Lowlevel, Ausgabe 4

Mastermesh (*Leitender Redakteur*)
TeeJay (*Redakteur*)
Phil (*Kolumnist*)

Datum: 27. 12. 2003
Version: 1.0

Inhalt

1.Vorwort

So, da wären wir mal wieder mit einer neuen Ausgabe von Lowlevel.

Als allererstes muss ich mich ersmal entschuldigen. Ich habe mir für diese Ausgabe sehr, sehr viel Zeit gelassen. Fast 3 Monate. Aber nicht ohne Grund... Stress in der Schule und im sonstigen Leben machen es schwer, mich auf Lowlevel zu konzentrieren...

Der angekündigte Artikel über Kernel-Entwicklung in C ist leider nicht in der Form erschienen, in der er sein sollte. Aus Zeitmangel habe ich daher Tim Robinson's englischen Artikel über C/C++ abgedruckt. Ich weiß... eines der Hauptziele dieses Magazins ist es, Informationen in deutscher Sprache zu liefern, aber ich konnte echt nicht anders... tut mir leid.

Übrigens gibt es noch eine Neuerung am Konzept. Die einzelnen Rubriken wie "OS Dev-Tutorial" oder "Lowlevel-Grundlagen" sind ab dieser Ausgabe nicht mehr vorhanden, die Artikel werden einfach ohne Angabe der Rubrik abgedruckt. Wir haben uns dafür entschieden, weil es einfach Artikel gab, die in keine von diesen Kategorien gepasst haben.

Trotzdem, viel Spaß beim Lesen.

2.Die Redaktion

Auf vielfachen Wunsch folgt hier eine Vorstellung unserer chaotischen Redaktion.

Der Chefredakteur

Jo, das bin ich. Ich bin für dieses ganze Geschribsel verantwortlich. Das bedeutet, dass ich den Großteil der Artikel schreibe (obwohl jetzt immer mehr Beiträge von Lesern kommen!). Gleichzeitig bin ich auch der Webmaster, ich schreibe also die PHP-Scripte und bin dafür verantwortlich, dass die Seite gepflegt wird.

Über mich persönlich habe ich schon in der ersten Ausgabe geschrieben. Nun möchte ich euch etwas anderes erzählen, und zwar was ich noch abseits von OS-Development mache.

Ich bin kein Programmier-Guru. Und ich werde auch nie einer sein. Stattdessen habe ich tatsächlich noch ein Leben außerhalb des PCs. Ich spiele gerne Geige und interessiere mich auch sonst viel für Musik. MIDI-Musik ist ein Hobby von mir, es ist einfach geil, was man allein mit einem MIDI-Sequenzer hinkriegt!

Noch ein Hobby von mir ist das DXen. DX steht für 'Distance X' und bezeichnet das Empfangen von Kanälen über Satellit von der ganzen Welt. In meiner Kanalliste gibt es ein paar Exoten wie z.B Kuba, Äthiopien, Chile, Dubai...

Leute, hängt nicht so viel vor'm PC rum! Das echte Leben ist besser :)

Wer zum Teufel ist phil?

Da der Mail-Eingang von mir vor solchen Fragen fast platzte, steht hier alles, was ich über Phil weiß.

Phil ist ein 1A Coder. Vor ein paar Jahren hatte er mal ein tolles Projekt, das Betriebssystem "Radiation" für Nokia's uralte d-Box. Das Betriebssystem galt unter DXern und in sonstigen Insiderkreisen als das beste Betriebssystem für die d-Box. Es ist unglaublich, welche Leistung Phil aus einem 16 MHz-Prozessor rausgequetscht hat. Durch "Radiation" bin ich überhaupt erst mit Phil zusammengekommen.

Hier bei Lowlevel schreibt Phil uns in seiner Kolumne über die Probleme der Welt und hilft mir zwischendurch ein wenig, Artikel zu schreiben.

TeeJay

Angefangen hat alles mit ein paar Leserbriefen. Nun ist TeeJay (fast) fester Bestandteil der Redaktion. Hier ein paar Zeilen, die er über sich selbst geschrieben hat:

Bin 21 Jahre alt. Hab bis jetzt mein Fachabitur gemacht. Dann war ich bei der Bundeswehr. Und ab Montag (MM: ist schon ein paar Wochen her) geh ich Studieren.

Zum Programmieren kam ich in der 10ten Klasse, als es ein Wahlpflichtfach Informatik gab. Da haben wir dann QuickBasic gemacht. Danach bin ich zu VB umgestiegen. In der Fachoberschule kam dann noch Java und Delphi dran. Java hab ich dann als guten Einstieg für C/C++ genommen, was bis jetzt auch mein Favorit ist.

Zu OS Dev kam ich eigentlich erst richtig über das Magazin. Über Assembler und so Zeugs hatte ich vorher zwar schon vieles gelesen gehabt, hatte aber nie wirklich damit angefangen. Durch das Magazin hatte ich mal wieder richtig Lust bekommen und hab mich gleich hingesetzt und was dazu gelesen.

Newsredakteur gesucht!

Wir suchen dringend noch einen Newsreporter. Wie ihr in den letzten paar Ausgaben gesehen habt, ist die News-Sektion doch sehr spärlich, was vor allem daran liegt, dass ich leider keine Flatrate habe und weil ich es mir deshalb nicht leisten kann, stundenlang die Seiten von OS-Projekten abzuklappern. Wenn jemand also Zeit/Lust hat, ein wenig zu recherchieren, soll er sich bei uns melden!

3.Das FAT12-Dateisystem

Vorweg folgendes:
Ich gehe in diesem Text davon aus, dass man gewisse Programmierkenntnisse hat und mit der Erwähnung von BITs und Bytes etwas anfangen kann.

Wenn ich hier von Sektornummern rede, dann meine ich damit die logische Sektornummer (LSN). Diese beginnt bei '0' und endet bei 'Anzahl aller Sektoren - 1'.

Normalerweise adressiert man einen Sektor über das 'Cylinder Head Sektore' (CHS) Format. Hier gehe ich jedoch davon aus, dass ein Programm vorhanden ist, dass die LSN in das CHS-Format übersetzen kann.

In diesem Text beschreibe ich den Aufbau des FAT12-Dateisystems. Theoretisch könnte man jeden Datenträger bis zu einer bestimmten Speicherkapazität mit FAT12 formatieren. Da jedoch die Diskette der gängigste Datenträger ist, auf dem FAT12 verwendet wird, beziehe ich mich in diesem Text ausschließlich auf die Diskette.

Das FAT12-Dateisystem besteht aus 4 Bereichen, die auf der Diskette gespeichert werden:

- Boot Sector (oder auch BIOS Parameter Block)
- FAT Tabelle
- Root Directory
- Datei- und Verzeichnis-Region

Boot Sector (BIOS Parameter Block)

Dieser Sektor enthält die wichtigsten Informationen, die wir benötigen, um mit dem FAT-Dateisystem korrekt arbeiten zu können. Wie auch ein regulärer Boot Sector, befindet sich dieser im ersten Sektor der Diskette. Dieser Sektor ist laut LSN der Sektor 0. Im Unterschied zu einem herkömmlichen Boot Sector ist jedoch dieser NICHT dazu gedacht, ein Betriebssystem zu booten!

Hier folgt nun der Aufbau des Boot Sectors. Dieser wird in einer Tabelle dargestellt, wobei folgende Spalten vorhanden sind:

- Bezeichnung (ein kurzer Name, der den Eintrag kennzeichnet)
- Offset (Startadresse in Bytes)
- Bytelänge (Anzahl der Bytes, die dieser Eintrag verwendet)
- Beschreibung (eine Beschreibung, wozu der betreffende Eintrag dient)

Bezeichnung	Offset	Bytes	Bezeichnung
JmpBoot	0	3	Hier steht ein JMP-Befehl. Wozu dieser Jump gut ist, wird später klar.
OSName	3	8	Hier steht ein 8-Buchstaben-String, der angibt, unter welchen Betriebssystem die Diskette formatiert wurde. Unter Windows sollte dort "MSWIN4.1" stehen. Man kann hier aber eigentlich schreiben, was man möchte. Es hat auf das Dateisystem keinen Einfluss.
BytesPerSec	11	2	Die Anzahl Bytes, die in einem Sektor vorhanden sind. Bei einer Diskette entspricht dies 512. Dieser Wert ist fest und kann ohne weiteres auch nicht geändert werden und sollte daher auch immer hier eingetragen werden.
SecPerClus	13	1	Die Anzahl Sektoren, die zu einem Cluster zusammengefasst werden. Was ein Cluster ist, erkläre ich weiter unten. Hier sei vorerst nur wichtig, das auf einer Diskette exakt EIN Sektor in einem Cluster ist. Also hier den Wert 1 eintragen.
RsvdSecCnt	14	2	Die Anzahl der Reservierten Sektoren. Der Boot Sector, in dem wir uns gerade befinden, ist ein reservierter Sektor. Da man in der Regel nur diesen reservierten Sektor benötigt, trägt man hier 1 ein.
NumFATs	16	1	Die Anzahl der FAT-Tabellen. Aus Redundanzgründen speichert man meistens 2 FAT-Tabellen auf der Diskette, im Falle, dass eine davon beschädigt sein sollte. Daher hier den Wert 2 eintragen.
RootEntCnt	17	2	Die Anzahl der möglichen Einträge ins Root-Directory. Wozu dieser Wert genau dient, erkläre ich im Abschnitt über das Root-Directory. Der Standartwert hier ist 224.
TotSec	19	2	Die Anzahl aller Sektoren, die auf der Diskette vorhanden sind. Bei einer 3,5" 1,44 MB-Diskette sind das 2880 Sektoren. 1,44 MB sind 1440 KB (die Hersteller multiplizieren immer mit 1000 und nicht

			1024). 1440 KB sind 1474560 Bytes (diesmal mal 1024). 147560 Bytes geteilt durch 512 Bytes (1 Sektor) ergibt 2880 Sektoren.
MediaType	21	1	Hier wird der Wert eingetragen, anhand dem erkannt wird, um was für einen Typ es sich bei diesem Datenträger handelt. 0xF0 ist der Wert für Diskette.
FATSize	22	2	Hier steht die Anzahl der Sektoren, die eine FAT-Tabelle belegt (man erinnere, es gibt meistens ZWEI). Der Wert bei einer 1,44 MB-Diskette ist 9. Wie man auf den Wert kommt, erläutere ich im Abschnitt über die FAT-Tabelle.
SecPerTrack	24	2	Die Anzahl der Sektoren pro Track. Track wird auch als Cylinder bezeichnet. Dieser Wert ist für den Interrupt 13h wichtig, da dieser die Sektoren über CHS adressiert. Der Wert für eine 1,44 MB Diskette ist *** (N/A).
NumHeads	26	2	Die Anzahl der Köpfe. Auch diese Angabe wird vom Interrupt 13h benötigt, um die Sektoren nach CHS adressieren zu können. Bei einer 1,44 MB-Diskette ist der Wert 2.
HiddenSec	28	4	Die Anzahl der versteckten Sektoren. Wozu es nützlich wäre, versteckte Sektoren zu haben, weiß ich im Moment nicht, weshalb man hier den Wert 0 eintragen sollte.
TotSec32	32	4	Dieser Eintrag ist nur für einen FAT32-formatierten Datenträger wichtig. Da wir uns aber auf FAT12 konzentrieren, tragen wir hier 0 ein.
DrvNum	36	1	Dieser Wert wird ebenfalls von Interrupt 13h benötigt und gibt Auskunft darüber um welches Laufwerk es sich handelt. Für Diskette tragen wir hier 0x00 ein.
Reserved	37	1	Dieser Eintrag ist für Windows NT reserviert und sollte mit dem Wert 0 beschrieben werden (auch wenn man WindowsNT/2000/XP benutzt).
VolumeID	39	4	Hier kann man eine Seriennummer angeben anhand deren man die Diskette beim Einlesen erkennen kann. Dieser Wert wird meistens aus der Zusammenfassung der beiden 16-BIT-Werte von Datum und Zeit gemacht. Wie man Datum und Zeit in 16 BIT speichert, erkläre ich weiter unten.
FileSysType	54	8	Dämlicherweise soll man hier angeben, welches FAT-Dateisystem man auf dem Datenträger verwendet. Dämlicherweise deshalb, weil dieser Eintrag nicht dazu verwendet wird, um festzustellen welches FAT-System man nun benutzt hat. Aus Gründen der Kompatibilität sollte man hier einfach "FAT12" eintragen und die restlichen Bytes mit 0x20 (Leerzeichen) füllen. Wie man exakt bestimmt, welches FAT-System denn nun auf dem Datenträger verwendet wird erläutere ich weiter unten.

Aus welchen Gründen auch immer man es so gewählt hat, muss man auch hier, wie in einem gewöhnlichem Boot Sektor, die Bytes 511 und 512 mit dem Wert 0xAA55 beschreiben. Da das BIOS diesen Boot Sektor nun auch als einen solchen erkennt, wird auch ersichtlich, wozu der JMP-Befehl in den ersten 3 Bytes dient. Damit springt man an des Ende der oben beschriebenen Tabelle und kann dort dann Code eintragen, der ausgeführt werden soll.

Im Regelfall lässt man dort eine Meldung ausgeben, dass es sich hier nicht um eine bootfähige Diskette handelt. Dann wartet man auf einen Tastendruck, um dann neu zu booten. Windows verfährt so, wenn es eine Diskette formatiert.

Formatieren heißt eigentlich nichts anderes, als diesen Boot Sektor neu zu schreiben

und die Einträge der FAT-Tabelle zu löschen. Dies wäre dann das QuickFormat. Eine komplette Formatierung löscht zusätzlich noch alle Sektoren der Diskette. Oder besser gesagt, man überschreibt den Inhalt der Sektoren mit lauter Nullen.

Hier jetzt ein paar Beschreibungen zu Dingen die offen geblieben sind.

Cluster

Die FAT-Tabelle verwaltet nicht die einzelnen Sektoren eines Datenträgers, sondern sogenannte Cluster. Bei einer Diskette ist dies aber irrelevant, da ein Cluster nur einen Sektor beinhaltet. Man könnte jedoch mehrer Sektoren zu einem Cluster zusammenfassen. Das Ganze hat folgenden Hintergrund:

FAT12 benutzt 12 BITs, um die Cluster durchzunummerieren, um jeden Einzelnen über eine Nummer ansprechen zu können. Nun kann man aber mit 12 BITs lediglich 4096 verschiedene Nummern darstellen. Unsere Diskette hat "glücklicherweise" nur 2880 Sektoren, weshalb diese 12 BITs vollkommen ausreichen, um alle Sektoren adressieren zu können. Hätte die Diskette jedoch mehr als 4096 Sektoren, dann hätten wir ein Problem. Wir könnten dann nicht mehr alle Sektoren ansprechen und würden somit Speicherplatz auf der Diskette ungenutzt lassen. Da man jedoch mehrere Sektoren zu einem Cluster zusammenfassen kann, könnten wir somit auch mehr als 4096 Sektoren adressieren. Würden wir 2 Sektoren zu einem Cluster zusammenfassen, dann könnten wir 8192 Sektoren auf der Diskette verwenden.

Wichtig: Ein Cluster ist die kleinste Einheit, die ich mit dem FAT-System ansprechen kann. Und eine Datei, die ich auf der Diskette speichere, benötigt daher mindestens einen ganzen Sektor. Selbst eine 32 Byte große Datei benötigt einen kompletten Sektor (512 Bytes). Die restlichen Bytes, die die Datei nicht belegt, sind somit verloren. Würden wir nun 2 Sektoren zu einem Cluster zusammenfassen, so hätte ein Cluster 1024 Byte. Würden wir nun wieder eine 32 Byte kleine Datei in einem Cluster ablegen, so würde wesentlich mehr Bytes verloren gehen. Daher wählt man die Anzahl Sektoren möglichst so klein, das man damit alle Sektoren ansprechen kann. Und da wir mit 12 BITs alle 2880 Sektoren der Diskette ansprechen kann, fassen wir lediglich einen Sektor zu einem Cluster zusammen.

Speicherweise von Datum und Zeit in 16 BITs

Da man früher noch sehr auf Speicherplatz achten musste, hat man möglichst alle Daten so komprimiert wie möglich zusammengefasst. So hat man es auch gemacht, dass man die Angabe von Stunde, Minute und Sekunde in lediglich 16 BIT unterbringen konnte. Die 16 BITs sind wie folgt aufgebaut:

Listing 1

```
Bit 0 - 4 (5 Bits) = 2er Sekunden; Werte : 0 - 29
```

Da 5 Bits nur 32 verschiedene Zahlen darstellen können hat man es so gemacht, das man nur die Wert 0 bis 29 eintragen kann und dieser Wert dann bei der Ausgabe mit 2 multipliziert werden muss.

Listing 2

```
Bit 5 - 10 (6 Bits) = Minuten; Werte : 0 - 59
Bit 11 - 15 (5 Bits) = Stunden; Werte : 0 - 23
```

Man sieht hier schon, dass man durch das begrenzen auf nur 16 BITs gewisse Einbußen hinnehmen muss. Siehe Sekunden. Im Datum werden Tag, Monat und Jahr wie folgt dargestellt:

Listing 3

```
Bit 0 - 4 (5 Bits) = Tag des Monats; Werte : 0 - 31
Bit 5 - 8 (4 Bits) = Monat; Werte : 1 - 12
Bit 9 - 15 (7 Bits) = Jahr; Werte : 0 - 127
```

Zu dem Wert des Jahrs muss noch 1980 hinzugezählt werden. Hier sehen wir wieder die Nachteile einer harten Speicherkompression. Das FAT12-Dateisystem wird spätestens 2107 seinen Geist aufgeben müssen, da wir nicht mehr Jahre darstellen können *g*.

Feststellen, welches FAT-Dateisystem benutzt wird

Ich beziehe mich in diesem Text auf das FAT12-System. Wie die Meisten jedoch wissen werden, gibt es neben dem FAT12 auch noch das FAT16 und FAT32. Und daher möchte ich hier noch kurz erklären, wie man feststellen kann, welches FAT-System den nun auf einem bestimmtem Datenträger verwendet wird. Dies erkennt man schlichtweg daran, wie viele Cluster in der FAT-Tabelle verwaltet werden. Und nur daran.

Da man ja mehrere Sektoren zu einem Cluster zusammenfassen kann, müssen wir erst mal errechnen, wie viele Sektoren auf der Diskette noch beschrieben werden können. Das heißt, abzüglich der Sektoren, die durch die FAT-Tabelle, BootSektor usw. belegt sind.

Listing 4

```
Sektoren = TotSec - RsvdSecCnt - (NumFATs * FATSize) - RootDirSectors
```

Wie man die Anzahl der RootDirSectors ermittelt, wird im Abschnitt des Root-Directorys, welcher im Anschluss folgt, gezeigt.

Nun müssen wir noch errechnen, wie viele Cluster das nun sind. Dazu teilt man die Anzahl der Sektoren durch die Anzahl der Sektoren die, zu einem Cluster zusammengefasst sind:

Listing 5

```
Cluster = Sektoren / SecPerClus
```

Nun müssen wir in einer Tabelle nachschauen, aus der wir ablesen können, welches FAT-Dateisystem verwendet wird:

Listing 6

```
Cluster < 4085 -> FAT 12
Cluster < 65525 -> FAT 16
Cluster > 65525 -> FAT 32
```

Root-Directory

Das Root-Directory ist im FAT12-Dateisystem eine kleine Besonderheit. Die Einträge in das Root-Directory sehen zwar genauso aus, wie alle anderen Einträge in Unterverzeichnisse, jedoch ist die maximale Anzahl der Einträge hier begrenzt. Die maximale Anzahl an Einträgen legt man in RootEntCnt fest.

Da ein Eintrag immer 32 Bytes belegt, sollte hier ein Vielfaches von 16 gewählt werden, damit wir immer einen ganzen Sektor belegen. (32 * 16 = 512). Der Standartwert ist hier 224, welches auch genügen sollte. Ich persönlich habe noch keine Diskette gesehen die 224 Einträge im Root-Directory hat.

Wichtig: Die Begrenzung der Einträge im Root-Directory hat keine auf die Anzahl der Einträge in den Unterverzeichnisse! Die Einträge in Unterverzeichnis sehen genauso aus wie die im Root-Directory, werden jedoch nach einem leicht anderem Schema gespeichert.

Ein Eintrag beschreibt den Namen und die Eigenschaften von Dateien oder Verzeichnissen. Der Eintrag einer Datei gleicht also dem eines Verzeichnisse bis auf ein Attribut, das bei einem Verzeichnis gesetzt wird.

Hier eine Erläuterung wie ein solcher Eintrag aufgebaut ist:

- Bezeichnung (ein kurzer Name, der den Eintrag kennzeichnet)
- Offset (Startadresse in Bytes)
- Bytelänge (Anzahl der Bytes, die dieser Eintrag verwendet)
- Beschreibung (eine Beschreibung, wozu der betreffende Eintrag dient)

Bezeichnung	Offset	Bytes	Bezeichnung
DIR_Name	0	11	Der Name der Datei / des Verzeichnisses. Hier gilt die alte 8+3 Konvention. 8 Zeichnen für den Namen; 3 Zeichen für die Extension. Die beiden werden aber ohne Punkt hintereinandergeschrieben. Zeichen die bei den 8 oder 3 Zeichen NICHT belegt sind, müssen mit Leezeichen (0x20) aufgefüllt werden.
DIR_Attribute	11	1	Hier können einzelne BITs oder Kombinationen aus BITs gesetzt werden, um Attribute festzulegen. ATTR_READ_ONLY: 0x01 ATTR_HIDDEN: 0x02 ATTR_SYSTEM: 0x04 ATTR_VOLUME_ID: 0x08 ATTR_DIRECTORY: 0x10 ATTR_ARCHIVE: 0x20 Ein Verzeichnis unterscheidet sich also nur von einer Datei, indem das ATTR_DIRECTORY BIT gesetzt ist.
DIR_Name	0	11	Der Name der Datei / des Verzeichnisses. Hier gilt die alte 8+3 Konvention. 8 Zeichnen für den Namen; 3 Zeichen für die Extension. Die beiden werden aber OHNE Punkt hintereinandergeschrieben. Zeichen die bei den 8 oder 3 Zeichen nicht belegt sind, müssen mit Leezeichen (0x20) aufgefüllt werden.
DIR_NTRes	12	1	Dieses Byte ist für Windows NT reserviert und sollte immer mit 0 initalisiert werden.
DIR_CrtTimeTenth	13	1	Hier werden die Millisekunden der Zeit gespeichert, an dem der Eintrag erstellt wurde. Der Wert, der hier eingetragen wird, muss man allerdings noch mit 10 multiplizieren, um den korrekten Wert der Millisekunden zu erhalten. Zulässige Werte sind 0 - 199. Also so gesehen von 0 bis 1990 Millisekunden.
DIR_LstAccDate	16	2	Datum der Erstellung der Datei / des Verzeichnisses. Siehe oben für Speicherweise von Datum in 16 BITs.

DIR_FstClusH	20	2	Dieser Eintrag ist für FAT12 nicht relevant und sollte immer 0 sein.
DIR_WrtTime	22	2	Zeit, an der die Datei das letzte Mal geschrieben wurde. Also wann sie das letzte Mal verändert wurde.
DIR_WrtDate	24	2	Datum, an dem die Datei das letzte Mal verändert wurde.
DIR_FstClusLO	26	2	Die Nummer des Startsektors der Datei / des Verzeichnisses.
DIR_FileSize	28	4	Dateigröße in Bytes.

Bei dem Feld DIR_Name gibt es ein paar Besonderheiten, die man beachten sollte.

1. Ist das erste Zeichen des Namens ein 0x0E, dann ist dieser Eintrag noch frei und kann benutzt werden.

2. Ist das erste Zeichen des Namens ein 0x00, dann ist dieser und die folgenden Einträge frei und können genutzt werde. Dabei kann man sich dann im Falle des Durchsuchens des Verzeichnisses sparen, die nächsten Einträge ebenfalls noch durchzuschauen.

3. Wie oben schon erwähnt, wird hier der 8+3 Dateiname gespeichert. Der Punkt wird jedoch nicht mitgespeichert, sondern wird von dem Programm, das die Einträge auflistet, bei der Ausgabe selbst mit eingefügt.

Und auch müssen die unbenutzten Zwischenräume mit Leerzeichen aufgefüllt werden. Beispiel:

Listing 7

```
"FOO     BAR"    -> FOO.BAR
"FOO     B  "    -> FOO.B
"PRETTYBIG"      -> PRETTY.BIG
```

Alle Einträge müssen in Großbuchstaben sein. Daher gibt es auch keine Unterscheidung zwischen Groß- und Kleinschreibung. Es kann also nicht die beiden Dateien Foo.bar und FOO.bar geben!

Folgende Zeichen (HEXCODES) dürfen NICHT im Dateinamen verwendet werden:

- Alle Werte unter 0x20

- 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C , 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D und 0x7C

Das Root-Directory wird in den Sectoren direkt hinter den beiden FAT-Tabellen gespeichert. Die Einträge sind nacheinander in den Sektoren gespeichert. Anhand der Anzahl Einträge, die wir im Bootsektor mit 224 festlegen, können wir auch errechnen, dass wir exakt 14 Sektoren benötigen, um alle Einträge zu speichern.

Listing 8

```
224 Einträge * 32 Bytes = 7168.
7168 / 512 Bytes (Sektor) = 14 Sektoren.
```

Wenn wir nun listen wollen, welche Dateien und Verzeichnisse sich im Root-Directory befinden, müssen wir diese 14 Sektoren einlesen und schauen, welche der Einträge belegt sind und diese dann formatiert ausgeben.

Wenn wir nun in ein Unterverzeichnis wechseln wollen, schauen wir in dem Eintrag des Verzeichnisses nach, in welchem Sektor dieses liegt. Dann lesen wir diesen Sektor ein und finden dort dann wieder die 32 Byte Einträge über weitere Dateien und Verzeichnisse, die sich in diesem Unterverzeichnis befinden.

Wenn wir allerdings eine Datei ansprechen wollen, dann müssen wir ebenfalls im Eintrag nachschauen, an welchem Sektor die Datei steht und können diesen dann einlesen. Jedoch stehen in diese Sektor die reinen Daten der Datei. Also keine Directoryeinträge!

Jedoch kann es sein, dass eine Datei mehr als einen Sektor belegt. Und zu unseren Ungunsten ist es auch oft so, dass die Sektoren, die eine Datei belegt, nicht alle hintereinander stehen.

Das heißt, wir müssen mehrere Sektoren zusammensuchen, um die komplette Datei lesen zu können. Dies bewerkstelligen wir mit der eigentlichen FAT-Tabelle, welche ich im nächsten Abschnitt erläutern werde.

FAT-Tabelle

So, nun endlich zum Herzstück des FAT12-Dateisystems. Die FAT-Tabelle ist eigentlich nichts anderes als eine Tabelle, die 12 BIT-Zahlen beinhaltet. Hier sehen wir nun auch, warum es FAT12 heißt.

Mit 12 BITs kann man 4096 verschiedene Zahlen darstellen. Da eine 1,44 MB Diskette 2880 Cluster (Ein Cluster = Ein Sektor) hat, benötigen wir also mindestens 12 BITs, um alle Cluster adressieren zu können. 12 BITs sind nun etwas unhandlich, da man normal ja nur mit vielfachen von 8 BITs (1 Byte) arbeitet.

Da man aber Platz sparen wollte, hat man sich dafür entschieden, lediglich 12 BITs zu benutzen, um die Cluster zu adressieren, obwohl man mit 16 BITs hätte ebenfalls alle adressieren können. Zumal man mit 16 BITs wesentlich einfacher arbeiten kann! Wer schon mal programmiert hat (und damit meine ich nicht so was wie VB *G*), der weiß, warum man mit 16 BITs einfacher arbeiten kann als mit 12 BITs.

Man könnte nun meinen, dass man die 12 BITs einfach verteilt in 2 Bytes (16 BITs) speichert, um dann einfach 4 BITs in einem der Bytes unberücksichtigt zu lassen. Das wäre eine schöne Sache, würde jedoch die Speichereinsparung, die wir mit den 12 BITs erzielen wollen, wieder zunichte machen. Also speichert man die Einträge der FAT-Tabelle immer so, dass man zwei 12 BIT Einträge auf 3 Byts (24 BITs) aufteilt.

Also müssen sich 2 Einträge 1 Byte teilen. Beispiel:

Listing 9

```
Eintrag 1 = 1111.1111.1111
Eintrag 2 = 2222.2222.2222
```

Gespeichert in 3 Bytes würde das so aussehen:

Listing 10

```
Byte 0     Byte  1     Byte 2
1111.1111  1111.2222   2222.2222
```

Man kann also schon sehen, das man hier mit BIT-Operationen arbeiten muss, um die einzelnen Einträge zu extrahieren.

Die Einträge selbst sind einfach fortlaufend hintereinander gespeichert. Bei einer 1,44 MB Diskette hat die FAT-Tabelle insgesamt etwas weniger als 2880 Einträge. Etwas weniger daher, weil die Cluster, die wir benutzen müssen, um den Bootsektor, FAT-Tabelle und Root-Directory zu speichern, dort nicht mit aufgelistet werden, weil

sie sogesehen nicht verfügbar sind.

Und um noch mehr Verwirrung zu stiften, nummeriert man die Einträge der FAT-Tabelle wieder bei 0 beginnend. Das heißt, wir ziehen von den 2880 Clustern, die eine Diskette insgesamt hat, die Cluster ab, die wir bereits mit dem Bootsektor, FAT-Tabelle und Root-Directory haben. Und die Cluster, die übrig bleiben, werden somit wieder bei 0 anfangend angesprochen. So steht der erste Eintrag in der FAT-Tabelle für den Cluster 0. Der zweite Eintrag für den Cluster 1 usw.

Hier darf man aber nicht verwechseln, dass der Cluster, den die FAT-Tabelle als Cluster 0 bezeichnet, nicht der Sektor 0 ist, den wir mittels LSN ansprechen. Also NICHT der erste Sektor der Diskette ist.

Hinzu kommt, dass die ersten beiden Cluster, die von der FAT-Tabelle angesprochen werden, nicht benutzt werden dürfen, da diese unter MS-DOS 1 und 2 mit bestimmten Werten gefüllt waren. Und man aufgrund der Abwärtskompatibilität auf diese beiden Cluster verzichtet. Man fängt dann also erst bei Cluster 2 an Daten zu schreiben.

So nun noch zum wichtigsten. Wir haben ja schon gesagt das jeder Eintrag 12 BITs hat. Nun bleibt noch die Frage, was genau wir denn in die 12 BITs schreiben. Und genau das ist der Kniff, um Dateien, die verstreut auf der Diskette liegen, wieder zusammenfügen zu können. In dem Eintrag steht die Nummer des Cluster, der die jeweils nächsten Daten der Datei beinhalten.

Ein kleines Beispiel soll das verdeutlichen. Angenommen, wir sind im Root-Directory und haben dort eine Datei "Test.dat" gelistet. Dann schauen wir in dem Directoryeintrag nach, an welchem Cluster (DIR_FstClusLO) diese Datei beginnt. Wir nehmen mal an, es wäre der Cluster 2. Dann suchen wir uns nun den 3ten Eintrag in der FAT-Tabelle raus. (Man erinnere sich, Cluster 2 ist der dritte Cluster, da wir bei 0 anfangen zu zählen). Nun schauen wir, was in dem Eintrag der FAT-Tabelle steht. Und hier müssen wir zwischen folgenden Werten unterscheiden:

0x0FFF -> Dieser Wert sagt uns, dass der Cluster der letzte ist, der Daten über diese Datei enthält! Das heißt, wir müssen nur diesen Sektor einlesen und haben somit die komplette Datei.

0x0FF7 -> Dieser Wert gibt uns an, dass es sich um einen bad cluster handelt. Hier sollte man also keine Daten ablegen, da der Datenträger an dieser Stelle möglicherweise defekt ist.

0x0000 -> Dieser Wert signalisiert, das der Cluster noch frei ist.

Jeder andere Wert niedriger als 0x0FFF ist die Nummer des Cluster, der die nächsten Daten der Datei enthält.

Wir nehmen also mal an, dass in dem Eintrag der FAT-Tabelle von Cluster 2 die Zahl 0x003 steht. Daran würden wir erkennen, das wir auch noch den Cluster 3 einlesen müssen. Also schauen wir nun auch noch in der FAT-Tabelle nach dem Eintrag von Cluster 3.

Dort möge nun der Wert 0x010 stehen. Das würde heißen, dass die Datei auch noch einen dritten Cluster, und zwar den Cluster 16 (0x10 = 16 dez), belegt. Also lesen wir auch diesen ein und schauen wieder in der FAT-Tabelle nach, was in dem Eintrag von Cluster 16 steht. Dort steht dann nun der Wert 0x0FFF, was uns signalisiert, das dies nun der letzte Cluster ist, der Daten der Datei enthält! Das heißt, wir müssen diesen Cluster 16 also auch noch einlesen und können dann aufhören, da wir die Datei komplett haben.

In diesem Beispiel sehen wir nun auch, was es heißt, wenn Dateien fragmentiert sind. Also wenn die Daten auf Cluster verteilt sind, die nicht alle hintereinander stehen. Somit muss der Lesekopf der Diskette hin und herfahren, um alle Cluster einzulesen, was ja Zeit beansprucht.

So, ich hoffe, das Beispiel hat es einigermaßen verdeutlicht, wie man sich in einer FAT12-Diskette zurechtfindet.

Anzumerken sei noch, dass Unterverzeichnisse (also nicht das Root-Directory) ebenfalls gespeichert werden. Man speichert in den Cluster dann halt nicht Daten, sondern Directoryeinträge. Und zwar genau 16 Stück pro Cluster. Sollte ein

Unterverzeichnis mehr als 16 Einträge haben, so müssen wir auch hier einfach wieder den nächsten Cluster einlesen auf den der Eintrag in der FAT-Tabelle deutet, um die nächsten Einträge zu erhalten. Wer das System verstanden hat, wird auch kapieren, wenn ich sage, das somit Unterverzeichnisse logisch gesehen unendlich viele Einträge haben können, im Gegensatz zum Root-Directory.

Warum genau man die Einträge des Root-Directorys im Bootsektor genau festlegen muss, bleibt wohl ein Rätsel. Beim FAT32-Dateisystem jedoch hat man dieses Hindernis beseitigt und speichert das Root-Directory dort genauso wie jedes andere Unterverzeichnis.

Bis hier her ist es wohl eine Menge Zeug, das man sich einpauken musste. Jedoch ist es nicht ganz so kompliziert. Da wir aber dennoch ein bisschen Mathematik benötigen, um die Startsektoren von FAT-Tabelle, Root-Directory und den eigentlichen Datensektoren ausfindig zu machen, stelle ich hier noch schnell die wichtigsten Formeln bereit, wie man diese berechnen kann.

4.Cool OS Features

Dies ist eine Übersetzung des Artikels "Cool OS Features" von Tim Robinson, gefunden in der Newsgroup alt.os.development.

Path aliasing

Jeder Windows-Benutzer kennt das Problem, dass man gigantische Pfadbezeichnungen hat. Zum Beispiel: das Desktop-Verzeichnis in meiner Windows 2000-Installation lautet H:\Documents and Settings\Tim Robinson.TIM\Local Settings\Desktop. Auch Unix-Benutzern dürfte dieses Problem bekannt sein.

Wäre es nicht nützlich, eine systemweite Datenbank von Alias-Namen zu haben, die auf ein bestimmtes Verzeichnis zeigen, unabhängig davon, wo dieses Verzeichnis sich befindet? Ich könnte meinem Desktop einen eigenen Alias geben und auf die Dateien mit Pfaden wie Desktop:\24919805.pdf zugreifen (in diesem Fall das Pentium4-Datenblatt, das ich rumliegen habe). Außerdem könnte ich mir viel Ärger beim Bewegen / Umbenennen von Verzeichnissen ersparen. Um z.B eine Visual C++ Include-Datei zu öffnen, würde ich nicht unter F:\Program Files\Microsoft Visual Studio\VC98\Include gehen, ich würde mir einen Alias erstellen, sodass ich bequem über Vc98:\Include auf die Dateien zugreifen könnte. Nun könnte ich das eigentliche Verzeichnis irgendwohin bewegen, der Pfad würde trotzdem gleich bleiben.

Natürlich werden VMS- und Amiga-Benutzer sagen, dass sie so etwas schon haben. Auch Windows hat eine PATH-Variable, die meinen Desktop-Pfad verkürzt (=> %userprofile%\Desktop). Unix benutzt im Dateisystem überall Links. Doch meiner Meinung nach wäre ein standarisiertes, systemweit implementiertes Aliasing-System eine sehr viel elegantere Lösung.

Ein echtes "generalisiertes" Dateisystem

So etwas gibt es schon in Unix, nämlich das "Alles-ist-eine-Datei"-Konzept, das für Windows-Benutzer so unverständlich ist. Das Unix-Dateisystem war so entworfen, dass alle Ressourcen eines Rechners als Dateien in einem zentralen Dateisystem liegen. Neben Programmen und Daten haben wir also die Gerätedateien (in /dev) und spezielle Dateien wie Pipes. Dieses Konzept war vielleicht in den 70er Jahren toll, aber warum hier aufhören? Warum kann man nicht jedes Objekt eines OSes als eine Datei haben mit standarasierten I/O-Operationen?

Zum Beispiel, stellt euch so ein Script vor, um auf Newsgroups zuzugreifen:

Listing 11

```
set f = open \System\Ports\Tcp\news.freeserve.net\nntp
# Using a logical name (see above):
# set f = open NewsServer:\nntp
echo "group alt.os.development" >> f
```

```
echo "article 21786" >> f
set a = read f
220 21786  article
Path: uni-berlin.de!200.43.253.62!not-for-mail
From: nospam@turdera.com.ar (David A. Caabeiro)
Newsgroups: alt.os.development
```

Oder auf einen anderen Rechner zugreifen (à la Windows UNC):

Listing 12

```
copy \Network\fred\public\projects\038\main.c Home:\
```

Ein kleines GUI-Programm:

Listing 13

```
mkdir \System\Windows\test
cd \System\Windows\test
echo "Hello, World!" > Title
copy \System\WindowTemplates\Button .\button1
echo "Click here to exit" > button1\Title
while set a = read Events
  if a.source = button1 then exit
end while
cd ..
rmdir/s test
```

Und so weiter...

5.Kernel-Entwicklung in C

So far, your only experience in operating system writing might have been writing a boot loader in assembly. If you wrote it from scratch it might have taken you several weeks (at least), and you might be wishing there was an easier way. Well: there is, particularly if you are already familiar with the C programming language. Even if you're not familiar with C (and you already know some other high-level language), it's well worth learning it, because it's trivial to start coding your kernel in C. It's a matter of getting a few details correct and having the right tools.

The C Programming Language

C was originally designed as a low-level HLL (Unix kernels have traditionally been written in C) and its main advantages are its portability and its closeness to the machine. It's easy to write non-portable programs in C, but given some knowledge of the language, it's possible to code with portability in mind. There's nothing special about compiled C code that makes it different to assembly - in fact, your assembler is nothing more than a low-level compiler, one in which statements translate directly into machine opcodes. Like any other high-level language, C makes the code more abstract and separates you from the machine more; however, C doesn't place too many restrictions on the code you write.

Tools

There are two main software components involved in generating machine-executable code from C source code: a compiler and a linker. The compiler does most of the work, and is responsible for turning C source code into object files; that is, files which contain machine code and data, but don't constitute a full program on their own. Many

C compilers have an option where they generate an executable file directly from the source code, but in this case the compiler probably just invokes the linker internally.

The linker bundles all the object files together, patches the references they make to each other and moves them about (relocation), and generates an executable file. Most linkers will work with source files from any compiler, as long as the compiler produces compatible object files. This allows you to, for example, code most of your kernel in C but write some of the machine-specific parts in assembly.

I'd recommend any package that contains the GNU gcc compiler and ld linker, because:

- They're free and open-source
- ld supports virtually any executable format known to man
- Versions of gcc are available for virtually any processor known to man

GNU packages are available for various operating systems; the MS-DOS port is called DJGPP, and one good Windows port is Cygwin. All Linux distributions should include the GNU tools. Note that each port generally only supports the generation of code for the platform on which it runs (unless you recompile the compiler and linker), and the bundled run-time libraries are generally of little use to the OS writer: stock DJGPP programs require DOS to be present, and stock Cygwin programs rely on cygwin1.dll, which in turn uses the Win32 API. However it is possible to ignore the default RTL and write your own, which is what we'll be doing. If you're writing code on Windows, I'd recommend that you use Cygwin (patch/rebuild it to support ELF if desired) because it's a lot faster than DJGPP and allows use of long command lines and file names natively. Note that, at the time of writing, Cygwin's flat-binary output is broken.

Various other tools come in useful in OS development. The GNU binutils package (bundled with gcc and ld) includes the handy objdump program, which allows you to inspect the internal structure of your executables and to disassemble them - vital when you're writing loaders and trying a tricky bit of assembly code.

One possible disadvantage of using a generic linker is that it won't support some custom executable format that you invent yourself. However, there's seldom any need to invent a new executable format: there are already a lot of them out there. Unix ELF and Windows PE tend to be most popular among OS writers, mainly because they're both well supported among existing linkers. ELF seems to be used more in amateur kernels, mainly because of its simplicity, although PE is more capable (and hence more complex). Both are well documented. COFF is also in use in some projects, although it lacks native support for features such as dynamic linking. An alternative for simple kernels is the flat binary format (i.e. no format at all). Here, the linker writes raw code and data to the output file, and the result is similar to MS-DOS's .COM format, although the resulting file can be larger than 64KB in length and can use 32-bit opcodes (assuming the loader enables protected mode first).

A disadvantage of all the mainstream IA-32 compilers is that they assume a flat 32-bit address space, with CS, DS, ES and SS each having the same base address. Because they don't use far (48-bit seg16:ofs32) pointers they make it a lot harder to write code for a segmented OS. Programs on a segmented OS written in C would have to rely on a lot of assembly code and would be a lot less efficient than programs written for a flat OS. However, at the time of writing, Watcom are still promising to release their OpenWatcom compiler, which will supposedly support far pointers in 32-bit protected mode.

Here's another couple of warnings about using gcc and ld (although these could potentially apply to any compiler linker). Firstly, gcc likes to put the literal strings used by functions just before the function's code. Normally this isn't a problem, but it's caught out a few people who try to get their kernels to write "Hello, world" straight off. Consider this example:

Listing 14

```
int main(void)
{
        char *str = "Hello, world", *ch;
        unsigned short *vidmem = (unsigned short*) 0xb8000;
        unsigned i;

        for (ch = str, i = 0; *ch; ch++, i++)
                vidmem[i] = (unsigned char) *ch | 0x0700;

        for (;;)
                ;
}
```

This code is intended to write the string "Hello, world" into video memory, in white-on-black text, at the top-left corner of the screen. However, when it is compiled, gcc will put the literal "Hello, world" string just before the code for main. If this is linked to the flat binary format and run, execution will start where the string is, and the machine is likely to crash. There are a couple of alternative ways around this:

- Write a short function which just calls main() and halts. This way, the first function in the program doesn't contain any literal strings.
- Use the gcc option -fwritable-strings. This will cause gcc to put literal strings in the data section of the executable, away from any code.

Of these, the first option is probably preferable. I like to write my entry point function in assembly, where it can set up the stack and zero the bss before calling main. You'll find that normal user-mode programs do this, too: the real entry point is a small routine in the C library which sets up the environment before calling main(). This is commonly known as the crt0 function.

The other main snag concerns object-file formats. There are two variants of the 32-bit COFF format: one used by Microsoft Win32 tools, and one by the rest of the world. They are only subtly different, and linkers which expect one format will happily link the other. The difference comes in the relocations: if you write code in, say, NASM, and link it using the Microsoft linker along with some modules compiled using Visual C++, the addresses in the final output will come out wrongly. There's no real workaround for this, but luckily most tools that work with the PE format will allow you to emit files in the Win32 format: NASM has its -f win32 option, and Cygwin has the pei-i386 output format.

The Run-Time-Library

A major part of writing code for your OS is rewriting the run-time library, also known as libc. This is because the RTL is the most OS-dependent part of the compiler package: the C RTL provides enough functionality to allow you to write portable programs, but the inner workings of the RTL are dependent on the OS in use. In fact, compiler vendors often use different RTLs for the same OS: Microsoft Visual C++ provides different libraries for the various combinations of debug/multi-threaded/DLL, and the older MS-DOS compilers offered run-time libraries for up to 6 different memory models.

For the time being, though, it should be sufficient to write only one run-time library (although writing a makefile which offers a choice of static or dynamic linking is often useful). You should aim to replicate the library defined by the ISO C standard because this will make porting programs to your OS easier. If you write a non-standard library you'll have to re-write any applications you want to port; the more standard library functions you define, the easier it will be to port open-source applications straight across.

It makes it a lot easier to write library code if you can get hold of the source code for an existing implementation, particularly if that library's environment is similar to the one you're developing in. There are a lot of C functions which are OS- and platform-independent: for example, most of string.h and wchar.h can be copied straight across. Conversely, there are a lot of functions that your compiler might offer

which won't make sense for your OS: a lot of DOS compilers offer bios.h header file which allows access to the PC BIOS. Unless you write a VM86 monitor in your kernel, you won't be able to call BIOS functions directly. However, such functions will be extensions to the C standard library, and, as such, will have names beginning with an underscore (e.g. _biosdisk()). You are under no obligation to implement these extensions, and you're free to define your own extensions if you wish, as long as you give them a name starting with an underscore.

Other C library functions are dependent on kernel support: most of stdio.h relies upon there being some sort of file system present, and even printf() needs some destination for its output. On the subject of printf(): most open-source C library implementations will define a generic printf() engine because the same functionality is needed in so many different places (printf(), fprintf(), sprintf(), and the v, w and vw versions of these functions). You should be able to extract this common engine and use it for your own, or at least try to emulate it: write a function which accepts a format string and a list of arguments and which sends its output to some abstract interface (either a function or a buffered stream).

Although a full run-time library is most useful when it comes to writing and porting user applications, it is also convenient to have good RTL support in the kernel. Including commonly-used routines (such as strcpy() and malloc()) will make it quicker and easier to write kernel code, and it will make the resulting kernel smaller, since common routines are only included once. It is also good practice to make such routines available to driver writers, to save them from having to include common routines in their driver binaries.

OS-Specific Support

As a kernel language, C isn't perfect: obviously, there's no standard way of allowing access to a particular machine's features. This means that it's often necessary to either drop into inline assembly when it is needed, or to code portions in assembly and link them with the compiler's output at link time. Inline assembly allows you to write parts of a C function in assembly, and to access the C function's variables as normal: the code you write is inserted among the code the compiler generates. Support for inline assembly varies among compilers, if you can stand the AT+T syntax, gcc's is best among PC compilers. Although Visual C++ and Borland C++ both use the more familiar Intel syntax, their inline assemblers aren't as fully-integrated with the rest of the compiler. gcc forces you to use more esoteric syntax but it allows you to use any C expression as an input to an assembler block and it allows you to place the outputs anywhere. The resulting assembly is also better integrated with the optimizer than in Borland's and Microsoft's compilers: for example, gcc allows you to specify exactly which registers are modified as the result of an assembly block.

Some assembly code will also be required in user mode if you use software interrupts to invoke kernel syscalls. You might place the interrupt calls directly into the C library code (as MS-DOS libraries do), or generate a separate library with a function for each call and link normally to that (as Windows NT does with ntdll.dll). It may also make sense to put the OS-dependent interface in a language-independent library, so that programs written in different languages can use the same OS library. This will allow people to write programs for your OS in languages other than C, or in languages which don't allow C bindings.

C++ in the Kernel

Opinions vary widely over the use of C++ in a kernel: most Linux kernel coders stay well away from it, whereas some people have entire operating systems in C++. I'd personally stick with C, although I see no real reason why you shouldn't use C++, as long as you know what you're doing. One thing you should realize is that to write a C++ kernel you'll need to write a bit more code for the framework, and that some C++ features are off-limits to you (unless you can write the necessary support code).

First off, you'll need to code the new and delete operators. This is easy if you've already coded malloc() and free(): new and delete can be single-line functions which call each of these. If you want to have global instances of classes you'll need to put some code in your startup routine to call each of their constructors. The way this is done will differ between compilers, so the best thing to do is to browse through your

compiler's run-time library to see how the vendor did it; try looking in files with names like crt0.c. You'll probably also need to implement atexit(), because the compiler is likely to emit code which uses atexit() to call global object's destructors when the program ends. If you want to use try/catch in your kernel you'll have to implement whatever mechanism your compile uses to implement them. Again, this will depend on the compiler you use, and whatever operating system that compiler targets.

In general, I'd steer clear of both exception handling and huge virtual classes in your kernel. Exception handling usually adds unneeded bloat, and calling masses of virtual functions add unnecessary indirection and defeats the optimizer somewhat. Remember, your kernel should be as efficient as possible, even to the detriment of a beautiful design where necessary.

Writing an OS in a high-level language such as C can be a lot more productive than coding one entirely in assembly, particularly if you're willing to forego the slight speed increase that assembly offers. There are compilers freely available, such as gcc, which make writing kernel code relatively easy, and using C will probably prove beneficial in the long run.

6.phil's Kolumne

Über Low-End-PCs

Früher habe ich immer über die Leute gelacht, die zu Hause uralte Rechner hatten. Leute, die angeblich kein Geld hatten, um sich einen neuen Rechner zu kaufen. Doch in den letzten paar Monaten hat sich meine Ansicht zu diesem Thema ein wenig verändert...

Angefangen hat es mit dem Besuch bei Mastermesh. Wir haben uns an einem Nachmittag zusammengesetzt und über die Zukunft des Magazins nachgedacht. Dabei hat Mastermesh mir auch seine "Workstation" gezeigt, an dem er das Magazin schreibt. Als er mir genau beschrieb, was das für ne Maschine war, bin ich vor Lachen fast geplatzt.

Pentium I-Prozessor, 233 MHz, 64 MB RAM, 2 GB Festplatte. Ich dachte erstmals, dass das ein dummer Witz war. Aber es hat gestimmt. Abseits von seinem Hauptrechner hat Mastermesh noch diese vorsintflutliche Workstation, an der er das Lowlevel-Magazin schreibt. Und der Rechner war auch noch ziemlich schnell. Hat weniger als eine Minute gebraucht, um Windows 98 zu booten. Unglaublich, oder?

Dieser "Zwischenfall" hat mich doch sehr nachdenklich gemacht. Als ich zu Hause angekommen war, sah ich in der Garage diesen alten Pentium2 400, den ich schon seit Jahren wegschmeißen wollte. Doch nun habe ich mir überlegt, für was ich ihn benützen könnte. Und tatsächlich, ich hatte etwas gefunden.

Seit ein paar Tagen benutze ich diesen Rechner, um auf meiner DBox2 Neutrino-Linux laufen zu lassen. Der Kernel und die Ramdisk werden über TFTP / PPCBoot von meinem Arbeitsrechner gezogen und gleich gebootet.

Und die Moral von der Geschicht? Tja... ich weiß nicht. Schmeißt eure Rechner nicht einfach weg, wenn ihr einen Neuen kauft. Auch alte Rechner können nützlich sein.

phil

7.Nachwort

Mit viel Mühe habe ich doch noch vor diese verdammte Ausgabe fertig gekriegt. Hoffen wir mal, dass es mit den nächsten Ausgaben besser läuft...

Mastermesh