Grundlagen der Rechnerarchitektur - Labor
Grundlagen der Betriebssysteme - Labor

Einführung in die Arbeitsumgebung des Labors

Zur Durchführung der Versuche steht an jedem Arbeitsplatz ein Rechner mit Linux (ohne grafische Oberfläche) bereit. Man kann sich dort wie in den Linux- oder Windows-Pools der SGI einloggen, also mit einem Accountnamen wie z.B. 'mm123'. Das eigene Homeverzeichnis findet man unter /home/<accountname>, also genau wie im Linux-Pool.

Während dem Semester werden einige Computer im Praktikum ständig laufen, so dass man auch z.B. von zu Hause oder von Pool-Maschinen aus mittels ssh die Praktikumsumgebung zur Verfügung hat. Hierzu benötigt man ein Terminalprogramm wie putty (in den Windows-Pools verfügbar) oder einfach das Kommando "ssh" einer Linux-Maschine. Damit kann man sich remote auf den dauernd eingeschalteten Maschinen einloggen. Welche Maschinen dauernd eingeschaltet sind entnehmen sie bitte der Web-Seite zum Praktikum, oder Sie probieren einfach aus, ob eine der Maschinen tip13, tip14, tip15 oder tip16 an ist. Typisch laufen diese in der Vorlesungszeit durchgehend.

Eine minimale Einführung in das Arbeiten mit der Kommandozeile unter Linux

Gedacht für absolute Neulinge zum Überleben der Kommandozeile

Zum Umgang mit der Kommandozeile von Linux existieren im Internet zahllose Tutorials, wie z.B. ein (englisches) unter http://linuxcommand.org/lc3_learning_the_shell.php. Da ich aber kein Tutorial finden konnte, das ich Empfehlen würde, existiert diese kurze Einleitung. Einerseits wird hier versucht die "denkweise" zu vermitteln, die Kommandozeilen eigen ist, andererseits ist das Ziel schnell bis zu den Kommandos zu kommen, die man zum Überleben anfangs benötigt.

Das Grundprinzip

Das Grundprinzip von Kommandzeilen ist wie folgt: man tippt ein paar Zeichen ein und drückt dann die Eingabe-Taste. Die Eingabe wird daraufhin in Wörter zerlegt, und das erste Wort muss ein Programm sein, das aufgerufen werden soll. Alle weiteren Wörter werden diesem Programm als Parameter übergeben (einfach als Wörter). Der Trick der Kommandozeile ist also eher: Welche Kommandos gibt es und was bedeuten weitere Wörter für dieses Kommando?

Ein einfaches Beispiel: Tippt man "echo huhu" ein (ohne die Gänsefüßchen), dann wird das Programm echo aufgerufen. Dieses Programm gibt seine Parameter einfach wieder aus: Es erscheint "huhu" auf dem Bildschirm.

Noch ein Beispiel: "rm beispiel.txt" löscht die bezeichnete Datei. "rm" ist das Programm remove (auf rm abgekürzt, damit man nicht so viel tippen muß), das seinen Parameter, das Wort beispiel.txt, als Dateinamen versteht.

Bei der Eingabe des Kommandos gibt es viele Editierhilfen, von denen man anfangs aber nicht viele braucht: die Pfeiltasten rechts-links, und die Backspace-Taste (über der Eingabetaste) um das Zeichen vor dem Cursor zu löschen reichen am Anfang aus. Viel Tipparbeit sparen einem die Pfeiltasten rauf-runter, die alte, bereits ausgeführte Kommandos wieder abrufen, leider je nach verwendeter Umgebung im Detail unterschiedlich: ausprobieren!

Dateinamen bzw. Pfade

Viele Programme erwarten Namen von Dateien oder Verzeichnissen als Parameter. Wichtig für die Kommandozeile ist es daher, dass man versteht, wie man Dateien auf der Kommandozeile spezifiziert, weil dies für alle Kommandos identisch passiert.

Alle Dateien sind in Verzeichnissen einsortiert, die wiederum einen Baum aufspannen. Der Windows-Explorer stellt genau einen solchen Verzeichnisbaum graphisch dar, um mit der Maus darin zu navigieren.

Unter Linux heißt das oberste Verzeichnis "/" (nur der schräge Strich). Darunter haben Verzeichnisse Namen und dürfen auch wieder Unterverzeichnisse haben. "/home/xy13" ist also im Wurzelverzeichnis "/" das Unterverzeichnis "home", und darin das Unterverzeichnis "xy13". Einzelne Dateien hängt man einfach hinten an einen solchen Pfad an, also z.B. "/home /xy13/meine_datei.txt" spezifiziert im genannten Verzeichnis die Datei meine_datei.txt. Ein Schrägstrich / ganz vorne ist der Name des Wurzelverzeichnisses, an späteren Stellen im Pfad dient er nur der Trennung der Verzeichnis- und Dateinamen.

Welches Unterverzeichnis wofür gut ist, ist im Prinzip frei wählbar, nur Konventionen und Erfahrung helfen hier weiter. Obiges Beispiel /home/xy13 hätte gute Chancen das Heimatverzeichnis des Users xy13 zu sein. Glücklicherweise braucht man anfangs die meisten Konventionen gar nicht zu kennen, sondern macht einfach im eigenen Home-Verzeichnis was man will.

Kommandozeilen unterstützen die Arbeit mit Verzeichnissen mit einem sogenannten "current directory", also einem aktuellen Verzeichnis. Man kann dieses wechseln, indem man das Kommando "cd", kurz für "change directory" benutzt. Mit "cd /home/xy13" sagt man also, dass man sein Home-Verzeichnis als aktuelles Verzeichnis haben will. Der Witz am "aktuellen Verzeichnis" ist, dass, wenn man einen Verzeichnis- oder Dateinamen NICHT mit / beginnt, dann wird automatisch das aktuelle Verzeichnis davor "gedacht". Ist also das aktuelle Verzeichnis "/home/xy13", dann kann man die Datei meine_datei.txt einfach genau so angeben. Der Mechanismus um das aktuelle Verzeichnis spart viel Tipparbeit. Im Fachjargon spricht man von "absoluter Pfad", wenn man eine Datei oder ein Verzeichnis mit einem / vorne benennt, oder "relativer Pfad", wenn man dies nicht tut und somit vom current directory aus startet.

Noch ein paar weiterführende Tipps zu Dateien und Verzeichnissen, die man aber grundsätzlich erst mal nicht braucht:

- Die Tab-Taste ist üblicherweise mit einer "Dateinamen-Vervollständigung" belegt. Tippt man an einem Dateinamen, ist noch nicht fertig, und drückt dann auf die Tab-Taste, dann sucht der Mechanismus den halb eingetippten Dateinamen im Verzeichnissystem und vervollständigt den Namen. Bleiben mehrere Möglichkeiten, dann passiert meist irgend etwas, was einem weiterhilft, evtl. auch erst beim zweiten Druck auf die Tab-Taste. Details: ausprobieren!
- Das Zeichen * steht in einem Dateinamen für beliebige andere Zeichen. Tippt man also z.B. "rm meine_d*.txt", dann wird ziemlich sicher die Datei meine_datei.txt gelöscht, aber eine Datei mit dem Namen meine_daten.txt auch! (Wie der Stern genau funktioniert wird hier nicht weiter beschrieben.)
- Jedes Verzeichnis hat immer einen Eintrag mit dem Namen ".." (zwei Punkte), der jeweils auf das Verzeichnis "eins weiter oben" zeigt. Einfach weil's praktisch ist: z.B. mit "cd .." setzt man immer das aktuelle Verzeichnis auf das Verzeichnis eins weiter oben im Baum. Aber man kann das auch mitten in einer Pfadangabe nutzen: "rm ../anderes_subdir/hallo.txt" löscht hallo.txt in einem Verzeichnis anderes_subdir, das parallel zum aktuellen Verzeichnis liegt. Und sogar so etwas geht:
  "rm ../anderes_subdir/../noch_ein_anderes_paralleles_subdir/xxx.txt". Der Anteil "anderes_subdir/.." nullt sich hier weg.
- Das zu startende Kommando, also das erste Wort einer Kommandozeile, darf nicht nur bekannte Kommandos aufrufen. Vielmehr wird als erstes Wort ein beliebiger Dateiname akzeptiert, auch mit Verzeichnisangaben. Ein eigenes Programm kann man z.B. mit "mein_Unterverzeichnis/hello_world" starten (wenn es das so gibt). Wie die Kommandozeile die Standard-Kommandos findet wäre ein eigenes Kapitel, und wird hier nicht weiter vertieft.

Optionen

Viele Kommandos kennen auch Optionen, mit denen man deren Verhalten modifiziert. Unter Unix sind Optionen Wörter auf der Kommandozeile, die mit einem Minuszeichen anfangen. Beispiel: beim Löschen von Dateien mit dem Kommando rm kann man durch die Option -i erreichen, dass vor dem Löschen jeder einzelnen Datei zurückgefragt wird, ob diese wirklich gelöscht werden soll. Mittels "rm -i *.txt" fragt rm also für jede txt-Datei einzeln, ob diese gelöscht werden soll.

Zwei weitere nützliche Optionen, als Beispiele:
"cp -a Quellverzeichnis nach_da" kopiert "Quellverzeichnis" mit allen Unterverzeichnissen und Dateien darin "nach_da".
"rm -r unterverzeichnis" löscht das Unterverzeichnis mit Inhalt. Ohne -r akzeptiert rm nur Dateien zum Löschen, keine Verzeichnisse.

Viele Programme können sich (sehr kurz) selbst erklären, indem man sie mit der Option -h oder --help aufruft. Längere Erklärungen erhält man meist, indem man das Handbuch mittels des Kommandos "man" (für engl. manual) benutzt. Beispiel: "man cp" liefert eine vollständige Nutzeranleitung, was das Kommando cp alles kann. Irgendwann kommt man auch auf die Idee: was kann eigentlich das man-Kommando alles? "man man" geht auch.

Erste Kommandos

ls
ls, für List, zeigt an, welche Dateien es gibt. Ohne Parameter die, die im aktuellen Verzeichnis liegen. Beispiel:
"ls"
"ls /" mal sehen, was es alles im Wurzelverzeichnis gibt.
Tipp: meist ist auch "ll" verfügbar, dass einfach für "ls -l" steht. Gibt deutlich mehr Informationen pro Datei aus als "ls" ohne "-l".

cp
cp, für copy, dient zum Kopieren von Dateien. Beispiele:
"cp a b" erstellt die Kopie b der Datei a
"cp hi.txt bild.jpg" auch hier entsteht eine Kopie. Wahrscheinlich eine Textdatei mit einem Namen, der eher auf ein Bild hindeutet.
"cp *.txt /sonstwo" - alle txt Dateien in das Verzeichnis /sonstwo kopieren.

mv
Wie cp, aber die Datei wird nicht kopiert, sondern verschoben. Auch gut zum Umbenennen von Dateien.

rm
rm, fuer remove, dient dem Löschen von Dateien:
"rm *" löscht alle Dateien im aktuellen Verzeichnis (Vorsicht beim ausprobieren!)
"rm *.jpg" - löscht alle Dateien, deren Name mit ".jpg" enden

Textdateien erstellen/bearbeiten
Macht man mit seinem bevorzugten Editor, z.B. nano, vi, ... Mit der Kommandozeile startet man aber nur einen Editor, der danach typisch Tastur und Bildschirm komplett übernimmt. Welche Tasten welcher Editor wie belegt muss man leider für jeden Editor einzeln lernen. Wenn Sie noch keinen kennen: nano ist relativ einfach zu erlernen, aber kaum besser als z.B. notepad. Starten würde man ihn z.B. mit "nano meine_datei.txt"

cd
Wechselt das aktuelle Verzeichnis, wie oben schon beschrieben.

mkdir
Legt ein neues Unterverzeichnis an. Beispiele:
"mkdir temp" - legt im aktuellen Vezeichnis ein neues Unterverzeichnis temp an
"mkdir /tmp/mein_temporaeres_verzeichnis" - legt im Verzeichnis "/tmp" ein Unterverzeichnis an. /tmp gibt es für alle Nutzer, wenn man mal kurz eine Datei oder ein Verzeichnis braucht. Nicht erwarten, dass Dateien hier auch nur einen Tag lang überleben!

rmdir
Umkehrung zu mkdir. Löscht aber nur leere Verzeichnisse. Um einen ganzen Dateibaum auf einmal zu löschen nimmt man "rm -r mein_unterverzeichnis", sollte damit aber SEHR VORSICHTIG umgehen.

exit oder logout
Wenn man fertig mit Arbeiten ist.

Wie geht es weiter?

Das Kommando "man" wurde oben schon kurz erwähnt. Es hilft um weitere Optionen für bekannte Kommandos zu finden. Oft findet man da auch Details, die einem erst mal überhaupt nichts sagen. Hier gilt: alles ignorieren, was man gerade nicht versteht, nur das suchen, was man sich erhofft. Dann klappt das ganz gut. Die man-Pages sind kein Handbuch für Anfänger, sondern eine Referenz, was es alles gibt. Auch als Profi schaut man immer wieder dort hinein, und über die Zeit versteht man immer mehr Details. Es gibt als Alternative zu "man" auch noch "info". Geschmacksfrage, welches man bevorzugt.

Wie finde ich ein Programm bzw. Kommando für mein Problem?

Jetzt sind wir beim eigentlichen Problem der Kommandozeile: als Anfänger fehlt einem die Erfahrung, welche Programme es gibt. Es gibt für alles was. Selbst z.B. Bildverarbeitung kann man auf der Kommandozeile betreiben. Praktisch, wenn man 1000 Bilder identisch verarbeiten will.

Was tut man also als Anfänger, wenn man das passende Kommando noch nicht kennt? Jemanden Fragen, der Erfahrung hat! Ist mit Abstand die beste Methode. Ansonsten, als zweitbeste Wahl, nimmt man Google.

Grundlagen der Betriebssysteme - Labor

Versuchsanleitung: Kommandozeilen-Interpreter

In diesem Versuch sollen die Prinzipien von Kommandozeilen erarbeitet werden. Ein einfacher Kommandzeilen-Interpreter ("Shell") wird erstellt, der in der Lage ist, eigene und fertige Programme des Betriebssystems zu starten. Vertiefend werden danach weitere Aspekte von Shells betrachtet und ein Programm erstellt, das für die Nutzung auf der Kommandzeile gedacht ist.

Grundsätzliche Arbeitsweise von Kommandozeilen

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

```
cp -r a b xyz
```

Das Kommando wird von der Shell zuerst in Wörter zerlegt:

```
cp
-r
a
b
xyz
```

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. Per Unix-Konvention verstehen Programme Worte mit führendem Minuszeichen als sogenannte Optionen, während die ohne Minuszeichen Parameter genannt werden, obwohl eigentlich alle Wörter Parameter sind. Die Bezeichnung Option lässt erahnen, dass diese optional sind, also nur angegeben werden, wenn man etwas erreichen will, was nicht Standard ist.

Welche Optionen es gibt und was die Parameter bedeuten legt nur das aufgerufene Programm fest, einfach indem es die Worte entsprechend versteht. Zur Dokumentation gibt heutzutage fast jedes Unix-Programm, das für die Kommandozeile gedacht ist, auf die Option "--help" eine kurze Erklärung von sich selbst. Außerdem gibt es die man-Pages (man für manual, dt. Handbuch), in denen ausführlichere Erklärungen stehen. "man cp" zeigt also eine ausführliche Erklärung des Programms cp an, und "man man" erklärt das Kommando man.

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

Die Parameter a, b und xyz 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 a oder gegebenenfalls das Verzeichnis a mit allen Unterverzeichnissen in das Verzeichnis xyz kopieren. Danach wird mit b genauso verfahren.

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.

Da eine Shell beliebige Programme starten kann, gibt es nicht "die" Kommandosprache für Linux oder Windows. Vielmehr gibt es nur eine Shell und eine Sammlung von Programmen, die mehr oder weniger nützliche Dinge tun können. Gibt es für ein Problem kein passendes Programm, dann kann die Kommandosprache erweitert werden, indem ein neues Programm installiert oder selbst geschrieben wird.

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.

Die Klasse KernelWrapper

Um diese 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 -r /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
```

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

```
nano hello.java
```

Übersetzen und Ausführen geht dann mit:

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

Das ganze ist gegenüber normalem Java so ausgelegt, dass man die Klasse "KernelWrapper" benutzen kann, die vorgegeben ist und die im Folgenden beschrieben wird.

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.

Nun endlich die Methoden selbst. Zuerst mal die, die man für Aufgabe 1 braucht:

- **int fork()**
  Kehrt doppelt, also in zwei Prozessen zurück. Der Eltern-Prozess erhält als Rückgabewert die pid (Prozess-Identifikations-Nummer), der Kind-Prozess eine 0. Andere Unterschiede zwischen den Prozessen gibt es kaum.
- **int execv(String path, String[] argv)**
  Kehrt gar nicht 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 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, die schon hier aufgelistet sind. 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 count bytes in den Puffer buf. Der Returnwert besagt, wieviele Bytes gelesen wurden. Der Rückgabewert -1 besagt, dass ein Fehler aufgetreten ist. read() liefert wirklich nur bytes und ist nicht zeilenorientiert. Das Zeilenende-Zeichen, unter Unix 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)**
  Siehe 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.
- **int close(int fd)**
  Das Gegenstück zum open(), wenn man mit der Verarbeitung einer Datei fertig ist.

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.

Aufgabe 2: Erweitern der eigenen, minimalen Shell

Vom Betreuer wird eine Aufgabenstellung vorgegeben, wie die in Aufgabe 1 selbst erstellte Shell zu erweitern ist.

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"
```

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) 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: |

Erweitern Sie ihre Shell aus Aufgabe 3a) so, dass auch Pipes funktionieren.

Was Pipes tun, sollte aus dem Beispiel oben am Anfang dieser Aufgabe klar sein, wie man umlenkt aus Aufgabe 3a). "pipe()" gibt es als Methode des Linux-Kernels und ist in der Klasse KernelWrapper verfügbar. Wenn man "man pipe" abruft sollten alle Informationen vorhanden sein, um diese Aufgabe 3b) zu lösen.

Aufgabe 4: Erstellen eines eigenen Programms für die Kommandzeile

Vom Betreuer wird eine Aufgabe vergeben, ein Kommandozeilen-Programm selbst zu schreiben, das auf der Linux-eigenen Kommandozeile und mit der eigenen Shell funktionieren muss.

Alle Programme, die vorgegeben werden, müssen Parameter von der Kommandozeile entgegennehmen, also den Parameter args auswerten von "public static void main(String[] args)". Die Parameter werden Optionen und Dateinamen beinhalten.

Ihr Programm muss sicherlich beliebig viele Dateien und (einmalig, aber auch zwischen anderen Dateinamen) "-" für stdin bearbeiten können müssen. Fehlen alle Dateinamen muss Ihr Programm automatisch stdin verarbeiten.

Es ist ebenso sicher, dass eine bzw. mehrere Dateien gelesen werden müssen. Hierbei ist es Pflicht, dass die Methoden open(), read(), readOffset() und close() benutzt werden. Java-eigene Methoden um Dateien zu lesen werden hier im Praktikum 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.

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 werden gleichzeitig abgearbeitet!

Arbeiten mehrere Prozesse am gleichen Programm, dann existieren typisch Bereiche, in denen nur ein Prozess pro Zeiteinheit gleichzeitig arbeiten darf, wenn das Programm korrekt funtktionieren 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**", "**deadlock**" 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 (schon nur einer) 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, 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 bereitsteht. Dieses Framework kopiert man sich komplett in ein eigenes Verzeichnis, z.B. mittels "cp -a /import/grpdrvs/ti_prak/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. Es gibt zwei Routinen, 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:

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, dass 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: Ein weiteres Problem zu Synchronisieren

Hier ist eine andere Aufgabe zu lösen. Der Betreuer gibt die Aufgabe vor.

Alle Aufgabenstellungen liegen dem Framework bereits bei. Der Quelltext in der Datei beinhaltet jeweils auch die Aufgabenstellung.

Aufgabe 4: Schreiben Sie selbst ein Programm

Auch hier gibt der Betreuer eine Aufgabe vor. Jedoch sollen Sie hier nicht einfach nur ein paar Semaphore geschickt einfügen, sondern selbst einige Codezeilen schreiben, inklusive den geeigneten Semaphor-Operationen. Nehmen Sie als Start eine Kopie eines Quelltextes, den Sie im Versuch vorher schon bearbeitet haben.

Wichtig ist es hier, 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. Dies gilt also diesmal nicht nur für die Semaphor-Operationen!

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

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, die jeweils auf der Datei "/tmp/filesystem_von_" als FAT32-Filesystem operieren. (bitte durch den eigenen Accountnamen ersetzen.)

- **ti_makefs**
  stellt immer exakt den Anfangszustand her, von dem aus jede Teilaufgabe bearbeitet werden kann. "ti_makefs" kann 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 "/tmp/filesystem_von_" 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**
  mountet die Datei "/tmp/filesystem_von_" 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.

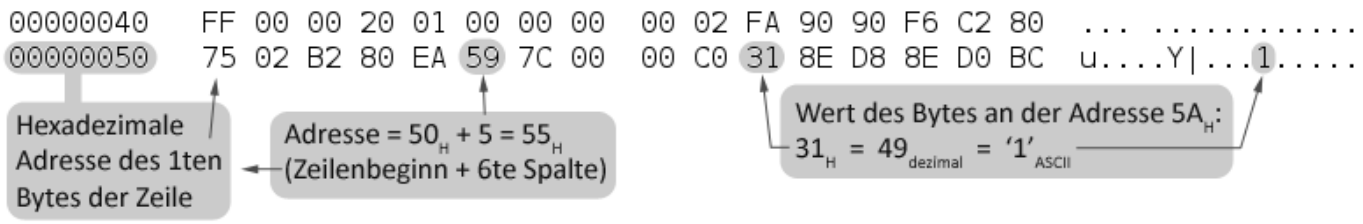
Hex-Editor-Einführung

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". Zudem teilt Ihnen Ihr Praktikumsbetreuer eine weitere Aufgabe zu:

_____ Sie den _____ der Datei _____.

Aufgabe 2: Modifizieren des Dateiinhalts

Fragen Sie wieder Ihren Betreuer nach einer individuellen Aufgabenstellung:

_____ Sie die Datei "_____" im Wurzelverzeichnis _____.

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

Aufgabe 3: Problemlösung und Abschlusskolloquium

In dieser letzten Teilaufgabe teilt Ihnen Ihr Betreuer zuerst wieder eine indiviuelle Problemstellung zu. Zum Abschlusskolloquium sollten Sie dieses 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`)	
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␣`.	
0x0B	2	Bytes per sector; the most common value is 512. The BIOS Parameter Block starts here.	
0x0D	1	Sectors per cluster. Allowed values are powers of two from 1 to 128.	
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.	
0x10	1	Number of file allocation tables. Almost always 2; RAM disks might use 1.	
0x11	2	Maximum number of FAT12 or FAT16 root directory entries. **0** for FAT32, where the root directory is stored in ordinary data clusters.	
0x13	2	Total sectors (if zero, use 4 byte value at offset 0x20)	
0x15	1	Media Descriptor Byte [gekürzt]	
0x16	2	Sectors per File Allocation Table for FAT12/FAT16, 0 for FAT32 (cf. offset 0x24 below)	
0x18	2	Sectors per track for disks with geometry, e.g., 18 for a 1.44MB floppy	
0x1A	2	Number of heads for disks with geometry, e.g., 2 for a double sided floppy	
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.	
0x20	4	Total sectors (if greater than 65535; otherwise, see offset 0x13)	

Further structure used by FAT32:

Byte Offset	Length (bytes)	Description	[wert]
0x24	4	Sectors per file allocation table	
0x28	2	FAT Flags (Only used during a conversion from a FAT12/16 volume.)	
0x2A	2	Version (Defined as 0)	
0x2C	4	Cluster number of root directory start	
0x30	2	Sector number of FS Information Sector	
0x32	2	Sector number of a copy of this boot sector (0 if no backup copy exists)	
0x34	12	Reserved	
0x40	1	Physical Drive Number	
0x41	1	Reserved	
0x42	1	Extended boot signature	
0x43	4	ID (serial number)	
0x47	11	Volume Label [ungenutzt]	
0x52	8	FAT file system type: "FAT32 "	
0x5A	420	Operating system boot code	
0x1FE	2	Boot sector signature (hex. `55AA`)	

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>
<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]
<table>
<tr><th>Bit</th><th>Mask</th><th>Description</th></tr>
<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:
<table>
<tr><th>Bits</th><th>Description</th></tr>
<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:
<table>
<tr><th>Bits</th><th>Description</th></tr>
<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.