelt. Diese bieten meist einen Editor, Compiler, Linker, Laufzeitbibliothek, Laufzeitumgebung und kleine Tools unter einem Dach. Auch die Vorkonfiguration für typische Projekte ist meist mit wenigen Mausklicks abrufbar.

Das ganze ist durchaus praktisch und nützlich. Einer der Nachteile ist aber, dass für Einsteiger ziemlich viel Magie darin steckt. Hier soll deshalb ein kleines Java-Projekt auf der Kommandozeile realisiert werden.

Erzeugen Sie eine Datei, die den Quelltext für Ihr Hello-World-Programm enthält. Dies geht mit einem Editor leicht, und das ist auch der richtige Weg. Zum Verständnis aber: fällt Ihnen ein Weg ein, wie sie diesen Schritt ohne Editor schaffen?

Suchen Sie passende Kommandos, damit Ihr Programm läuft und einen Ausgabe erzeugt!

Aufgabe 0.4: Sieb des Erathostenes

Erweitern Sie danach das Programm, so dass es mittels des Algorithmus "Sieb des Erathostenes" alle Primzahlen bis 99 findet und ausgibt. Widerstehen Sie der Versuchung etwas aus dem Internet zu kopieren und/oder mit einer IDE zu entwickeln, sondern probieren Sie ein Gefühl für die Programmentwicklung auf der Kommandozeile mitzunehmen! Der Zyklus ist Editor - Compiler - Programm-testweise-starten. Zur Fehlersuche benutzt man zusätzliche Ausgaben, also System.out.println(...) um Zwischenwerte zur Kontrolle auszugeben.

Es folgt das "Sieb des Erathostenes" als Prosa-Text, so dass man es in zwei Schritten implementieren und jeweils testen kann:

1. ein Array wird von 2 bis 99 mit "true" vorbesetzt: die Annahme ist, dass alle diese Zahlen Primzahlen sind. Eine Schleife gibt alle Zahlen aus, die im Array noch "true" sind.
2. Jedesmal, wenn eine Zahl als Primzahl ausgegeben wird muss eine zweite Schleife, die in der äußeren geschachtelt ist, alle vielfachen der Primzahl im Array auf "false" setzen.

Warum überhaupt noch Kommandozeile?

Die Kommandozeile ist oft mächtiger als grafische Oberflächen. Nur als Beispiel: wie würden Sie die Aufgaben von 0.2 mit der Maus erledigen? Im Windows-Pool ist zur Suche nach Platzfressern auf dem Home-Laufwerk ein Spezial-Tool "Ridnacs" installiert, damit man das überhaupt kann.

Wichtiger ist aber: Kommandozeilen kann man immer auch zum Automatisieren von Aufgaben nutzen, während graphische Oberflächen dafür eher ungeeignet sind:

- Es ist nicht einfach einen Button zuverlässig automatisiert zu treffen.
- Automatisierte Aufgaben sollen meist "im Hintergrund" ablaufen. Welchen Zweck hat es Fenster und Buttons zu "malen", obwohl kein Mensch damit arbeiten soll?

Die Notwendigkeit ist auch daran sichtbar, dass große Softwarepakete, wie z.B. MS Office und Adobe Creative Suite Skriptfähigkeit mitbringen. Offensichtlich ist die Notwenigkeit zur

Skriptfähigkeit auch den Herstellern von solchen doch eher graphisch orientierten Programmen wichtig genug als Verkaufsargument.

Skripte in Shell-Syntax

Shells sind grundsätzlich Script-fähig. Es gibt Variablen, IF, Schleifen, Unterprogramme, usw. Dennoch kann man nur davon abraten Skripte in einer Shell-Sprache zu schreiben.

Shells sind dafür ausgelegt, damit man schnell interaktiv damit arbeiten kann. Zum Skripten sind sie nur sehr bedingt geeignet. Als Richtschnur: wenn man nur einige Kommandos nacheinander ausführen will, dann ist ein shell-Skript geeignet. Sobald man mehr will sollte man eine fortgeschrittene Skript-Sprache wählen: Python (aktuell relativ beliebt), Perl (mein Favorit), Ruby, LUA, ... um nur einige zu nennen.

Alle diese Sprachen erlauben "normales" Programmieren in Blöcken, mit lokalen Variablen, sauberer Strukturierung und haben Möglichkeiten integriert, die Shells nicht bieten, wie z.B. Listen, Hash-Maps, ... Ganz wichtig dabei auch: alle diese Sprachen binden das Ausführen von Shell-Kommandos gut ein, so dass man die ganzen Möglichkeiten der Shell auch in diesen Skriptsprachen nutzen kann!

"Skript" vs. "Programm"

Die Begriffe Skript und Programm (und noch einige mehr wie App, Anwendung, ...) sind nicht fest definiert. Mit Skript assoziiert man wohl:

- Von einem Interpreter ausgeführt.
- Schnell geschrieben.
- Leicht modifizierbar.
- Ungeeignet für rechenintensive Aufgaben.
- Oft ein "Wegwerfartikel", zur seltenen oder gar nur einmaligen Anwendung.

Bei Programmen denkt man eher an:

- Nach Übersetzung durch einen Compiler als eigenständiges Programm ausgeführt.
- Schnell in der Ausführung.
- Aufwendig zu erstellen.
- Modifikationen möglich, aber teuer.
- (Hoffentlich) Getestet, halbwegs Fehlerfrei und robust gegenüber Fehleingaben.

Um ein Schachprogramm zu schreiben würde wohl niemand ein Skript in Erwägung ziehen. Man braucht möglichst viel Rechenleistung, wenn eine nennenswerte Spielstärke erreicht werden soll, d.h., man kann es sich kaum leisten den Faktor 10 oder mehr zu verschenken, den Programme gegenüber Skripten leicht erzielen.

Umgekehrt wäre es eher dumm ein Programm zu schreiben, um einen Nutzer für einen Computerpool anzulegen. Man braucht ziemlich sicher für den eigenen Pool einige Details, die nicht vorgefertigt existieren, so dass etws programmiert werden muss. Über die Jahre müssen immer wieder kleine Modifikationen eingepflegt werden. Dagegen ist die Laufzeit des Skripts nebensächlich, da wohl nur einige, wenige Nutzer pro Tag angelegt werden.

Faustregel:
Skripte sind schnell erstellt, Programme sind schnell in der Ausführung.

Grundsätzliche Arbeitsweise von Kommandozeilen

Um die Arbeitsweise von Kommandozeilen zu erklären, soll beispielhaft das folgende copy Kommando betrachtet werden:

```
cp -a file1 verzeichnis2 ziel3
```

Das Kommando wird von der Shell zuerst in Wörter zerlegt:

```
cp
-a
file1
verzeichnis2
ziel3
```

Das erste Wort ist für die Shell der Dateiname eines zu startenden Programms. Beim Start werden diesem Programm alle Wörter der Zeile als Parameter übergeben. In Java erhält das Programm diese Wörter im args[]-Array von "public static void main(String[] args)". Eine Java-Eigenheit ist, dass das erste Wort fehlt, also der Name des gerade gestarteten Programms.

Das Programm cp erhält die Wortliste als Parameter. Die restlichen Wörter werden vom Programm cp mit Bedeutung versehen, allerdings folgt es einigen Konventionen, damit der Nutzer nicht unnötig verwirrt wird.

Per Unix-Konvention verstehen Programme Worte mit führendem Minuszeichen als sogenannte Optionen, während die ohne Minuszeichen Parameter genannt werden. Die Bezeichnung Option lässt ahnen, dass diese optional sind, also nur angegeben werden, wenn man etwas erreichen will, was nicht Standard ist. Die Parameter sind hingegen Pflicht, damit das Progamm überhaupt weiß, was es tun soll. Bei cp sind es Datei- oder Verzeichnisnamen.

Im Beispiel ist die Option "-a" angegeben, die bei cp (im wesentlichen) besagt, dass rekursiv gearbeitet werden soll, was hier bedeutet, dass ganze Verzeichnisbäume kopiert werden sollen.

Die Parameter file1, verzeichnis2 und ziel3 werden von cp als Datei- bzw. Verzeichnisnamen verstanden und es gilt bei cp die Regel, dass nur der letzte dieser Namen das Ziel ist. Alle Parameter davor sind Quellen. cp wird also die Datei file1 oder gegebenenfalls das Verzeichnis file1 mit allen Unterverzeichnissen in das Verzeichnis ziel3 kopieren. Danach wird mit verzeichnis2 genauso verfahren. (Übrigens muss ziel3 vorher existieren.)

Im Prinzip sind drei Schritte passiert:

1. Die Shell zerlegt die Zeile in Worte, interpretiert das erste Wort als Dateiname mit einem Programm darin und startet dieses.
2. Das aufgerufene Programm konfiguriert sich entsprechend der Optionen.
3. Das aufgerufene Programm erledigt die Arbeit, entsprechend den Parametern.

Die Klasse KernelWrapper

Um eine Shell zu schreiben fehlen Java einige Kernel-Aufrufe. Deshalb ist für das Praktikum die Klasse Kernelwrapper vorgegeben, die alles nötige enthält.

Für alle Methoden der Klasse KernelWrapper kann man unter jedem (halbwegs vollständig installierten) Linux die offizielle Kernel Dokumentation abrufen mittels des Kommandos "man". "man fork" gibt also die offzielle Erklärung dieser Methode. Die folgende Liste ist deshalb kurz gehalten und man sollte die man-Pages zusätzlich lesen.

Unter Unix ist es üblich, dass Systemfunktionen einen Int-Rückgabewert mit einem Fehlercode liefern. Welcher Wert was bedeutet (Erfolg, Fehlertyp1, Fehlertyp2, ...) schaut man für jede Funktion in deren man-Page im Abschnitt "RETURN VALUE" nach. Wenn eine Funktion etwas Sinnvolles zurückgibt und somit eigentlich kein Platz für einen Fehlercode ist, dann ist üblicherweise ein spezifischer Wert angegeben, der Misserfolg meldet und die globale Int-Variable "errno" enthält den eigentlichen Fehlercode. Als Helfermethode gibt es die Methode perror(), die den Wert der

Variablen errno als lesbaren String ausgibt. Im Praktikum enthalten die Methoden der Klasse KernelWrapper gleich den passenden Aufruf von perror(), um Fehlermeldungen auszugeben, kehren aber anschließend trotzdem mit dem Fehlercode zurück.

- **int fork()**
  Kehrt doppelt, also in zwei Prozessen zurück. Das bedeutet, das das aufrufende Programm jetzt zwei mal läuft! Der Eltern-Prozess erhält als Rückgabewert die pid (Prozess-Identifikations-Nummer) des Kind-Prozesses, der Kind-Prozess eine 0. Andere Unterschiede zwischen den Prozessen gibt es kaum.
  Insbesondere sollte man in diesem Versuch auch daran denken, dass offene Dateien und Pipes nach fork() von beiden Prozessen geöffnet sind.
- **int execv(String path, String[] argv)**
  Kehrt nicht zum Aufrufer zurück bzw. nur im Fehlerfall. Das laufende Programm wird ersetzt durch das Programm, das in der Datei mit dem Namen path angegeben ist. Es erhält die Liste argv als Parameter. Per Konvention ist dies die Liste der Worte der aufrufenden Kommandozeile, wobei das erste Wort unter Index 0 der Name des gerade gestarteten Programms ist, also eine Kopie des Parameters path. Oft ist die Kopie allerdings nicht exakt derselbe String. In path steht also z.B. "/bin/cp", der vollständige Dateiname des zu startended Programmes und in argv[0] nur "cp", exakt das, was der Benutzer getippt hat. Sonstige Details: siehe man-Page.
- **int waitpid(int pid, int[] status, int options)**
  waitpid(child_pid, status, 0) wartet, bis der Prozess mit der pid child_pid sich beendet hat. status ist ein Array mit einem Element, in dem der Returncode des Kind-Prozesses zurückgegeben wird. (Eigentlich ist status ein out-parameter, aber Java hat so etwas nicht. Das Array mit einem Element wird hier als out-Paramater genutzt.)
- **exit(int returncode)**
  Beendet den laufenden Prozess und gibt returncode an den Eltern-Prozess zurück. Der Wert wird bei waitpid() im status-Parameter abgerufen.

Die Klasse KernelWrapper enthält für die späteren Aufgaben noch mehr Funktionen. Zuerst einmal alles, um Dateien zu bearbeiten:

- **int open(String path, int flags)**
  Öffnet die Datei, die in "path" angegeben ist, zum Lesen und/oder Schreiben. Zur Auswahl dafür übergibt man in "flags" eine der Konstanten O_RDONLY, O_WRONLY oder O_RDWR, evtl. mit weiteren Flags zusammenge-oder-t (siehe man-Page). Der Return-Wert ist bei Erfolg der File-Deskriptor, später als Parameter "fd" benötigt, den man einigen der folgenden Funktionen übergibt, damit klar ist, von welcher Datei z.B. gelesen werden soll. (Für Objekt-Orientiertes Denken: fd ist die Nummer der Instanz, die für eine geöffnete Datei steht.) Der Returnwert -1

besagt, dass ein Fehler aufgetreten ist. Hinweis: die man-Pages liefern auf "man open" die falsche Seite, "man 2 open" ist hier richtig.

- **int read(int fd, byte[] buf, int count)**
  Lese von der geöffneten Datei fd (bis zu) count bytes in den Puffer buf. Der Returnwert besagt mit dem speziellen Wert -1, dass ein Fehler aufgetreten ist, oder normalerweise, wieviele Bytes wirklich gelesen wurden. Insbesondere am Dateiende wird der Rückgabewert einmal kleiner sein als count.
  read() liefert wirklich nur bytes und ist nicht zeilenorientiert. Das Zeilenende-Zeichen, unter Unix <lf> bzw. \n, kann mitten in buf[] stehen, auch mehrfach, und nur per Zufall auch mal am Ende von buf[] auftauchen.
- **int readOffset(int fd, byte[] buf, int offset, int count)**
  Java fehlt gegenüber C die freie Verwendung von Zeigern. readOffset() wird im Praktikum angeboten, um diesen Unterschied zu überbrücken: Die gelesenen Bytes werden im Puffer buf ab dem Index offset abgelegt. Natürlich dürfen dann maximal nur noch so viele Bytes eingelesen werden, wie der Restbereich des Puffers noch an Platz hat. Ansonsten verhält sich diese Methode genau wie read(). readOffset() hat demnach auch keine eigene man-Page, sondern es gilt die von read().
- **int write(int fd, byte[] buf, int count)**
  So ähnlich wie read(), nur halt write(). Wie bei open liefert "man write" die falsche Seite, "man 2 write" ist die richtige Hilfe.
- **int writeOffset(int fd, byte[] buf, int offset, int count)**
  Analog entsprechend der Mischung aus read(), write() und readOffset().
- **int lseek(int fd, int offset, int whence)**
  Den Lese-/Schreibzeiger im Filedeskriptor fd umpositionieren. (Das ist die Positionsangabe, ab der weiter gelesen bzw. geschrieben werden soll.) whence ist eine der Konstanten SEEK_SET, SEEK_CUR oder SEEK_END, welche besagt, ob das offset vom Anfang, der aktuellen Position oder vom Dateiende her gerechnet werden soll. offset darf auch negativ sein, solange das sinnvoll ist.
  Beispiel: lseek(fd, 0, SEET_SET) setzt den Lesenzeiger wieder auf den Anfang der Datei.
- **int close(int fd)**
  Das Gegenstück zum open(), wenn man mit der Verarbeitung einer Datei fertig ist. Wird beim Beenden des Programms automatisch für alle noch offenen Dateien ausgeführt. Auch nützlich nach fork(), z.B. wenn eine offene Datei nur noch vom Kind-Prozess geöffnet benötigt wird. Dann sollte der Eltern-Prozess auf dieser Datei close() aufrufen.

Und noch mehr Methoden:

- int pipe(int[] fd)

- siehe "man 2 pipe". Man sollte sich erinnern, dass es pipe() gibt, sobald es in den Aufgaben unten erwähnt wird.
- int dup2(int oldfd, int newfd)
- siehe "man dup2". Braucht man zusätzlich, sobald man pipe() braucht ...
- …
- In der Klasse sind noch einige Methoden mehr, die aber für das Praktikum nicht relevant sind.

Die Arbeitsumgebung

Um die Klasse KernelWrapper zu benutzen, kopiert man sich auf einem Rechner im Praktikum das Verzeichnis /opt/shell_versuch in ein neues, privates Verzeichnis, z.B. wie folgt:

```
cp -a /opt/shell_versuch .
```

Beginnt man zu arbeiten oder will man weiterarbeiten, dann wechselt man in das Verzeichnis und muss einmal das Skript source_mich aufrufen:

```
cd shell_versuch
source source_mich
```

Übrigens: "source <name>" führt die Zeilen von der Datei name aus, als ob man sie direkt eintippen würde.

Danach kann man mit einem beliebigen Editor wie z.B. "nano" Java-Quelltext bearbeiten:

```
nano hello.java
```

Dabei sollte der eigenen Quelltext auch die Klasse Kernelwrapper importieren, genau wie es im Quelltext hello.java schon enthalten ist:

```
import static cTools.KernelWrapper.*;
```

Übersetzen und Ausführen geht dann mit:

```
javac hello.java
java hello
```

Dateien und Filehandles

Bevor es an die Aufgaben des Praktikums geht noch ein Exkurs zum Thema Dateien und Filehandles. Insbesondere soll auch kurz das Zusammenspiel mit fork() betrachtet werden.

Dateien

Dateien dienen der persistenten Speicherung von Daten, typisch auf Festplatten/SSDs, USB-Sticks, SDRAM-Karten, Magnetbändern, usw. Heutzutage, unter Linux und Windows, sind Dateien meist als eine Sequenz von Bytes organisiert. Die Anzahl der Bytes ist die Länge der Datei und nur limitiert durch die Größe des Speichermediums.

Dateien sind Typfrei. Allerdings wird oft die Endung des Dateinamens ("extension") als Hinweis genutzt, was eine Datei enthält. Bekannt ist sicherlich .txt für Dateien, die Texte in UTF-8, früher ASCII enthalten. Andere Endungen enthalten ebenfalls UTF-8, meist in speziellerer Form: .java, .html, .json, .xml um nur einige zu nennen. Andere Endungen deuten auf ganz andere Kodierungen hin. .jpeg, .gif, .png enthalten z.B. wahrscheinlich Bilddaten, deren Kodierung man im Internet recherchieren kann. .exe deutet unter Windows auf ein ausführbares Programm hin. usw.

Editoren sind Programme, die eine Datei anzeigen, ändern und modifiziert wieder schreiben können. Für Textdateien gibt es Notepad, emacs, vi, nano und viele mehr. Für andere Dateiformate nutzt man andere Editoren, die den Inhalt sinnvoll bearbeiten können, wie z.B. Photoshop als Bildeditor. Aber auch vi kann durchaus Bilddateien bearbeiten, wenn man unbedingt will. Dies funktioniert, weil Dateien typfrei sind, wie schon gesagt.

Dateien aus der Sicht eines Programmierers

Jedes Betriebssystem bietet Programmierern Dateien als Hilfsmittel an, um Daten dauerhaft zu speichern. Die Schnittstelle dazu ist bei allen Betriebssystemen mehr oder weniger identisch und soll hier am Beispiel von Linux vorgestellt werden.

Lesen einer Datei

Ein minimales Programm (ohne Fehlerbehandlung), das den Inhalt einer Datei auf den Bildschirm bringt:

```
int fh = open("subdirectory/dateiname.txt", O_RDONLY);
uint8_t one_byte;
while (read(fh, &one_byte, 1)==1) {
printf("%c", one_byte);
}
close(fh);
```

open() öffnet die angegebene Datei (in "subdirectory" mit dem Namen "dateiname.txt") nur zum Lesen (O_RDONLY). Man erhält als Ergebnis ein sogenanntes "filehandle", im Programm die Variable fh.
read() liest über das filehandle ein Byte in eine Variable (Parameter "1" und "one_byte"). Solange read ein byte liefert ("==1") wird dieses auf den Bildschirm kopiert (mit "printf").

close() sagt dem Betriebssystem, dass mit der Datei nicht weiter gearbeitet wird. Das filehandle ist ab sofort ungültig.

Heutzutage würde man das ganze Objekt-Orientiert wie folgt beschreiben:

open() ist Konstruktor für die Klasse Filehandle. Die Instanz ist verbunden mit einer Datei auf dem persistenten Plattenspeicher/Filesystem. Wie dies geschieht ist versteckt in der Klasse.
one_byte = filehandle->read(1) ist der Aufruf um ein Byte zu lesen, liefert aber irgendwie auch zurück, wenn kein Byte mehr da ist. Z.B. als Exception.
filehandle->close() ist der Destruktor.

Ungewohntes Detail in obigem Programm: die Instanz der Klasse filehandle ist vom Typ int?!? Dies ist aber leicht zu erklären: der Objektorientierung ist es herzlich egal, wie man eine Instanz wiederfindet. In der vorliegenden, etwas antiquierten Fassung verwaltet die Klasse intern ein Array aller Instanzen, und indiziert eben dieses Array mit dem int.

Filehandles

Ein Filedeskriptor steht für eine "geöffnete" Datei. Dieser Schritt wird gemacht, weil der Vorgang des Öffnens ziemlich Aufwendig ist, und dieser Aufwand vermieden werden soll bei den folgenden, häufigen Lese- und Schreiboperationen.

Was genau im Filedeskriptor steht variiert je nach Betriebssystem. Im Prinzip findet man aber immer folgendes:

- Das Grundlegendste: Die Verbindung zum Filesystem, also wo die Daten der Datei liegen.
- Welche Rechte hat diese Öffnung? Lesen? Schreiben? Nur Anhängen?
- Ist der Zugriff exklusiv? Manche Betriebssysteme verwalten hier auch Sperren (Locks, siehe Synchronisations-Versuch dieses Praktikums).

Im Filedeskriptor steht aber auch das für den Programmierer wichtigste:

- Wo wird weitergelesen/weitergeschrieben?

Das Konzept von geöffneten Datei beinhaltet, dass eine Schreib-/Leseposition im Filedeskriptor verwaltet wird. Diese Position wird bei jedem Schreiben/Lesen automatisch weitergeschaltet. Ausserdem gibt es den Aufruf "seek()" (im Linux Kernel lseek(): "man lseek"), mit dem das Programm diese Schreib-/Leseposition frei manipulieren kann, z.B. um eine Datei ein zweites Mal auszulesen ohne sie erneut zu öffnen.

Filehandles und fork()

Der Systemaufruf fork() bewirkt, dass auch alle filehandles dupliziert werden. Eine geöffnete Datei ist nach fork() doppelt geöffnet.

Was tun, wenn man eine Datei nur im Vater- oder nur im Kind-Prozess geöffnet haben will? Die einfachste Lösung ist: erst nach dem fork() öffnen! Dies funktioniert beim Umlenken mit < und > sehr gut (in Aufgabe 3a). Wenn das nicht geht, dann alle 'falsch geöffneten' Filehandles sofort nach dem fork() schliessen, damit diese offenen Filehandles nicht irgend etwas blockieren. (Das ist einer der kniffligen Punkte bei der Aufgabe 3b, beim umlenken mittels "|".)

Aufgabe 1: Erstellen einer minimalen Shell

Eine Unix-Shell muss minimal die folgenden Aufgaben (in dieser Reihenfolge) erledigen:

1. Zeile einlesen

   Zuerst sollte ein Prompt ausgeben werden. Das sind einfach einige Zeichen Text, an denen der Nutzer erkennen kann, dass die Shell jetzt wieder auf eine Eingabe wartet. Dann muss eine Kommandozeile, ein String eingelesen werden. Eine gute Shell sollte hier Hilfen anbieten, wie:

   - Editiermöglichkeiten, damit man z.B. mit den Pfeiltasten nach rechts und links in der Zeile hin- und herfahren kann, um Tippfehler zu verbessern
   - einen "recall Buffer", d.h., die Pfeiltasten nach oben und unten sollten bereits benutzte Kommandos wieder hervorholen
   - "file name completion", d.h., üblicherweise die <tab>-Taste versucht, angefangene Worte zu vollständigen Dateinamen zu ergänzen, indem sie im Verzeichnisbaum schaut, ob es Dateien mit dem getippten Wortanfang gibt.

   Im Praktikum müssen Sie sich darum nicht kümmern, da es uns nur um das Prinzip der Kommandozeile geht.

2. Zerlegen der Zeile in Worte

   Die Shell muss die Zeile in Worte zerlegen, d.h., man hat als Ergebnis eine Liste von Worten.

3. Das Programm starten mit Parametern

   Das erste Wort der Kommandozeile wird als Dateiname verstanden. Die Datei muss existieren und ein ausführbares Programm enthalten. Falls hier Probleme auftreten, sollte eine geeignete Fehlermeldung ausgegeben werden und danach zurück nach 1), also eine neue Kommandozeile anfordern.

Wenn kein Problem vorlag, soll das Programm gestartet werden. Schlimmstenfalls könnte das aufzurufende Programm aber abstürzen, was wiederum die Shell nicht stören darf! Als Schutzmechanismus ist in Unix ein Prozess das geeignete Mittel. Eine Shell unter Unix startet das aufzurufende Programm deshalb in einem eigenen Prozess und muss dann nur noch warten, bis dieser zweite Prozess sich beendet hat (egal, ob reguläres Ende oder Absturz).

Unter Unix richtet man einen neuen Prozess mittels des Systemaufrufs fork() ein. Dieser verdoppelt den aufrufenden Prozess: das fork() aufrufende Programm "Shell" läuft jetzt zwei mal in parallelen Prozessen! Der einzige Unterschied dieser zwei Prozesse ist der Rückgabewert von fork(): einer der beiden Prozesse wird Elternprozess genannt und erhält die Prozess-Nummer (Process-ID, PID) des Kind-Prozesses (ein Integer größer als Null), der Kind-Prozess erhält den Rückgabewert Null. (Ein negativer Rückgabewert von fork() wäre eine Fehlermeldung.) Außer dem Rückgabewert von fork() sind die Prozesse identisch. Typisch folgt nach dem Aufruf von fork() deshalb sofort ein if, das die beiden Prozesse unterschiedliche Dinge tun läßt.

Der Kind-Prozess tauscht nun mittels einem Aufruf an execv() seinen Programmcode (den der Shell) gegen den Code aus einer Datei aus. Dies sollte natürlich die Datei des zu startenden Programmes sein. Der Dateiname ist ein Parameter beim Aufruf von execv(), das außerdem die Wortliste der Kommandozeile erhält (incl. dem Programmnamen selbst). Der Aufruf an execv() stoppt die Shell, entfernt sie aus dem Speicher, lädt als Ersatz die Datei, und startet das neue, nun im Speicher befindliche Programm. Dieses kann nun seine Arbeit verrichten. Wenn es fertig ist, teilt es dem Betriebsystem mittels exit() mit, dass es fertig ist. Das Betriebssystem beendet daraufhin das Programm und den Prozess.

Der Elternprozess verbleibt nach dem Aufruf an fork() im Programm Shell und wartet mittels waitpid() darauf, dass der Kindprozess endet.

4. Damit man das nächste Kommando eingeben kann: zurück zu Schritt 1)

Beim Aufruf von exit() wird als Parameter ein Integer, der sogenannte Returncode an das Betriebssystem übergeben. Für den beendeten Prozess merkt sich das Betriebssystem diesen Wert, um ihn dem Elternprozess bei waitpid() mitzuteilen. Dies ist wichtig, wenn man Skripte schreibt, in denen ein Programm dem nächsten Schritt mitteilen will, ob alles in Ordnung war. Der nächste Schritt könnte ja ein if sein, das den Returncode abfragt. Per Konvention bedeutet der Returncode Null "alles ok", alles andere sind programmspezifische Fehlercodes.

In einer Shell ist man in einer Endlosschleife gefangen, die Kommandos einliest und ausführt. Zum Verlassen dieser Schleife besitzt praktisch jede Shell ein spezielles Kommando "exit", auf das hin sich die Shell selbst beendet.

Der Mechanismus um fork(), execv() und waitpid() wirkt merkwürdig, ist aber auf seine Weise genial. Z.B. übernimmt der Kindprozess über den execv() hinweg die offenen Dateien. Ist die Verbindung zum Terminal noch von der Shell her offen, dann kann das Programm im Kindprozess direkt mit "system.out.println()" loslegen. Aber noch besser: die Shell kann dem Programm auch andere offene Dateien vorbereiten, und das aufgerufene Programm erfährt gar nicht, dass es gerade nicht auf den Bildschirm, sondern in eine Datei schreibt.

Die Aufgabe 1 besteht darin, genau eine solche eben beschriebene minimale Shell selbst zu programmieren. Dazu braucht man natürlich Zugriff auf die genannten Systemfunktionen. Diese und noch einige mehr sind im Praktikum auf Java-Ebene in der Klasse KernelWrapper bereitgestellt und im nächsten Kapitel beschrieben.

Falls Sie noch wenig Programmiererfahrung haben: Die grundlegendste Art Fehler zu suchen ist "System.out.println" zu benutzen, um nach jedem (!)kleinsten(!) Teilschritt zu schauen, ob die Ausgabe das erwartete ist. Ein kleines Fehlerchen in einem "trivialen" Schritt kann drei Zeile später fürchterliche Dinge bewirken!

Aufgabe 2: Wildcards * und ?

Erweitern Sie Ihre Shell aus Aufgabe 1 um die Möglichkeit, Gruppen von Dateinamen mittels des Wildcards * anzugeben. Gibt man z.B. "abc*xyz" auf der Kommandozeile an, dann wird dieses Wort expandiert in alle Dateinamen, die auf dieses Muster passen, also "abcxyz", "abc123xyz", "abc...xyz", "abc*xyz", "abcxyzxyz", aber nur auf Dateien, die im aktuellen Verzeichnis existieren. Der Stern ersetzt also einen beliebigen, auch leeren Substring. Das eine Wort mit dem Wildcard kann dabei zu einem oder mehreren Worten für das aufgerufene Programm werden. Als Sonderfall gilt: wenn keine einzige Datei auf das Muster paßt, dann bleibt das Wort incl. dem * unverändert stehen, d.h., es wird in seiner ursprünglichen Form an das aufgerufene Programm übergeben.

Ganz entsprechend dem * soll auch das ? unterstützt werden, das nicht einen beliebigen Substring, sondern genau ein Zeichen ersetzt (also keine zwei, drei, und insbesondere auch nicht null Zeichen).

Natürlich müssen die Kommandozeilen-Programme alle darauf vorbereitet sein, dass mehrere Dateinamen als Parameter übergeben werden. Dies ist unter *ix Standard. Unter Windows, bei der Shell cmd.exe ist dies nicht so!

Es wird dringend empfohlen die Wildcards nicht selbst zu implementieren. Suchen Sie eine geeignete Java-Bibliothek!

Zum Testen empfiehlt sich das Programm "/bin/echo", das genau seine Parameter wieder ausgibt: man kann direkt die Worte sehen, die die Expandierung von "*" und "?" bewirkt hat.

Aufgabe 3: stdin und stdout umlenken mit < und >

Kommandozeilen-Programme unter Linux folgen meist einer weiteren Konvention, dass man beliebig viele Dateinamen als Parameter angeben kann, die nacheinander verarbeitet werden. Der spezielle Name "-" bedeutet, dass stdin verarbeitet werden soll. Findet das Programm keinen einzigen Dateinamen in den Wörtern der Kommandozeile, dann wird automatisch stdin gelesen, auch ohne dass "-" angegeben werden muss.

Diese Konvention ermöglicht es unter anderem, Kommandozeilen-Programme als Filter zu nutzen, wie das folgende Beispiel zeigt:

```
du -sm * | grep '^[0-9]\{4,\}' | sort -n > xxx
```

Das erste Kommando "du"==disk-usage liefert für alle Argumente, hier "*"==alle Namen im aktuellen Verzeichnis, den belegten Plattenplatz in Megabyte (-m) und ohne Unterverzeichnisse einzeln aufzulisten (-s). Das Ergebnis sind Zeilen mit einer Zahl am Zeilenanfang und dahinter einem Datei- oder Verzeichnisnamen.

Die erste Pipe "|" verbindet stdout von "du" mit stdin des nächsten Programms, nämlich "grep". Dieses erwartet als ersten Parameter einen Suchstring und weitere Parameter mit zu durchsuchenden Dateien. Da letztere fehlen, durchsucht grep stdin. Der Suchstring ist bei grep eine regular-Expression, so dass hier nach einer wenigstens vierstelligen Zahl am Zeilenanfang gesucht wird. Das Ergebnis ist, dass alle Zeilen vom du-Kommando fehlen, die weniger als 4 Ziffern am Zeilenanfang haben.

Das Ergebnis der Suche wird mittels einer weiteren Pipe an sort gegeben, welches wegen -n nicht alphabetisch, sondern nach Zahlenwerten sortiert. Zuletzt wird das Ergebnis von sort, also dessen stdout mittels ">" (vom Bildschirm weg) umgelenkt in die Datei xxx.

Insgesamt liefert das Beispiel also eine Datei in der eine nach Plattenplatz sortierte Liste steht: welche Dateien und/oder Verzeichnisse belegen mehr als 1 GByte Platz? So etwas tippt ein Anfänger nicht ein, aber ein Unix-erfahrener Anwender tut dies, ohne besonders darüber nachzudenken.

Noch ein wichtiges Detail: der "*" für 'alle Dateien' wird von der Shell durch alle Dateinamen ersetzt. Das Kommando "du" bekommt also den Stern gar nicht zu sehen, sondern einfach ganz viele Dateinamen als einzelne Wörter.

Aufgabe 3a: < und >

Erweitern Sie Ihre Shell aus Aufgabe 2 so, dass stdin und stdout umgelenkt werden können. D.h., die Syntax "< Datei_xxx" lenkt stdin um, und mit ">Datei" wird stdout umgelenkt. (Bei "richtigen"

Shells ist es egal, ob Leerzeichen zwischen ">" bzw. "<" und dem Dateinamen sind. Im Praktikum reicht es, wenn sie eine Syntax unterstützen.)

Das Umlenken wird erreicht, indem die Shell vor dem Start des Programms, das vorne auf der Kommandozeile steht, den Filedeskriptor 0 bzw. 1 umlenkt. 0 und 1 sind per Konvention die Filedeskriptoren für stdin bzw. stdout. "Umlenken" ist der Fachjargon für: Schließen der bisherigen Datei, neu öffnen einer anderen unter derselben Filedeskriptor-Nummer. Das Schließen der bisherigen Datei ist oft das Beenden der Verbindung mit dem Terminal (Tastatur, Bildschirm; natürlich jeweils nur Lesend/Schreibend je nach stdin/stdout). Wichtig: Schließt man die Verbindung zum Terminal an der falschen Stelle, dann kann die Shell selbst die Verbindung zum Terminal verlieren. Nicht gut. Wenn Ihre Shell sich also auf einmal nicht mehr am Bildschirm meldet, dann könnte Ihnen dieser Fehler unterlaufen sein. Auch wichtig: Unter "man 2 open" findet man einen Satz, der hier sehr wichtig ist, damit man überhaupt "umlenken" kann: "The file descriptor returned by a successful call [to open()] will be the lowest-numbered file descriptor not currently open for the process."

Wenn alles fertig ist, dann muss folgendes alles gehen (die Datei abc muss existieren, cat kopiert alle seine Eingabedateien als eine Ausgabedatei nach stdout):

```
cat abc          # Inhalt der Datei abc wird auf dem Terminal ausgegeben
cat < abc        # Inhalt der Datei abc wird auf dem Terminal ausgegeben
cat - < abc      # Inhalt der Datei abc wird auf dem Terminal ausgegeben
cat abc - abc < abc   # abc wird drei mal ausgegeben
cat abc > xyz    # wirkt wie "cp abc xyz"
cat < abc > xxx    # wirkt wie "cp abc xxx"
cat > yyy < abc    # wirkt wie "cp abc yyy"
cat - > zzz < abc  # wirkt wie "cp abc zzz"
```

Das Kommando "cat -" bewirkt, dass cat vom Terminal liest, weil dies normalerweise als stdin geöffnet ist. Damit man cat mitteilen kann, dass End-of-File erreicht ist kann man ^D (Strg-D) am Zeilenanfang eingeben. Der Terminal-Treiber gibt daraufhin ein End-of-File weiter, mit dem Nebeneffekt, dass, wenn man weiterliest, doch wieder Zeichen vom Terminal kommen.

Aufgabe 3b: (entfällt)

Aufgabe 4: head

Schreiben Sie eine Version des Programms "head", das dazu dient Anfänge von Dateien auszugeben.

Das Programm erhält auf der Kommandozeile eine beliebige Menge von Dateinamen, die nacheinander auf stdout (System.out.print benutzen) ausgegeben werden müssen. Einmalig darf dazwischen der spezielle Dateiname "-" auftauchen, der besagt, dass stdin an der Reihe ist. Ist kein

einziger Dateiname auf der Kommandozeile angegeben, also nicht einmal das "-", dann muss (nur) stdin gelesen werden.

An Optionen müssen -c, -n und --help unterstützt werden. Es reicht, wenn der Wert der Optionen c und n nur für positive Zahlen funktioniert. Was die Optionen tun entnimmt man der man-Page ("man head").

Dateinamen und Optionen werden in Java dem Parameter args entnommen, von "public static void main(String[] args)".

Beim Lesen der Eingabedateien ist die Verwendung der Methoden open(), read(), readOffset() und close() aus KernelWrapper Pflicht. Java-eigene Methoden um Dateien zu lesen werden nicht akzeptiert!

Ihr Programm muss in jedem Fall mit exit() enden. Im Erfolgsfall muss dies ein exit(0), in jedem Fehlerfall ein exit(1) sein.

In jedem Fall gilt das original-head von Linux als Referenz. Der Tutor darf, muss aber keinesfalls Abweichungen tolerieren.

Grundlagen der Betriebssysteme - Labor

Versuchsanleitung: Prozeßsynchronisation mit Semaphoren

Im Folgenden werden zuerst Zähler-Semaphore rekapituliert, die aus der Vorlesung bekannt sein sollten, bei Bedarf aber auch z.B. bei
Wikipedia https://de.wikipedia.org/wiki/Semaphor_(Informatik) nachlesbar sind. Danach werden Grundmuster zur Verwendung derartiger Semaphoren gezeigt, die in der Praxis häufig vorkommen. Vor den eigentlichen Aufgaben wird dann noch das Framework beschrieben, das die Versuchsabläufe vereinfacht.

Grundlagen

In diesem Versuch steht der Begriff **Prozess** allgemein für einen Vorgang, bei dem Befehle eines Programms nacheinander abgearbeitet werden, und beinhaltet auch sogenannte "Threads". Ein Prozess entspricht der normalen Abarbeitung von Programmen, die man als Programmierer kennt.

Ein Programm kann sich selbst weitere Prozesse erzeugen, z.B. mittels fork(), siehe Shell-Versuch, so dass das Programm an zwei oder noch mehr Stellen gleichzeitig abgearbeitet wird. Mehrere Befehle eines Programms können dann gleichzeitig abgearbeitet werden!

Arbeiten mehrere Prozesse am gleichen Programm, dann existieren typisch Bereiche, in denen nur ein Prozess pro Zeiteinheit gleichzeitig arbeiten darf, wenn das Programm korrekt funktionieren soll. Derartige Abschnitte heissen **kritische Abschnitte**. Das wohl einfachste Beispiel für einen kritischen Abschnitt ist das sogenannte **lost update Problem**, das in Aufgabe 1 behandelt wird. Kompliziertere Synchronisationsprobleme existieren, haben aber keine einheitlichen, weit verbreiteten Namen.

Semaphore sind der Synchronisationsmechanismus, der in diesem Versuch benutzt wird. Genauer: es werden sogenannte Zähler-Semaphore benutzt, die flexibler sind als sogenannte binäre Semaphore.

In der Modellvorstellung besitzen Zähler-Semaphore intern eine Zählvariable (ein Integer) und eine Menge, in der wartende Prozesse eingetragen werden:

```
struct semaphore {
  int count;
  collection waiting_processes;
}
```

Semaphore wurden vom Niederländer Dijkstra eingeführt, der die zwei wichtigsten Operationen "passeren" und "vrijgeven" mit P und V abgekürzt hat. Auf gut Deutsch: p steht für passieren (im Sinne von: eine Bahnschranke passieren) und v steht für freigeben. Die Operationen sind wie folgt definiert:

```
void p(semaphor) {
  semaphor.count -= 1;
  if (semaphor.count < 0) {
    1) add calling process to semaphor.waiting_processes;
    2) have the calling process sleep, until some v operation reactivates it;
  }
}
void v(semaphor) {
  semaphor.count += 1;
  if (at least one process in semaphor.waiting_processes) {
    remove one process from semaphor.waiting_processes and reactivate it;
  }
}
```

Die Operationen P und V sind etwas Besonderes, weil sie garantieren, dass sie unteilbar ablaufen. Es ist Sache der Implementierung dies sicherzustellen. Semaphore werden deshalb üblicherweise vom Betriebssystem zur Verfügung gestellt, weil nur dieses eine derartige Garantie geben kann. Linux bietet Semaphore in etwas anderer Form an, und die weiter unten beschriebene Bibliothek konstruiert die gewünschten Zähler-Semaphore auf der Basis der vom Linuxkernel angebotenen Semaphore.

Detail am Rande: die Menge der wartenden Prozesse im Semaphor ist nach Definition wirklich nur eine Menge ohne Ordnung, d.h., man weiss nicht welcher Prozess bei der V-Operation geweckt wird. In der Praxis wird aber eigentlich immer eine Queue benutzt, d.h., der Prozess, der am längsten gewartet hat wird geweckt. Dies erleichtert auch das Verhindern des Verhungerns von Prozessen. Offiziell darf man sich darauf aber nicht verlassen.

Aufgabe 0: Begriffe selbst recherchieren

Suchen Sie selbst Erklärungen für die Begriffe "**busy wait**" (dt. "**Aktives Warten**"), "**deadlock**" (dt. "**Verklemmung**") und "**verhungern**", z.B. in Wikipedia. (Vor dem Versuchstermin zu erledigen!)

Muster

Gegenseitiger Ausschluss

Der in der Praxis wohl bei weitem häufigste auftretende Synchronisationsfall ist ein sogenannter kritischer Abschnitt, der nur korrekt funktioniert, wenn maximal ein Prozess gleichzeitig darin Aktiv ist. Ein solcher kritischer Abschnitt erfordert gegenseitigen Ausschluß aller Prozesse.

Einen kritischen Abschnitt erkennt man daran, dass mehrere Prozesse potentiell gleichzeitig den Wert von einer einzigen Variablen schreiben wollen. Ebenfalls kritisch ist, wenn mehrere Variablen gelesen werden sollen, die voneinander abhängig sind. Schreibt nämlich ein Prozess diese Variablen, dann kann ein lesender Prozess evtl. eine Variable im neuen, eine andere noch im alten Zustand sehen, was insgesamt eine fehlerhafte Zustandserkennung ergibt.

Die Kunst beim Synchronisieren besteht darin, derartige Variablen bzw. kritische Abschnitte zu minimieren. Viele Programme bzw. Algorithmen sind nicht oder nur schlecht parallelisierbar, weil die Synchronisierung für viele oder sogar alle Variablen nötig wäre, mithin das ganze Programm ein einziger riesiger kritischer Abschnitt ist.

Kritische Abschnitte erfordern gegenseitigen Ausschluß, was alle Synchronisationsmechanismen beherrschen. Mit Semaphoren sieht das wie folgt aus:

```
{  // einmal beim Programmstart nötig:
  mutex_semaphor = sem_init(1);
}
{  // der kritische Abschnitt wird mit P und V eingeklammert:
  sem_p(mutex_semaphor);
  ... // kritischer Abschnitt
  sem_v(mutex_semaphor);
}
```

Der kritische Abschnitt wird in P und V eingeklammert, und zwar mit einem Semaphor, das mit eins initialisiert wird. Ein derartig benutztes Semaphor heißt meist mutex für "mutual exclusive", und, falls mehrere davon auftauchen, meist verkettet mit dem Namen der Datenstruktur, die von diesem mutex geschützt wird, also z.B. ringbuffer_mutex. (Ein solches Semaphor nicht mutex zu nennen geht in die Richtung, als ob man die Laufvariable einer Schleife nicht i nennt.)

Der Gedankenschritt dabei ist: es gibt ein Betriebsmittel, das nur einmal verfügbar ist, weshalb das Semaphor mit eins initialisiert wird. Das zu schützende Betriebsmittel ist hier die Datenstruktur, also die einzelne Variable, die von diesem Semaphor geschützt wird.

Handshake

Wenn ein Prozess auf etwas wartet, was ein anderer Prozess bereitstellt, dann kommt ein Muster zum Einsatz, das keinen etablierten Namen hat und deshalb hier einen Namen bekommt: Handshake. (Entsprechend einer Technik aus der Hardware, wo zwei Geräte Daten mittels Handshake austauschen.)

```
{
  semaphor = sem_init(0);
}

{
  ... // Bereitstellen von Daten
  sem_v(semaphor); // Daten freigeben
}

{
  sem_p(semaphor); // warten auf Daten
  ... // Benutzen der Daten
}
```

Das Semaphor wird mit 0 initialisiert, mittels V auf eins hochgezählt, sobald Daten bereit sind. Der andere Prozess wartet mittels P bis ihm ein anderer Prozess mittels V mitteilt, dass Daten bereit sind.

Die Denkweise ist hier: Anfangs gibt es noch gar keine Daten, lies: gar kein Betriebsmittel (deshalb die Initialisierung mit 0), bis ein Prozess ein solches bereitstellt, und dies mit V bekannt macht.

Oft braucht man dieses Muster symmetrisch mit einem zweiten Semaphor, mit dem der zweite Prozess mitteilt, dass der Bereich, in dem Daten bereitgestellt wurden wieder frei ist.

Eine Menge gleichartiger Betriebsmittel

Zum Beispiel eine Druckerwarteschlange kann mehrere gleichartige Betriebsmittel enthalten, in diesem Fall Druckaufträge, die nacheinander ausgedruckt werden sollen. Mit Semaphoren verwaltet man derartiges wie folgt:

```
{
  anzahl_druckauftraege = sem_init(0);
}

{
  ... // Druckauftrag fertigmachen und in der Queue ablegen
  sem_v(anzahl_druckauftraege);
}

while (1) {
  sem_p(anzahl_druckauftraege);
  ... // Einen Druckauftrag aus der Queue holen und bearbeiten
}
```

Das Semaphor anzahl_druckauftraege dient dem Mitzählen der vorhandenen Druckaufträge. Die P-Operation wird dadurch zum Warten, bis wenigstens ein Druckauftrag vorliegt. Welcher Druckauftrag das ist, das entscheiden die Daten innerhalb der Queue, für deren Schutz ein weiteres Semaphor nötig ist, typisch ein Mutex-Semaphor. Letzteres schützt im Beispiel oben dann die Teile "in der Queue ablegen" und "aus der Queue holen". Das ausser dem zählenden Semaphor zusätzlich ein oder mehrere Mutex-Semaphore nötig sind ist typisch bei diesem Muster.

Die Denkweise bei diesem Muster ist, dass der Zähler des Semaphors direkt die Anzahl der Druckaufträge in der Warteschlange enthält, also mal wieder "die Anzahl der vorhandenen Betriebsmittel". Übrigens kann dieses Muster nicht ohne weiteres von binären Semaphoren oder Locks nachgebildet werden, weil das Zählen im Semaphor hier essentiell ist.

Das Framework

Für die Aufgaben ist ein Framework in der Sprache C vorgegeben, das unter Linux lauffähig ist, und unter p:\ti_prak\sync bzw. unter /import/grpdrvs/ti_prak/sync auch in den Rechnerpools der SGI bereitsteht. Im Praktikum liegt es unter /opt/sync. Dieses Framework kopiert man sich komplett in ein eigenes Verzeichnis, z.B. mittels "cp -a /opt/sync ~/sync". In diesem Verzeichnis ("cd sync") kann man dann z.B. für die Aufgabe 1 die Datei "lost_update.c" laufen lassen, einfach indem man "make lost_update" (ohne die Endung ".c" !!!) eingibt. Zum Ansehen/Ändern kann man mit "nano lost_update.c" einen Editor starten.

Die Datei lost_update.c für Aufgabe 1 enthält auch weitgehende Beschreibungen dessen, was man zur Bearbeitung dieses Versuchs benötigt. Parallel zum folgenden Text anschauen!

Nach einigen einleitenden #include Statements (hier nicht erklärt) und der Definition von Konstanten mit #define folgt der Bereich, in dem man globale Variablen deklarieren sollte. Derartige Variablen müssen mit "volatile" (dt. flüchtig) deklariert werden, damit der Compiler weiß, dass er diese Variablen im Speicher lassen muss. Ohne volatile könnte es passieren, dass der Compiler Variablen zur Optimierung nur im Prozessor hält, und damit eine Kommunikation zwischen Prozessen unmöglich wird.

Danach folgt die Methode test_setup(), die es ermöglicht, Vorgaben zu machen, z.B. welchen Startwert eine Variable haben soll. Diese Routine muss ausserdem unbedingt die beiden Variablen "readers" und "writers" setzen, entsprechend dem Wunsch, wieviele Prozesse für die Routinen "reader()" und "writer()" jeweils gestartet werden sollen. Setzt man z.B. readers=3 und writers=5, dann werden acht Prozesse parallel gestartet.

Diese beiden genannten Routinen reader() und writer() befinden sich am Ende der Datei lost_update.c, und beinhalten hier den Code für das Lost-Update Problem. Genauer: writer() alleine enthält den Code, da eine Routine für das Problem ausreicht. reader() ist hier leer und wird auch nie aufgerufen. Das Framework bietet zwei Routinen an, weil man bei vielen der folgenden Probleme zwei verschiedene Codestücke benötigt, die auf gemeinsamen Variablen arbeiten.

Beide Routinen erhalten als Parameter eine fortlaufende Nummer, damit sich gleichartige Prozesse damit unterscheiden können. Der erste Prozess, der reader() bearbeitet, erhält als Parameter eine 0 übergeben, der zweite eine 1, usw. Ebenso erhalten alle Prozesse für writer() einen Wert ab 0. Beim Lost-Update Problem werden die Parameter allerdings nicht benötigt und einfach ignoriert.

Erst wenn sich alle Prozesse wieder beendet haben, die reader() und writer() bearbeitet haben, dann wird noch test_end() aufgerufen, beispielsweise um Ergebnisse testen oder ausgeben zu können.

Zur Synchronisation gibt es Semaphore, genauer Zähler-Semaphore, bereitgestellt in der Datei semaphores.c. Einmal kurz anschauen sollte man sich eher die Datei semaphores.h, in der die Operationen auf Semaphoren aufgelistet sind. sem_init(int startwert) gibt einen Zeiger auf ein neues Semaphor zurück, das mit dem gegebenen Startwert initialisiert wurde. sem_p(semaphor) und sem_v(semaphor) sind die oben bereits beschriebenen Operationen. Die Routinen sem_t(semaphor) und sem_count(semaphor) werden im Praktikum nicht benötigt.

```
// Eine Variable vom Typ sempahor deklariert man damit wie folgt:
  semaphore mein_semaphor;
// Der unbedingt als erstes nötige Initialisierungsaufruf:
  mein_semaphor=sem_init(123); // 123 ist hier ein zufaelliger Wert
// Semaphor-Operationen ruft man danach z.B. wie folgt auf:
  sem_v(mein_semaphor);
```

Makefile, main.c und sempahores.c muss man nicht anschauen, sind aber auch kein Geheimnis.

Für alle Aufgaben gilt:

Für diejenigen, die noch nie C programmiert haben: nehmen Sie die vorgegebenen Quelltexte als Muster für Zuweisungen, Array-Zugriffe, Schleifen, etc. Wir erwarten hier nicht, dass sie die Sprache C ausreizen. Im Zweifelsfall den Betreuer fragen.

Grundsätzlich werden keine Lösungen akzeptiert, die in irgendeiner Form noch busy-wait machen, sofern die Aufgabe dies nicht explizit zuläßt.

Auch fehlerhaft synchronisierte Programme laufen oft fehlerfrei. Dies liegt daran, dass man "Glück" haben muss, dass der zeitliche Wechsel zwischen den Prozessen einen Fehler bewirkt. Ist ein nicht synchronisierter kritischer Abschnitt kurz, dann kann es durchaus passieren, dass auch tausende von Testläufen nie den Fehler bewirken. Der tritt erst ein, wenn das damit gesteuerte Atomkraftwerk seit 10 Jahren in Betrieb ist. In diesem Praktikum lassen Sie ein Programm nach einem erfolgreichen Lauf bitte immer gleich noch ein paar mal laufen, um wenigstens die Chancen zur Fehlererkennung ein bisschen zu erhöhen. Tritt nur ein einziger Fehler auf, dann deutet das definitiv auf fehlerhafte Synchronisation hin!

Man kann ein Programm immer mit Strg-C abbrechen. Dies ist z.B. nötig bei fehlerhafter Synchronisation, wenn ein Prozess für immer auf einem sem_p() hängen bleibt. Leider ist dieser Zustand nicht zu unterscheiden von einem Programm, dass still vor sich hin arbeitet. Hier deshalb der Hinweis, dass bei den Musterlösungen kein Programm dabei ist, das länger als eine Minute läuft.

Aufgabe 1: Der Klassiker "Lost Update"

Schauen Sie sich die vorgegebene Datei lost_update.c an. Die drei Prozesse (worker) sollen hier gemeinsam eine Variable hochzählen. Die Frage, die im Kolloquium sicherlich kommt: warum funktioniert das nicht korrekt?

Benutzen Sie dann ein Semaphor um den Fehler im Programm zu beheben. Tipp: eines der oben beschriebenen Muster paßt (fast) direkt.

Aufgabe 2: Ringpuffer

Ein Synchronisations-Sonderfall kommt in der Praxis häufig vor, weshalb er in dieser Aufgabe zum Thema wird: ein Ringpuffer, der mit nur genau einem Leser und nur genau einem Schreiber ganz ohne Synchronisationsmechanismen korrekt funktioniert. Der zugehörige Quelltext ist in der Datei "ringpuffer.c" vorgegeben. Erste Kolloquiumsfrage wird sein: warum funktioniert alles? Genauer: welche besonderen Umstände kommen hier alle(!) zusammen, dass das Programm fehlerfrei läuft?

Sobald man mehr als einen reader oder writer hat funktioniert das Programm nicht mehr korrekt. Erhöhen Sie deshalb in test_setup() die Werte für writers auf 2 oder mehr. Readers muss im Testprogramm auf eins bleiben, da die Animation sonst nicht mehr funktioniert. Mehrere Testläufe sollten zeigen, dass das Programm nun nicht mehr korrekt funktioniert. Führen Sie anschließend Semaphore ein, damit das Programm dann wieder funktioniert!

Aufgabe 3: Datenbank

Im Framework finden Sie die Datei 'datenbank.c', die im Quelltext auch die Aufgabenstellung als Kommentar enthält. Bearbeiten Sie Diese!

Aufgabe 4: Zehn zählende Prozesse

Schreiben Sie ein Programm, bei dem zehn Prozesse gemeinsam eine Variable auf 10000 hochzählen, wobei jeder Prozess nur hochzählen darf, wenn "seine" Endziffer von 0 bis 9 vorliegt.

Zum Testen, ob das Programm korrekt arbeitet, sollte jeder Prozess beim Hochzählen folgendes Statement enthalten:

```
printf("%i\n", zahl);
```

Die Ausgabe muss dann einfach zählen, ohne dass man sieht, welcher Prozess gerade die nächste Zahl erzeugt hat. (Gerne darf die Ausgabe dies zusätzlich enthalten.)

Es ist nicht nötig C zu lernen: starten Sie mit einem der vorgegebenen Programme, das Sie schon bearbeitet haben. Die nötigen Änderungen benötigen nichts spezielles der Sprache C, sondern sind vielmehr recht Java-ähnlich.

Wichtig ist, das Programm geschickt zu strukturieren. Beispielsweise sollen kritische Abschnitte, mit einem mutex geschützt, keine einzige Codezeile enthalten, die nicht zum kritischen Abschnitt gehört. Bei der Abnahme sollten Sie in der Lage sein für jede einzelne Codezeile Rede und Antwort zu stehen, warum sie an der jeweiligen Stelle steht (bezüglich der Semaphor-Operationen).

Natürlich sollten Sie sowieso in der Lage sein die Anordnung der Semaphor-Operationen zu Begründen. Busy-Wait ist als Synchronisationsmethode nicht erlaubt, kommt aber häufiger vor, auch aus Versehen.

Testen Sie (Pflicht!), ob Ihr Programm Synchronisations-Fehler erkennt, wenn alle Semaphor-Operationen auskommentiert sind!

Grundlagen der Betriebsysteme - Labor

Versuchsanleitung: Filesysteme

In diesem Versuch soll FAT32 als Beispiel für ein Filesystem näher betrachtet werden. Als Grundlage liegt dieser Anleitung ein leicht überarbeiteter Auszug des entsprechenden Artikels der englischen Wikipedia bei: (http://en.wikipedia.org/wiki/File_Allocation_Table). Die deutsche Wikipedia erklärt die wesentlichen Punkte ebenfalls, allerdings weniger ausführlich.

Ein Filesystem liegt normalerweise direkt auf einem Medium, also z.B. auf einem USB-Stick oder einer Festplattenpartition. Darauf wird im Praktikum aus verschiedenen Gründen verzichtet. Stattdessen wird ein Filesystem in einer Datei angelegt, welche unter Linux über ein sogenanntes Loopback-Device wie ein physikalisches Medium genutzt werden kann. Die Datei enthält hier also ein Filesystem so, dass es Bit-für-Bit auf eine Festplattenpartition kopiert ebenfalls funktionieren würde.

Im Versuch wird das (in einer Datei enthaltene) Filesystem abwechselnd entweder in den regulären Verzeichnisbaum des Betriebssystems eingebunden, oder es kann mit einem sogenannten Hex-

Editor bearbeitet werden. Beides gleichzeitig geht nicht, weil das Betriebssystem bei einem gemounteten Dateisystem Daten im Hauptspeicher zwischenspeichert, und davon ausgeht, dass es exklusiven Zugriff hat. Ansonsten würde es Probleme geben, siehe Synchronisationsversuch.

Für das Praktikum sind folgende Kommandos vorbereitet:

- **ti_makefs**
  erstellt im eigenen Home-Verzeichnis die Datei "ti_filesystem.fat32" neu, und zwar immer exakt den Anfangszustand, von dem aus jede Teilaufgabe bearbeitet werden kann. "ti_makefs" kann insbesondere auch benutzt werden, wenn durch fehlerhaftes Modifizieren das Dateisystem unbenutzbar wurde. Intern legt das Kommando die Datei neu an und erzeugt darin ein FAT32-Filesystem. Danach wird die Datei gemountet, einige vorgegebene Dateien hineinkopiert und zuletzt wird die Datei wieder ungemountet (die Fachjargon-Begriffe "mount" und "umount" werden weiter unten erklärt).
- **ti_hexedit**
  ermöglicht das Editieren der Datei "ti_filesystem.fat32" mit dem Hex-Editor *Hexedit* (im Sektor-Modus gestartet). Dies funktioniert nur, wenn das Filesystem gerade nicht gemountet ist. Eine Anleitung zu Hexedit findet sich unter http://merd.sourceforge.net/pixel/hexedit.html. Ein Auszug davon ist am Ende dieser Praktikumsanleitung zu finden.
- **ti_mount**
  erzeugt eine Kopie der Datei "ti_filesystem.fat32" unter "/tmp/filesystem_von_" und mountet diese unter "/mnt/". Man kann nun mit den normalen Unix-Kommandos wie z.B. "ls /mnt/" schauen, ob evtl. gemachte Änderungen vom FAT32-Treiber des Betriebssystems verstanden werden. Um anschließend wieder mit den anderen Kommandos arbeiten zu können, muß man "/tmp/filesystem_von_" unmounten mit dem folgenden Kommando:
- **ti_umount**
  unmountet "/tmp/filesystem_von_" von "/mnt/". Die Datei kann danach wieder bearbeitet werden.

Die Begriffe "mounten" und "unmounten" sind von den Unix-Kommandos "mount" und "umount" abgeleitet.
mount ermöglich die Einbindung eines Mediums, z.B. eines USB-Sticks, in eine beliebige Stelle des Verzeichnisbaums des Betriebsystems (unter Windows geschieht automatisch etwas ähnliches, wenn ein USB-Stick z.B. unter E: verfügbar gemacht wird).
umount kehrt diesen Prozess um, der entsprechende Teilbaum wird also aus dem Verzeichnissystem entfernt.

/mnt/ ist ein leeres Verzeichnis solange dort nichts gemountet ist. (Oder es fehlt sogar ganz.) Wenn "ls /mnt/" ein leeres Verzeichnis anzeigt, dann deutet das darauf hin, daß die Datei nicht gemountet

ist, dies z.B. aufgrund von Fehlern nicht geklappt hat, oder schlicht das Kommando *ti_mount* vergessen wurde.

Eine beliebig fehlerhafte Datei, die als Filesystem gemountet werden soll, könnte evtl. den Treiber von Linux überfordern, so dass der Rechner beim Kommando "ti_mount" abstürzt. Dies ist bisher zwar nie vorgekommen, aber als Vorsichtsmaßnahme sollte man während diesem Versuch keine wichtigen Dateien auf dem eigenen home-Laufwerk öffnen.

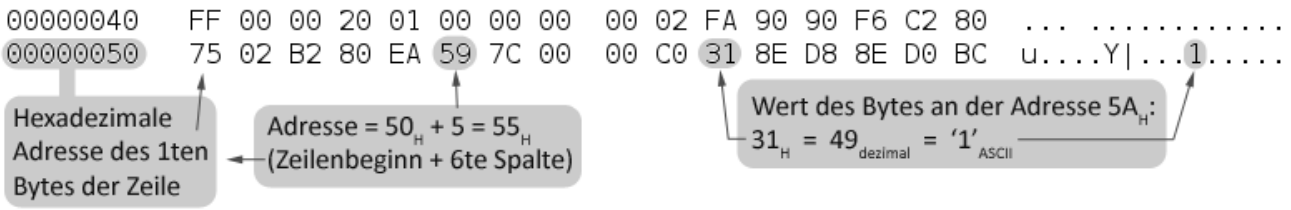
Hex-Editor-Einführung

Eine Datei kann als Kette von Bytes gesehen werden. Bei Texteditoren werden diese Bytes als Zeichen (Buchstaben, Zahlen) nach einer festgelegten Kodierung wie etwa ASCII aufgefasst und dem Benutzer so angezeigt. Eine Sonderstellung nehmen dabei nicht druckbare Zeichen wie z.B. das Zeichen für Zeilenumbruch ein. Für andere Dateien existieren andere Programme zum Anschauen und/oder Bearbeiten, wie z.B. Bildbetrachter für *.jpg oder *.bmp-Dateien.

Bei manchen Dateien existiert jedoch gar kein sinnvolles, passendes Programm. Sei es, dass die Datei defekt ist oder rein als Zwischenspeicher für programminterne Daten gedacht ist. Muss man eine solche "unsinnige" Datei bearbeiten, dann helfen Hex-Editoren.

Hex-Editoren ähneln normalen Texteditoren. Beide können den Inhalt einer Datei anzeigen und verändern, jedoch zeigen sie den Inhalt auf unterschiedliche Art an. Hex-Editoren zeigen die Bits in hexadezimaler Darstellung an, ohne irgend etwas über den Sinn der Daten anzunehmen. Nicht einmal das Zeichen für Zeilenumbruch bewirkt den Wechsel in eine neue Zeile, was auch sinnvoll ist, weil der Hex-Editor nicht einmal davon ausgehen kanbn, dass es sich um Text handelt.

Der hier im Praktikum verwendete hexedit zeigt Dateien wie folgt an:



Links wird die Startadresse der Zeile in hexadezimaler Form angegeben. In der Mitte befindet sich der wichtigste Teil, der Inhalt in Zahlendarstellung. Dabei wird jedes Byte (8 Bit) platzsparend als hexadezimales (4 Bit pro Ziffer, 2 Ziffern) Zahlenpaar dargestellt. Die 256 möglichen Werte eines Bytes erscheinen damit als 00 – FF. Ganz rechts folgt der Zeileninhalt wie in einem Texteditor, wobei nicht darstellbare Zeichen i. d. R. durch einen Punkt repräsentiert werden.

Das Ändern des Dateiinhalts ist nun wie Folgt möglich: Entweder man überschreibt in der Mitte eine Hexadezimalzahl mit einem neuen Wert, oder — sofern die neuen Daten einer Zeichenkette entsprechen — ändert in der rechten Spalte genau wie in einem Texteditor den bestehenden Text. Zu beachten ist, dass als Operation i. d. R. nur Überschreiben und Einfügen nicht möglich ist. Dies hängt vor allem damit zusammen, dass das Verschieben des gesamten nachfolgenden Dateiinhalts keine triviale Operation ist.

Weitere Aspekte wie Navigation in der Datei, Suche usw. sind in vielerlei Hinsicht ähnlich wie bei Texteditoren und können der Referenz des Editors entnommen werden. Wie bereits erwähnt, befinden sich die wichtigsten Befehle für den im Praktikum verwendeten Editor am Ende dieser Anleitung.

Weitere Hinweise

Für diesen Versuch könnte ein Taschenrechner hilfreich sein, insbesondere einer, der Hex-Zahlen beherrscht. Der Versuch ist aber auch ohne machbar.

Beachten Sie bitte auch das Thema Byte-Reihenfolge (**Endianness**), nachzulesen etwa bei Wikipedia. Bei vielen Formaten, auch FAT32, werden Zahlen in der Form *Little-Endian* geschrieben. Byte-weise im Hex-Editor erscheint die Zahl dann verdreht, d.h. dass z.B. die 16-Bit-Zahl "12AF" als "AF 12" auftaucht.

Die Suchfunktion des Editors darf man natürlich benutzten. Bei der Abnahme wird allerdings immer eine algorithmische/verallgemeinerbare Lösung gefordert, die z.B. von einem Dateisystemtreiber implementiert werden könnte. Durch die Suchfunktion gefundene Adressen erfüllen diesen Anspruch nicht.

Aufgabe 0: Einarbeitung in die Thematik

Diese Aufgabe wird nicht für sich abgenommen, sondern dient dem Einstieg in die Aufgabe. Die Fragen hier können in den folgenden Aufgaben als Kolloquiumsfragen auftauchen und zudem als Lösungshinweise zu diesen gesehen werden.

- Eine Einarbeitung in Hexedit ist sinnvoll. Zuerst könnten etwa die Funktionen wie Sprung-an-eine-Adresse usw. an einer kleinen Textdatei ausprobiert werden. Dazu kann Hexedit so gestartet werden: "hexedit <dateiname>"
- Es lohnt sich in die Beschreibung der Tabellen für FAT32 (im Anhang dieser Anleitung) die Werte einzutragen, die man mit dem Hex-Editor in der Datei "/tmp/filesystem_von_" vorfindet. Es erspart einem später Arbeit beim Rechnen.

- Wie findet ein FAT32-Treiber mittels der Verwaltungsdaten die Größe eines Sektors/Clusters?
- Wie stehen die drei Einheiten Sektornummer, Clusternummer und Dateiadresse im Zusammenhang? Wie lassen sich die Einheiten umrechnen?
- Wie errechnet sich der Beginn der ersten/zweiten FAT?
- Wo beginnt der erste Cluster auf dem Medium und was ist seine Nummer (Hinweis: es ist weder 0 noch 1)? Wo der nächste?
- Wie kann ein Treiber das Wurzelverzeichnis finden?
- Wie sieht ein Verzeichniseintrag aus?

Aufgabe 1: Ändern eines Dateinamens

Ändern Sie den Namen der Datei "falsch.txt" im Wurzelverzeichnis um in "richtig.txt". Zusätzlich ändern Sie den Datumsstempel der Datei auf den 6.12.2012 (6. Dezember 2012).

Aufgabe 2: Modifizieren des Dateiinhalts

Kürzen Sie die Datei "lang.txt" im Wurzelverzeichnis von 4148 auf 4032 Bytes!

Beachten Sie, dass nicht nur im Verzeichniseintrag, sondern auch in den beiden FATs Änderungen nötig sind.

Aufgabe 3: Freien Plattenplatz ermitteln und Abschlusskolloquium

- Ermitteln Sie die nutzbare Größe des gesamten Filesystems anhand der Informationen, die Sie mit dem Hex-Editor erhalten (Lösungsweg ist auch gefragt). Welche Einheit ist für die Angabe am sinnvollsten?
- Ermitteln Sie den belegten und den freien Platz, der bei der aktuellen Belegung noch zur Verfügung steht.
- Die Kontrolle des Wertes ist bei gemountetem Filesystem mit dem Unix-Kommando "df -B <Einheitengröße>" möglich.

Zum Abschlusskolloquium sollten Sie diese Aufgabe gelöst haben und sich mit der Materie ausreichend vertraut gemacht haben, um dann im Stegreif weitergehende Fragen beantworten zu können.

FAT32-Dokumentation

Es folgt ein Auszug aus http://en.wikipedia.org/wiki/File_Allocation_Table, teilweise mit [Modifikationen] versehen, insbesondere Kürzung vieler FAT12-/FAT16-spezifischer Passagen.

Design

The following is an overview of the order of structures in a FAT partition or disk:

Contents	Boot Sector	FS Information Sector (FAT32 only)	More reserved sectors (optional)	File Allocation Table #1	File Allocation Table #2	Root Directory (FAT12/16 only)	Data Region (for files and directories) ... (To end of partition or disk)
Size in sectors	(number of reserved sectors)			(number of FATs)*(sectors per FAT)		[...]	NumberOfClusters*SectorsPerCluster

A FAT file system is composed of four different sections.

1. The **Reserved sectors**, located at the very beginning. The first reserved sector (sector 0) is the Boot Sector (aka *Partition Boot Record*). It includes an area called the *BIOS Parameter Block* (with some basic file system information, in particular its type, and pointers to the location of the other sections) and usually contains the operating system's boot loader code. The total count of reserved sectors is indicated by a field inside the Boot Sector. [...]
2. The **FAT Region**. This typically contains two copies (may vary) of the *File Allocation Table* for the sake of redundancy checking, although the extra copy is rarely used, even by disk repair utilities. These are maps of the Data Region, indicating which clusters are used by files and directories. In FAT16 and FAT12 they immediately follow the reserved sectors.
3. The **Root Directory Region**. This is a *Directory Table* that stores information about the files and directories located in the root directory. It is only used with FAT12 and FAT16, and imposes on the root directory a fixed maximum size which is pre-allocated at creation of this volume. FAT32 stores the root directory in the Data Region, along with files and other directories, allowing it to grow without such a constraint. Thus, for FAT32, the Data Region starts here.
4. The **Data Region**. This is where the actual file and directory data is stored and takes up most of the partition. The size of files and subdirectories can be increased arbitrarily (as long as there are free clusters) by simply adding more links to the file's chain in the FAT. Note however, that files are allocated in units of clusters, so if a 1 kB file resides in a 32 kB cluster, 31 kB are wasted. FAT32 typically commences the Root Directory Table in cluster number 2: the first cluster of the Data Region.

FAT uses little endian format for entries in the header and the FAT(s). It is possible to allocate more FAT sectors than necessary for the number of clusters. The end of the last FAT sector can be unused if there are no corresponding clusters. The total number of sectors (as noted in the boot record) can be larger than the number of sectors used by data (clusters × sectors per cluster), FATs (number of FATs × sectors per FAT), and hidden sectors including the boot sector — this would result in unused sectors at the end of the volume. If a partition contains more sectors than the total number of sectors occupied by the file system it would also result in unused sectors at the end of the volume.

Boot Sector

On non-partitioned devices, e.g., floppy disks, the boot sector is the first sector. For partitioned devices such as hard drives, the first sector is the Master Boot Record defining partitions, while the first sector of partitions formatted with a FAT file system is again the FAT boot sector.

Common structure of the first 36 bytes used by all FAT versions are:

Byte Offset	Length (bytes)	Description	[wert]
0x00	3	Jump instruction. This instruction will be executed and will skip past the rest of the (non-executable) header if the partition is booted from. See Volume Boot Record. If the jump is two-byte near jmp it is followed by a NOP instruction (hex. EB??90)	EB 58 90
0x03	8	OEM Name (padded with spaces 0x20). This value determines in which system disk was formatted. MS-DOS checks this field to determine which other parts of the boot record can be relied on. Common examples are IBM 3.3, MSDOS5.0, MSWIN4.1, mkdosfs, and FreeDOS.	6D 6B 66 73 2E 66 61 74
0x0B	2	Bytes per sector; the most common value is 512. The BIOS Parameter Block starts here.	00 02
0x0D	1	Sectors per cluster. Allowed values are powers of two from 1 to 128.	08
0x0E	2	Reserved sector count. The number of sectors before the first FAT in the file system image. At least 1 for this sector, usually 32 for FAT32.	20 00
0x10	1	Number of file allocation tables. Almost always 2; RAM disks might use 1.	02
0x11	2	Maximum number of FAT12 or FAT16 root directory entries. 0 for FAT32, where the root directory is stored in ordinary data clusters.	00 00
0x13	2	Total sectors (if zero, use 4 byte value at offset 0x20)	20 03
0x15	1	Media Descriptor Byte [gekürzt]	F8

0x16	2	Sectors per File Allocation Table for FAT12/FAT16, 0 for FAT32 (cf. offset 0x24 below)	00 00
0x18	2	Sectors per track for disks with geometry, e.g., 18 for a 1.44MB floppy	20 00
0x1A	2	Number of heads for disks with geometry, e.g., 2 for a double sided floppy	40 00
0x1C	4	Count of hidden sectors preceding the partition that contains this FAT volume. This field should always be zero on media that are not partitioned.	00 00 00 00
0x20	4	Total sectors (if greater than 65535; otherwise, see offset 0x13)	00 00 00 00

Further structure used by FAT32:

Byte Offset	Length (bytes)	Description	[wert]
0x24	4	Sectors per file allocation table	01 00 00 00
0x28	2	FAT Flags (Only used during a conversion from a FAT12/16 volume.)	00 00
0x2A	2	Version (Defined as 0)	00 00
0x2C	4	Cluster number of root directory start	02 00 00 00
0x30	2	Sector number of FS Information Sector	01 00
0x32	2	Sector number of a copy of this boot sector (0 if no backup copy exists)	06 00

0x34	12	Reserved	00 00 00 00 00 00 00 00 00 00 00 00
0x40	1	Physical Drive Number	80
0x41	1	Reserved	00
0x42	1	Extended boot signature	29
0x43	4	ID (serial number)	CF E7 A5 33
0x47	11	Volume Label [ungenutzt]	-
0x52	8	FAT file system type: "FAT32 "	46 41 54 33 32 20 20 20
0x5A	420	Operating system boot code	–
0x1FE	2	Boot sector signature (hex. 55AA)	**55 AA**

A simple formula translates a given cluster number `CN` to a logical sector number `LSN`:

1. Determine (once) `SSA=RSC+FN×SF+ceil((32×RDE)/SS)`, where the reserved sector count `RSC` is stored at offset 0x0E, the FAT number `FN` at offset 0x10, the sectors per FAT `SF` at offset 0x16 (FAT12/16) or 0x24 (FAT32), the root directory entries `RDE` at offset 0x11, the sector size `SS` at offset 0x0B, and `ceil(x)` rounds up to a whole number.
2. Determine `LSN=SSA+(CN-2)×SC`, where the sectors per cluster `SC` are stored at offset 0x0D.

A translation of CHS to `LSN` is also simple: `LSN=SPT×(HN+(NOS×TN))+SN-1`, where the sectors per track `SPT` are stored at offset 0x18, and the number of sides `NOS` at offset 0x1A. Track number `TN`, head number `HN`, and sector number `SN` correspond to Cylinder-head-sector — the formula gives the known CHS to LBA translation.

File Allocation Table

A partition is divided up into identically sized **clusters**, small blocks of contiguous space. Cluster sizes vary depending on the type of FAT file system being used and the size of the partition, typically cluster sizes lie somewhere between 2 kB and 32 kB. Each file may occupy one or more of these clusters depending on its size; thus, a file is represented by a chain of these clusters (referred

to as a singly linked list). However these clusters are not necessarily stored adjacent to one another on the disk's surface but are often instead *fragmented* throughout the Data Region.

The **File Allocation Table** (**FAT**) is a list of entries that map to each cluster on the partition. Each entry records one of five things:

- the cluster number of the next cluster in a chain
- a special *end of clusterchain* (*EOC*) entry that indicates the end of a chain
- a special entry to mark a bad cluster
- a zero to note that the cluster is unused

The first two entries in a FAT store special values: The first entry contains a copy of the media descriptor (from boot sector, offset 0x15). The remaining 8 bits (if FAT16), or 20 bits (if FAT32) of this entry are 1.

The second entry stores the end-of-cluster-chain marker. The high order two bits of this entry are sometimes, in the case of FAT16 and FAT32, used for dirty volume management: high order bit 1: last shutdown was clean; next highest bit 1: during the previous mount no disk I/O errors were detected.

Because the first two FAT entries store special values, there is no cluster 0 or 1. The first cluster (after the root directory if FAT 12/16) is cluster 2.

FAT entry values [FAT32]:

FAT32	Description
0x00000000	Free Cluster
0x00000001	Reserved, do not use
0x00000002-0x0FFFFFEF	Used cluster; value points to next cluster
0x0FFFFFF0-0x0FFFFFF5	Reserved in some contexts, or also used
0x0FFFFFF6	Reserved; do not use
0x0FFFFFF7	Bad sector in cluster or reserved cluster

0x0FFFFFF8-0x0FFFFFFF	Last cluster in file (EOC)

Note that FAT32 uses only 28 bits of the 32 possible bits. The upper 4 bits are usually zero (as indicated in the table above) but are reserved and should be left untouched.

Each version of the FAT file system uses a different size for FAT entries. Smaller numbers result in a smaller FAT, but waste space in large partitions by needing to allocate in large clusters. [...]

Directory table

A **directory table** is a special type of file that represents a directory (also known as a folder). Each file or directory stored within it is represented by a 32-byte entry in the table. Each entry records the name, extension, attributes (archive, directory, hidden, read-only, system and volume), the date and time of creation, the address of the first cluster of the file/directory's data and finally the size of the file/directory. Aside from the Root Directory Table in FAT12 and FAT16 file systems, which occupies the special *Root Directory Region* location, all Directory Tables are stored in the Data Region. The actual number of entries in a directory stored in the Data Region can grow by adding another cluster to the chain in the FAT.

Note that before each entry there can be "fake entries" to support the Long File Name. (See further down the article).

[...]

<table>
<tr><th>Byte Offset</th><th>Length (bytes)</th><th>Description</th></tr>
<tr><td>0x00</td><td>8</td><td>DOS file name (padded with spaces)

The first byte can have the following special values:

<table>
<tr><td>0x00</td><td>Entry is available and no subsequent entry is in use</td></tr>
<tr><td>0x05</td><td>Initial character is actually 0xE5.</td></tr>
<tr><td>0x2E</td><td>'Dot' entry; either '.' or '..'</td></tr>
<tr><td>0xE5</td><td>Entry has been previously erased and is available.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>0x08</td><td>3</td><td>DOS file extension (padded with spaces)</td></tr>
<tr><td>0x0B</td><td>1</td><td>File Attributes [mostly ignored by Linux]

Bit Mask Description

<table>
<tr><td>0</td><td>0x01</td><td>Read Only</td></tr>
<tr><td>1</td><td>0x02</td><td>Hidden</td></tr>
<tr><td>2</td><td>0x04</td><td>System</td></tr>
<tr><td>3</td><td>0x08</td><td>Volume Label</td></tr>
<tr><td>4</td><td>0x10</td><td>Subdirectory</td></tr>
<tr><td>5</td><td>0x20</td><td>Archive</td></tr>
<tr><td>6</td><td>0x40</td><td>Device (internal use only, never found on disk)</td></tr>
<tr><td>7</td><td>0x80</td><td>Unused</td></tr>
</table>
An attribute value of 0x0F is used to designate a long file name entry.</td></tr>
<tr><td>0x0C</td><td>10</td><td>Reserved [gekürzt]</td></tr>
<tr><td>0x16</td><td>2</td><td>Last modified time. The hour, minute and second are encoded according to the following bitmap:

Bits Description

<table>
<tr><td>15-11</td><td>Hours (0-23)</td></tr>
<tr><td>10-5</td><td>Minutes (0-59)</td></tr>
<tr><td>4-0</td><td>Seconds/2 (0-29)</td></tr>
</table>
Note that the seconds is recorded only to a 2 second resolution. }</td></tr>
<tr><td>0x18</td><td>2</td><td>Last modified date. The year, month and day are encoded according to the following bitmap:</td></tr>
</table>

<table>
<tr><td></td><td></td><td>Bits Description
<table>
<tr><td>15-9</td><td>Year (0 = 1980, 127 = 2107)</td></tr>
<tr><td>8-5</td><td>Month (1 - 12)</td></tr>
<tr><td>4-0</td><td>Day (1 - 31)</td></tr>
</table></td></tr>
<tr><td>0x1A</td><td>2</td><td>[...] Low 2 bytes of first cluster in FAT32. Entries with the Volume Label flag, subdirectory ".." pointing to root, and empty files with size 0 should have first cluster 0.</td></tr>
<tr><td>0x1C</td><td>4</td><td>File size in bytes. Entries with the Volume Label or Subdirectory flag set should have a size of 0.</td></tr>
</table>

[Einschub aus der deutschen Wikipedia: Soll nun eine Datei gelesen werden, wird der zugehörige Verzeichniseintrag herausgesucht. Neben den Attributen kann hier nun der Startcluster selektiert werden. Die weiteren Cluster werden dann über die FAT herausgesucht. Am Ende terminiert die Weitersuche durch einen FAT-Tabelleneintrag mit dem Wert FFFFFFh.]

Long file names

Long File Names (LFN) are stored on a FAT file system using a trick—adding (possibly multiple) additional entries into the directory before the normal file entry. The additional entries are marked with the Volume Label, System, Hidden, and Read Only attributes (yielding 0x0F), which is a combination that is not expected in the MS-DOS environment, and therefore ignored by MS-DOS programs and third-party utilities. Notably, a directory containing only volume labels is considered as empty and is allowed to be deleted; such a situation appears if files created with long names are deleted from plain DOS.

Each phony entry can contain up to 13 UTF-16 characters (26 bytes) by using fields in the record which contain file size or time stamps (but not the starting cluster field, for compatibility with disk utilities, the starting cluster field is set to a value of 0. See 8.3 filename for additional explanations). Up to 20 of these 13-character entries may be chained, supporting a maximum length of 255 UTF-16 characters.

After the last UTF-16 character, a 0x00 0x00 is added. The remaining unused characters are filled with 0xFF 0xFF.

LFN entries use the following format:

Byte Offset	Length (bytes)	Description
0x00	1	Sequence Number
0x01	10	Name characters (five UTF-16 characters)
0x0B	1	Attributes (always 0x0F)
0x0C	1	Reserved (always 0x00)
0x0D	1	Checksum of DOS file name
0x0E	12	Name characters (six UTF-16 characters)
0x1A	2	First cluster (always 0x0000)
0x1C	4	Name characters (two UTF-16 characters)

If there are multiple LFN entries, required to represent a file name, firstly comes the *last* LFN entry (the last part of the filename). The sequence number also has bit 6 (0x40) set (this means the last LFN entry, however it's the first entry seen when reading the directory file). The last LFN entry has the largest sequence number which decreases in following entries. The *first* LFN entry has sequence number 1. Bit 7 (0x80) of the sequence number is used to indicate that the entry is deleted.

For example if we have filename "File with very long filename.ext" it would be formatted like this:

Sequence number	Entry data
0x03	"me.ext"
0x02	"y long filena"
0x01	"File with ver"

???	Normal 8.3 entry

A checksum also allows verification of whether a long file name matches the 8.3 name; such a mismatch could occur if a file was deleted and re-created using DOS in the same directory position. [...]

If a filename contains only lowercase letters, or is a combination of a lowercase *basename* with an uppercase *extension*, or vice-versa; and has no special characters, and fits within the 8.3 limits, a VFAT entry is not created on Windows NT and later versions of Windows such as XP. Instead, two bits in byte 0x0c of the directory entry are used to indicate that the filename should be considered as entirely or partially lowercase. Specifically, bit 4 means lowercase *extension* and bit 3 lowercase *basename*, which allows for combinations such as "`example.TXT`" or "`HELLO.txt`" but not "`Mixed.txt`". Few other operating systems support it. This creates a backwards-compatibility problem with older Windows versions (95, 98, ME) that see all-uppercase filenames if this extension has been used, and therefore can change the name of a file when it is transported between OSes, such as on a USB flash drive. Current 2.6.x versions of Linux will recognize this extension when reading (source: kernel 2.6.18 /fs/fat/dir.c and fs/vfat/namei.c); the mount option *shortname* determines whether this feature is used when writing. [...]

Bedienung von Hexedit

Auszug aus http://merd.sourceforge.net/pixel/hexedit.html

COMMANDS (full and detailed)

Right-Arrow, Left-Arrow, Down-Arrow, Up-Arrow	move the cursor
Ctrl+F, Ctrl+B, Ctrl+N, Ctrl+P	move the cursor
Ctrl+Right-Arrow, Ctrl+Left-Arrow, Ctrl+Down-Arrow, Ctrl+Up-Arrow	move n times the cursor
Esc+Right-Arrow, Esc+Left-Arrow, Esc+Down-Arrow, Esc+Up-Arrow	move n times the cursor

Esc+F, Esc+B, Esc+N, Esc+P	move n times the cursor
Home, Ctrl+A	go the beginning of the line
End, Ctrl+E	go to the end of the line
Page up, Esc+V, F5	go up in the file by one page
Page down, Ctrl+V, F6	go down in the file by one page
<, Esc+<, Esc+Home	go to the beginning of the file
>, Esc+>, Esc+End	go to the end of the file (for regular files that have a size)
Ctrl+Z	suspend hexedit
Ctrl+U, Ctrl+_, Ctrl+/	undo all (forget the modifications)
Ctrl+Q	read next input character and insert it (this is useful for inserting control characters and bound keys)
Tab, Ctrl+T	toggle between ASCII and hexadecimal
/, Ctrl+S	search forward (in ASCII or in hexadecimal, use TAB to change)
Ctrl+R	search backward
Ctrl+G, F4	go to a position in the file
Return	go to a sector in the file if --sector is used, otherwise go to a position in the file
Esc+L	display the page starting at the current cursor position

F2, Ctrl+W	save the modifications
F1, Esc+H	help (show the man page)
Ctrl+O, F3	open another file
Ctrl+L	redisplay (refresh) the display (usefull when your terminal screws up)
Backspace, Ctrl+H	undo the modifications made on the previous byte
Esc+Ctrl+H	undo the modifications made on the previous bytes
Ctrl+Space, F9	set mark where cursor is
Esc+W, Delete, F7	copy selected region
Ctrl+Y, Insert, F8	paste (yank) previously copied region
Esc+Y, F11	save previously copied region to a file
Esc+I, F12	fill the selection with a string
Esc+T	truncate the file at the current location
Ctrl+C	unconditional quit (without saving)
F10, Ctrl+X	quit

For the Esc commands, it sometimes works to use Alt instead of Esc. Funny things here (especially for froggies :) egrave = Alt+H , ccedilla = Alt+G, Alt+Y = ugrave.