

Inhaltsverzeichnis

	Seitenzahl
1. Einleitung und Übersicht	1
1.1 Funktionen von Tabos	1
2. Kernel Design	2
2.1 Module	2
2.2 Memory Management	3
2.3 Timer	4
2.4 Tasksystem	5
2.5 IPC	8
2.6 Device Manager	9
2.7 VFS	9
2.8 Device Filesystem	9
2.9 Graphical User Interface	10
3. Booten des Systems	10
3.1 Die verschiedenen Medien	10
3.2 Boot Parameter	11
3.3 Der Boot Prozess	11
4. Wie benutzt man Tabos?	12
5. Das Verzeichnis Layout	12
6. Kompilierung des Systems	13
7. Anhänge	13

1. Einleitung und Übersicht

Tabos ist ein kleines, modulares und zugleich effizientes 32-bit Protected-Mode [4] Betriebssystem für x86 kompatible Computer. Es wurde von uns zum einen als Unterrichtsmaterial entwickelt, zum anderen aber als allgemeines Server/Desktop Betriebssystem. In der jetzigen Form passt es mit allen vorhandenen Treibern und Programmen auf eine 1.44MB Diskette (ca. 500kb, wobei der Kernel je nach Bedarf zwischen 50-70kb groß ist).

Eine der vielen besonderen Eigenschaften ist die Schnelligkeit auf älteren Systemen, so dass man es ohne Probleme auf alten 386er laufen lassen kann. Weiterhin legen wir viel Wert auf Stabilität (Virtual Memory, geschützte Speicherbereiche/Ports, registrierte IRQs und DMAs...), Sicherheit (verschlüsseltes Dateisysteme) und Einfachheit. Demnächst wird dazu eine GUI hinzukommen, die dem User einfachen Zugriff auf alle Komponenten ermöglichen wird.

Um diese drei Punkte zu gewährleisten, bauen wir auf den 32-bit Protected-Mode auf, der uns viele Optionen gibt, um Programme, Module und den Kernel zu trennen. Neben diesen Punkten ist uns auch die Portabilität auf andere Systeme sehr wichtig, weshalb wir den Kern von den eigentlichen low-level code trennen.

Minimalen Systemanforderung für Tabos:

- 80386
- 16MB RAM
- Grafikkarte
- Keyboard
- Floppy Laufwerk

Empfohlene Systemanforderung:

- 80386
- 32MB RAM
- Grafikkarte
- Keyboard
- Floppy, Festplatte

1.1 Funktionen von Tabos

Im Moment hat Tabos die folgenden Funktionen:

- 32-Bit Protected-Mode
- Multitasking
- Virtual Memory
- Dateisysteme
 - Virtuelles Dateisystem (VFS)
 - Geräte-Dateisystem (DevFS)
 - FAT12/16/32 Dateisystem
 - ISO9660 Dateisystem (CDFS)
- Unterstützung für Module im ELF-Format
- IDE Unterstützung für Festplatten, CD-ROM, CD-Writer und DVD
- tw. Parallel ZIP Drive 100 Unterstützung
- Floppy Treiber
- ISA Karten - Erkennung und Konfiguration

02/14

- PCI Karten - Erkennung und Konfiguration (Direct Access)
- Keyboard Treiber mit verschiedenen Layouts (US, UK, DE)
- Mäuse:
 - PS/2 Maus
 - Logitech Bus Maus
 - Microsoft Bus Maus
 - Serielle Maus
- Hardware-Sensor für Winbond-Chipsätze
- Creative Soundblaster oder Kompatible (SB 2.0)
- Standard VGA Treiber (320x200x256, 640x480x16)
- Null Block Treiber
- Serieller Schnittstellen-Treiber
- Paralleler Schnittstellen-Treiber
- Real Time Clock
- IPC
- Spinlocks
- Block Cache

2. Kernel Design

2.1 Module

Module sind eine spezielle Art von Programmcode, die bei Bedarf zur Laufzeit nachgeladen und in den Kernel integriert werden. Module werden von Tabos dabei generell nur im ELF-Object Format [3] akzeptiert, zusätzlich muss auf eine Reihe von Merkmalen geachtet werden, um das Erkennen und Laden eines Modules zu ermöglichen.

Zunächst eine kurze Erklärung zum ELF-Object Format, bevor der Prozess des Modulladens näher erklärt wird.

Wie schon erwähnt, wird in Tabos das ELF-Object Format für Module verwendet. Bei der Erstellung des Modul bringt der Compiler den Modul-Sourcecode nicht in direkt ausführbare Programmform, da ein Modul später ja Bestandteil des *Kernels* sein soll, nicht aber direkt ausführbar.

Beim ELF-Object Format werden alle internen Speicherreferenzen (dies sind: globale Variablen [lokale Variablen werden bei Bedarf auf dem Kernelstack erzeugt], lokale und globale Funktionsaufrufe, globale Konstanten) sowie alle externen Speicherreferenzen (dies sind alle Referenzen, die nicht im Modulcode, aber irgendwo im Kernel implementiert werden) mit zum Modulbeginn *relativen* Adressangaben abgelegt. Diese Angaben werden in speziellen, vom Compiler erzeugten, Tabellen an fest definierten Stellen im Modulcode abgelegt. Diese sogenannten *Reloziierungstabellen* (*engl.: relocation tables*) werden dann während des Ladeprozesses von einer speziellen Kernelfunktion (in `kernel/bfmt/elf_module.c:512, relocate_module()`) geparkt und die relativen Speicherangaben durch die absoluten ersetzt, abhängig an welche Stelle im Hauptspeicher der Modulcode geladen wurde.

Wie schon erwähnt, können auch externe, also nicht im Modulcode definierte, Variablen und Funktionen von einem Modul benutzt werden. Während der

Reloziierung eines Modules muss der Reloziierer jedoch wissen, an welcher Stelle im Hauptspeicher eine Funktion oder Variable definiert wurde. Diese Aufgabe erfüllen die *export_tables* (siehe `include/kernel/export_table.h`).

```

struct export_table_entry
{
    char *export_name;
    unsigned long exort_addr;
    int used;
};

struct export_table
{
    struct export_table *next, *prev;
    struct export_table_entry *entries;
};

```

Zur Bootzeit wird eine initiale *export_table* erstellt, die alle *Symbole* (das sind Variablen, Funktionen) enthält, die als solche zu exportierende deklariert wurden (siehe `kernel/exported_symbols.c`). In `kernel/ksymtab.c` wird dazu die Funktion `export_kernel_symbols()` aufgerufen, die eine erste *export_table* erstellt und alle zur Bootzeit verfügbaren Symbole einträgt. Weiterhin wird für jedes geladene Modul eine neue *export_table* angelegt, wobei *alle* Symbole des Moduls exportiert werden (es gibt also keinen direkten Mechanismus, um bestimmte Symbole eines Moduls vor der Exportierung zu schützen). Diese neue *export_table* wird in eine globale Liste (`kernel/ksymtab.c:52`) eingetragen, so dass nun bei Bedarf für ein zu ladendes Modul diese *export_table* liste nach dem zu suchenden Symbol geparkt und bei Erfolg reloziert werden kann. Kann auch nur ein Symol eines Moduls (egal ob interne/externe Referenz) nicht reloziert werden, schlägt das Laden des Moduls fehl und die Funktion `relocate_module()` gibt einen Errorcode zurück. Können alle Symbole (Referenzen) aufgelöst werden, gibt die Funktion `relocate_module()` einen Zeiger auf eine spezielle Struktur zurück, die die Adresse von einer `module_init()` Funktion, sowie weiteren, für den internen Gerauch wichtigen Funktionen, enthält. Diese `module_init()` Funktion wird schliesslich direkt nach erfolgreichem Laden des Moduls aufgerufen und kann zum Beispiel die Initialisierungsroutine für einen Treiber enthalten.

Den generellen Aufbau eines Moduls beschreibt die Datei `drivers/kmod/mod.c`, welche auch das erste während des Bootprozesses geladene Modul enthält.

2.2Memory Management

Memory Managment [1] ist die Komponente im Kernel, die die genaue Verteilung und Verwendung des kompletten Hauptspeichers verwaltet. Zur Bootzeit ist es nötig, bestimmte Speicherbereiche zu belegen bevor die eigentliche Speicherverwaltung, wofür der *boot_allocator* (in `kernel/mm/bootmm.c`) implementiert wird. Er ist aber nur zur Bootzeit verfügbar und auch nur in sehr beschränktem Ausmass. Das eigentliche Memory Managment teilt sich daher in 2 Bereiche auf.

Der erste Teil verwaltet die sogenannten *Pages* (der komplette Hauptspeicher ist in Tabos in Pages, das sind jeweils 4096 Bytes grosse Speicherabschnitte, unterteilt). Dazu kommt die *memory_map* Methode zum Einsatz: jede Page des Speichers wird in einem Array (`static unsigned char mm_map[]`) gespeichert. Das Array wird zur

Bootzeit der Grösse des Hauptspeichers entsprechend angelegt. Die Funktionen `__get_free_page()`, `__get_free_pages()` reservieren bei einem Aufruf entsprechend eine oder mehrere pages.

Der zweite Teil wird vom `kernel_memory_allocator` kurz `kmalloc` gebildet. `kmalloc()` besitzt die Möglichkeit, Speicherbereiche zwischen 32 Bytes und 128 KByte zu liefern (während der `page_allocator` nur Seitengrösse oder vielfache davon liefert). Der allocator arbeitet nach dem *slab-verfahren*. Dabei werden Speicherbereiche einer gleichen Grösse (*order*) in der Struktur `struct order_header_orders[]` zusammengefasst. Die Funktion `kfree()` befreit einen gewünschten Speicherbereich wieder.

2.3Timer

In Tabos werden beide auf der Hauptplatine vorkommenden Zeitgeräte verwaltet: der *programmable interval timer*, kurz *pit* oder 8253, und die *real time clock*, kurz *rtc*. Beide Geräte sind grundlegend unterschiedlich in ihrem Aufbau (siehe entsprechende Hardware Dokumentation) und werden in Tabos auch für verschiedene Aufgaben benutzt.

Der *pit* ist ein Zeitgeber mit relativ grober Auflösung. Sie kann während der Initialisierung frei eingestellt werden, in den meisten Systemen im 100 ms Bereich. In Tabos wird sie exakt auf den Wert 100 ms eingestellt.

Nach der Initialisierung bewirkt der *pit* dann periodisch im eingestellten Zeitintervall eine Unterbrechungsanforderung (`irq0`), welche eine Interruptroutine (in `kernel/timer.c: 134, timerhandler()`) ausführt. Diese Interruptroutine bewirkt in Tabos ein Aufruf der `schedule()` Funktion, was einen Threadwechsel ausführt. Zusätzlich zu dieser festeingebauten Aktion besteht die Möglichkeit, eigene Timerevents zu definieren.

Mit den Funktionen

```

add_timer( lifetime, handle, argument, mode )
remove_timer( timer_structure )

struct timer
{
    struct timer *next, *prev;
    unsigned long lifetime, value, mode;
    void (*event)( void *data );
    void *data;
}

static struct timer *timer_events

```

können eigene Aktionen festgelegt werden. Der Wert *lifetime* bezeichnet dabei die Zeiteinheit, die vergeht, bis der Event (übergeben als Zeiger in *handle*) stattfindet. `add_timer` legt während dessen Aufrufs eine neue `timer_structure`, füllt die entsprechenden Felder und verlinkt den neuen Timer mit `timer_events`, der globalen `timer_events` liste.

Bei jedem Aufruf des `irq0 timer_handler()` wird diese globale Liste mittels `do_timer()` (`kernel/timer.c`) geprüft, ob ein `timer_event` seine Zeitgültigkeit erreicht hat. Ist dies

05/14

der Fall, ruft `do_timer()` den angegebenen `timer_event` auf.

Neben den zwei beschriebenen Funktionen fungiert der `pit` noch als interner Zeitzähler. Bei jedem Aufruf des `irq0 timer_handler` wird die Variable `system_ticks` (in `kernel/timer.c`) um einen Ganzzahlwert inkrementiert. Nachteil ist jedoch der, dass nach einem Neustart nicht mehr geprüft werden kann, wie lange das System insgesamt up war.

Die zweite Zeiteinheit ist wie schon oben erwähnt die *real time clock*. Sie unterscheidet sich vom `pit` dadurch, dass sie genaue Zeitabgaben auch nach einem Neustart ermöglicht. Neben der Eigenschaft periodisch einen Interrupt auszulösen (der `irq` Mechanismus ist sehr ähnlich zu dem des `pit`), bietet die `rtc` vor allem die Eigenschaft, eine interne Uhr weiter zu betreiben, auch wenn der PC ohne Strom ist. Die Energieversorgung dazu kommt aus einer kleinen Batterie auf dem Mainboard, welche neben der `rtc` auch weitere Speichereinheiten auf der Platine nach dem `power-off` mit Strom versorgt.

Nach dem Setup der `rtc` mittels `init_rtc()` (in `kernel/rtc.c`) kann die genaue Zeit mittels `sys_get_date()` abgefragt werden. Dazu muss vorher ein Speicherbereich geholt und `sys_get_date()` als Argument übergeben werden.

```
Struct rtc_td
{
    unsigned int dayweek:3;
    unsigned int daymonth:5;
    unsigned char month;
    unsigned short year;
    unsigned char second;
    unsigned char minute;
    unsigned char hour;
}

int sys_get_date( struct rtc_td *rtc_td );
```

Die Funktion liest daraufhin den kompletten Inhalt der `rtc` und schreibt ihn an den angegebenen Speicherbereich.

2.4 Tasksystem

Das in Tabos implementierte Prozesssystem wurde bewusst einfach gehalten. Grundlegend wird zwischen Prozessen und sogenannten Threads [2] unterschieden. (siehe dazu auch: in `kernel/process/proc.h`)

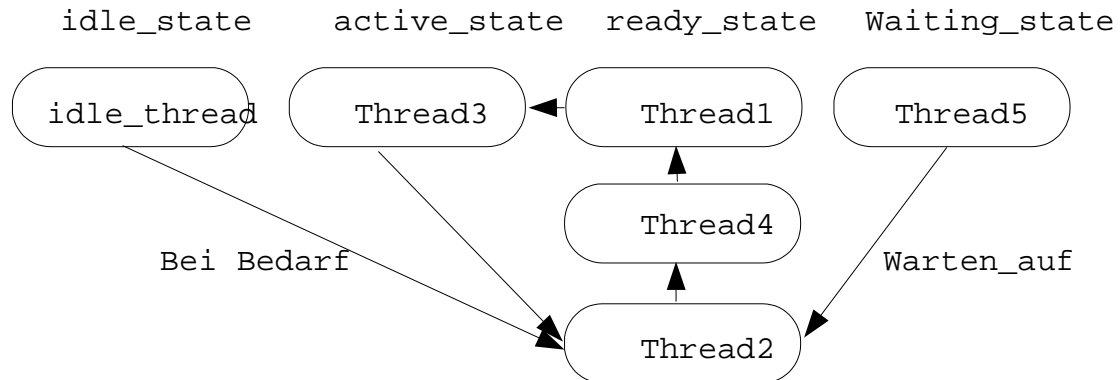
```
struct _thread { // enthält die Definition für einen Thread }
struct _proc { // enthält die Definition für einen Prozess }
```

Ein Prozess ist eine logische Struktur, die einen oder mehrere Threads („Aufgaben“) enthält. Beim Start eines neuen Programms wird eine neue Prozessstruktur mit einem Thread (dem `main_thread`) angelegt. Dieser `main_thread` enthält die `main()` Funktion des Programmes. Während der Programmausführung können neue Threads gestartet werden oder auf andere gewartet werden.

Die Gesamtheit aller Prozesse bildet den Prozesszustand des Computers. Dabei

können in Tabos alle Prozesse aktiv sein, mehrere Prozesse unterschiedliche Zustände besitzen (schlafen, warten, suspendiert) oder aber alle Prozesse nicht aktiv sein. Bei diesem Spezialfall wird der sogenannte idle-Prozess ausgeführt. Dieser Prozess wird als erster von allen Prozessen während der Systeminitialisierung angelegt und nur dann ausgeführt, wenn alle anderen Prozesse nicht aktiv sind.

Prozesszustand zu einem beliebigen Zeitpunkt:



Grafik zeigt den groben Ablauf eines Threadwechsels

Multitasking wird in Tabos dadurch erreicht, dass jeder Thread innerhalb eines Prozesses eine feste Zeit zum Rechnen hat, wonach er dann aus dem aktiv-Zustand in den ready-Zustand überführt wird. Wichtigste Komponente dabei ist die Funktion `schedule()` in `kernel/process/sched.c`. Sie wird periodisch bei jedem Auftreten des Timerinterrupts (Systemzeitgeber, etwa alle 100ms) aufgerufen und bewirkt den Threadwechsel. Der in Tabos implementierte scheduler arbeitet strikt nach dem „round-robin“ Verfahren, wobei jedem Thread die exakt gleiche Rechenzeit zugewiesen wird. Das benachteiligt zwar Threads, die eine hohe Dringlichkeit haben, bewirkt aber einen sehr guten mittleren Durchsatz bei gleichzeitiger Wartungsfreundlichkeit des Codes.

Wird ein solcher `schedule`-event durch den timer ausgelöst, wird der Status des aktiven Threads in den ready-Status überführt, die gesamten Prozessdaten werden in einem speziellen Bereich im Hauptspeicher gesichert und der Prozess wird, sofern kein spezieller Systemaufruf den Prozess zwingt, seine Rechenzeit frühzeitig aufzugeben, an das Ende einer speziellen Warteschlange, der sogenannten ready-queue, gehängt. Danach lädt der scheduler das oberste Element der ready-queue, welches den nächsten auszuführenden Thread enthält. Prozessdaten von diesem Thread werden aus dem Hauptspeicher geladen und die Ausführung an der Stelle fortgesetzt, an der dieser Thread unterbrochen wurde.

In Tabos wird zum Threadwechsel die *stack-switching* Methode zum Prozesswechsel benutzt. Sie unterscheidet sich zur älteren *tss-jumping* Methode in mehreren Punkten. Obwohl *tss-jumping* einfacher zu implementieren ist, bringt es einige Nachteile mit sich. Zunächst aber die Arbeitsweise beider Methoden.

Die x86-Architektur kennt zur Verwaltung eines Programmthreads (auf cpu-Ebene wird der Begriff thread unterschiedlich zur in Tabos benutzen Bedeutung verwendet) dass *task-state-segment* (kurz *tss*), dass alle wesentlichen

Objekte und Maschinenzustände, wie sie der Programmthread zur Laufzeit hält, beinhaltet. Dies sind im speziellen alle Register der cpu sowie Steuerflags und spezielle Zeiger. Wird ein Kontextwechsel vorgenommen, muss der Programmierer dafür Sorge tragen, dass der komplette Zustand des ausgeführten Threads im tss gespeichert wird, bevor er den nächsten Thread (weiter-)ausführen kann. Dazu bedient er sich einem einfachen *jump*-Befehl, wobei er an die Adresse des Bereiches steht, der das tss des neuen Threads enthält. Die cpu führt dann einen speziellen, fest implementierten *microcode* aus, um das tss des neuen threads wiederherzustellen und die Programmausführung an der unterbrochenen Stelle fortzusetzen.

Im allgemeinen ist diese Methode einfacher zu implementieren, hat aber zwei entscheidende Nachteile: zum einen hat der Code eine sehr schlechte Performance, da generell sehr viele Daten in das tss gesichert und wieder hergestellt werden müssen, zum anderen sind viele der Daten schon während des timer-interrupts auf dem Stack gesichert worden oder müssen generell nicht gesichert werden, da alle Threads identische Daten verwenden.

Daher liegt es nahe, den Stack komplett zum Kontextwechsel zu verwenden. Während des timer-interrupts werden also *alle* relevanten Daten des Threads auf dessen Stack gesichert, um in einer speziellen Funktion dann nur noch die *Stackzeiger ss:esp* mit dem neuen Thread zu vertauschen. Da jeder Thread den gleichen Code während eines (Timer-)interruptes durchläuft, wird garantiert, dass der neue Stack in gleichem Maße wiederhergestellt wird, wie Daten vom alten Thread auf den Stack gebracht wurden (*die Stackoperationen sind spiegel-identisch*). Der komplette Kontextwechsel wird also nicht wie bei der tss Methode durch Kopieren von Inhalten bewirkt, sondern durch Vertauschen eines einzigen Zeigerpaares. Die Implementation dieses Verfahrens muss lediglich sicherstellen, dass der Stackwechsel erfolgreich abläuft, dass also wie oben erwähnt, Stackoperationen spiegel-ähnlich stattfinden.

Der zum Kontextwechsel (*engl.: context switch*) relevante Teil des codes ist (in `kernel/process/context_Switch.S`):

```
movl 48(%esp), %eax    // <- lade zeiger auf neuen stack
lss (%eax), %esp      // <- lade neuen stack in esp
```

Nachdem der Stackzeiger neu geladen wurde, operieren alle Stackoperationen auf dem neuen Stack. Mittels

```
popl    %eax
movw   %ax,%ds
popl    %eax
movw   %ax,%es
popal
ret
```

werden also die Maschinenregister wiederhergestellt, sowie der Programmcode an der Stelle fortgesetzt, an der er unterbrochen wurde.

2.5 IPC

IPC (*Inter-Process-Communication*) wird durch sogenannte *Message-Ports* ermöglicht. Ein Message-Port ist eine Datenstruktur (siehe unten), die durch einen

08/14

Aufruf von `sys_port_create()` (in Applikationen durch `port_create()`) erstellt wird. Jeder Message-Port bekommt eine systemweit einzigartige `port_id`, anhand derer er eindeutig auch für andere Prozesse zu identifizieren ist. Message-Ports werden nach ihrer Erstellung *lokal*, das heisst, von der Processgruppe, in dessen der erstellende Thread liegt, verwaltet. Dadurch ist sichergestellt, dass zunächst nur Threads aus der gleichen Processgruppe kommunizieren können. Will ein Thread einen Port für andere Prozesse öffnen, so muss dieser Thread seine `port_id` an einen zentralen Platz exportieren.

Mit den beiden Funktionen `sys_port_read()` bzw. `sys_port_write()` können Nachrichten von einem Port gelesen werden bzw. auf diesen geschrieben. Die ports sind *Inhaltsgesteuert*, das heisst, ein Thread wartet bei einem Leseversuch auf einen leeren Port solange, bis dieser Port eine Nachricht erhält. Umgekehrt wartet ein Thread bei einem Schreibversuch auf einem vollen Port (die Maximalanzahl von gepufferten Nachrichten wird beim Aufruf von `sys_port_create()` eingestellt) solange, bis Platz für die Nachricht ist. Mit der Funktion `sys_port_count()` wird die Anzahl der Nachrichten auf dem Port abgefragt. Mit `sys_port_close()` wird ein Port geschlossen. Jeder Schreibversuch auf einen geschlossenen Port schlägt automatisch fehl, auf lesende Prozesse wird gewartet, so dass keine Nachrichten verloren gehen. `sys_port_destroy()` zerstört sofort bei Aufruf einen Port, lesende Prozesse werden nicht mehr berücksichtigt.

Ports-structure aus include/kernel/process/msg.h:

```
struct _port
{
    struct _port *port_next, *port_prev;
    struct _port *group_next, *group_prev;
    struct _proc *pgroup;

    int port_id;
    int port_limit;
    int port_closed;
    int port_destroyed;

    int bytes_used;

    struct t_spinlock port_lock;
    struct _sem *read_sem;
    struct _sem *write_sem;

    struct _msg_queue msg_queue;
};
```

2.6 Device Manager

In Tabos unterscheiden wir zwischen 3 Gerätetypen: INFO, CHAR und BLOCK. INFO Geräte sind reine Informationsquellen, d.h. sie liefern nur bestimmte Informationen vom Kernel an die Programme und haben keine Möglichkeiten weiteren Einfluss auf das System zu nehmen. In der jetzigen Form werden sie als Informationsgerät für das Memory Management (Ausgabe über den Speicherzustand, siehe /apps/ps/) und das Process System (Ausgabe über laufende

09/14

Prozesse, siehe /apps/ps/ und /apps/top/) benutzt.

CHAR (Character) Geräte sind, wie der Name schon suggeriert, Zeichengeräte, d.h. sie sind für die Eingabegeräte verantwortlich. Zu Eingabegeräten gehören unter anderem Keyboards, Mäuse, und Joysticks. Sie besitzen neben der Leseoption auch die Fähigkeit, Eingaben der Programme entgegen zu nehmen und direkt darauf zu reagieren. Des Weiteren hat dieser Gerätetyp die Möglichkeit sich selbst zu erweitern. Dies geschieht durch eine ioctl-Funktion, die neben einen Datei-Informationen Zeiger das Kommando und den dazugehörigen Speicherzeiger entgegennimmt.

*int ioctl(struct file *file, unsigned int cmd, unsigned int argument)*

BLOCK Geräte haben wie die zwei vorherigen Gerätentypen Lese- und Schreibzugriffe und die ioctl-Funktion. Im Gegensatz zu den CHAR Geräten kümmern sie sich aber um die BLOCK Geräte, also angeschlossene Datenträger im System. Zu ihren Aufgaben gehört es neben der Erkennung der Datenträger, den Ein-/Ausgaberoutinen auch die Wartung des Gerätes (Formatierung).

Zu den einzelnen Aufgaben des Device Manager gehört es, die einzelnen Geräte im System zu registrieren und gegebenenfalls benötigte Interrupts, DMAs und Ports zu reservieren. Diese können dann nicht mehr von anderen Geräten bzw. Programmen benutzt werden. Somit bietet der DeviceManager einen Schutz vor unbekanntem Treibern, die illegalerweise direkten Hardwarezugriff benutzen wollen.

2.7 Virtual FileSystem (VFS)

Im Gegensatz zu anderen Dateisystemen handelt es sich bei dem Virtual FileSystem um ein temporäres Dateisystem. Das VFS wird zu Beginn des Bootens als erstes Dateisystem registriert und bildet die Basis für alle weiteren Dateisysteme. Anstatt wie bei Windows/DOS einzelnen Datenträgern einen Laufwerksbuchstaben zu zuweisen, werden diese in dieses virtuelle Dateisystem als Verzeichnis hinzugefügt (gemountet). Somit ist das System nicht auf 26 Datenträger beschränkt, was vor allem im Serverbereich ein wichtiges Kriterium ist. Des Weiteren bildet das Virtual FileSystem eine Zwischenschicht zwischen den Programmen und den gemounteten Dateisystemen. Es übernimmt die Behandlung von Dateien und Verzeichnissen und leitet gegebenenfalls entsprechende Befehle direkt an das gemountete Dateisystem weiter.

2.8 Device Filesystem

Neben den VFS existiert ein weiteres virtuelles Dateisystem, das DevFS. Im DevFS werden alle im System angesprochene Geräte als virtuelle Datei angelegt, um den einzelnen Programmen und dem Kernel selbst Zugriff auf diese Geräte zu ermöglichen. Dieses Design wurde bewusst gewählt, da sich Geräte in diesem Fall genau wie eine Datei verhalten und auch mit den gleichen Befehlen angesprochen werden. Deshalb ist es sehr leicht möglich, anstatt eines Gerätes eine Datei in das VFS zu mounten. Für dieses Verhalten würde man in vielen anderen Betriebssystemen wieder zusätzliche Programme benötigen, die in diesem Fall logischer Weise entfallen. Des Weiteren ist es möglich die einzelnen Geräte in Unterverzeichnisse zu gliedern, das vor allem zu der Übersichtlichkeit und der

010/14

Sauberkeit des Systems beiträgt.

2.9 Graphical User Interface

Die Oberfläche (GUI) wird im Laufe der Zeit einen entscheidenden Faktor in der Entwicklung von Tabos einnehmen. Das System ist in der Lage die im System integrierten Grafikkarten selbstständig zu erkennen und aus einer Treiberliste den passenden Treiber zu laden. Im Moment benutzen wir den integrierten Standard-VGA Treiber, der als Notfall-Lösung im Kernel eingebunden ist. Falls der Kernel nicht in der Lage ist, den benötigten Treiber zu laden oder einen passenden zu finden, so wird der Standard-VGA Treiber aktiviert, so dass der Benutzer trotzdem in der Lage ist, eine funktionstüchtige GUI laufen zu lassen.

Die Arbeit zwischen den verschiedenen Grafiktreibern und der GUI übernimmt die *libvideo*. Diese Bibliothek enthält grundlegende Funktionen, um die Grafikkarten anzusprechen und bietet der GUI allgemeine Funktionen an, die sofern der Treiber diese unterstützt, an diesen weitergeleitet werden. Für den Fall, dass der Grafiktreiber die passenden Funktion nicht unterstützt, wird eine einfache, nicht beschleunigte Standardfunktion verwendet.

Tabos verlangt von den einzelnen Grafiktreibern nur **3** Grundfunktionen:

1. `setMode`, die den gewünschten Modus setzt
2. `setpixel`, um einen Pixel auf den Bildschirm zu setzen
3. `getpixel`, um einen Pixel vom Bildschirm zu lesen

3. Booten des Systems

Es gibt verschiedene Möglichkeiten um Tabos zu starten, die im Folgenden genauer beschrieben werden:

3.1 Die verschiedenen Medien

Im Moment gibt es drei verschiedene Methoden um Tabos zum Laufen zu bringen:

- mit einer Diskette
- mit einer Festplatte
- mit einer CD-ROM

3.1.1 Booten mit einer Diskette

Um Tabos mit einer Diskette zum Laufen zu bringen, muss man lediglich eine leere Diskette besitzen und diese mit Hilfe der Dokumentation aus den Quellen erzeugen. Hier folgt eine Zusammenfassung:

1. Man entpackt das Archiv mit dem Packprogramm tar:
`tar xvzf tabos-develX.tar.gz`
2. Im Unterverzeichnis `tools/` dieses Ordners ruft man das Programm `make_disc` auf:
`./make_disc`
3. Nun übersetzt man den Kernel und die notwendigen Treiber:
`make && make modules install modules_install`

011/14

Im Folgenden wird nun eine Diskette mit dem Verzeichnislayout erstellt und alle notwendigen Konfigurationsdateien in die dafür vorgesehenen Verzeichnissen kopiert.

3.1.2 Booten mit einer Festplatte

Der Weg erfolgt genauso wie bei einer Diskette, nur muss man das Boot-Gerät auf die gewünschte Partition einstellen. Dazu ändert man in der Datei `./make_disc` den Laufwerksbuchstaben auf die gewünschte Festplatte bzw. Festplattenpartition.

3.1.3 Booten mit einer CD-ROM

Benutzen Sie hierfür ein Brenn-Programm, welches in der Lage ist bootfähige CD-ROMs zu erstellen. Erstellen Sie eine Bootdiskette von Tabos (3.1.1.) und erstellen Sie von diesen ein Image, z.b. mit dem Programm `dd`:

`dd if=/dev/floppy of=tabos.img.`

Dieses können Sie dann mit dem Brenn-Programm auf die CD-ROM schreiben.

3.2 Boot Parameter

Vor dem Start von Tabos haben Sie im Bootmenü von GRUB die Möglichkeit, Tabos einige Boot Parameter mitzuteilen, die den Bootvorgang von Tabos beeinflussen. Alternativ können Sie auch die Datei `menu.lst` im Verzeichnis `boot/grub/` der Bootdiskette ändern.

Bisher gibt es zwei Boot Parameter:

`video="mode"` : Hiermit können Sie den gewünschten Videomodus "modus" eingeben, mit dem Tabos gestartet werden soll.

`root="root device"` : Mit diesem Parameter können Sie den Datenträger, von dem geladen werden soll, ändern. Normalerweise wird der Datenträger selbständig erkannt und benutzt.

3.3 Der Boot Prozess

Sobald der Bootloader GRUB den Kernel in den Speicher geladen hat, wird der Kernel ausgeführt. Im Folgenden setzt der Kernel notwendige Protected Mode Optionen, damit der Computer entsprechend diesen Richtlinien arbeiten kann. Nachdem alle weiteren wichtigen Bereiche initialisiert wurden (MM, VFS, DevFS, Tasksystem), beginnt der Init-Thread seine Arbeit. Er mountet das Bootmedium als Root-Verzeichnis (`/`) und lädt das deutsche Tastaturlayout. Im nächsten Schritt lädt der Init-Thread die Datei `/etc/mod.cnf` und integriert die dort aufgeführten Module in das System. Als letzten Schritt startet die Shell „Trash“ auf der ersten Konsole und diese wartet auf die Eingaben des Benutzers.

Wie benutzt man Tabos?

Sobald man Tabos gebootet hat, befindet man sich innerhalb der Shell „Trash“ am Kommandoprompt `./#`. Dieser Prompt gibt das aktuelle Verzeichnis an, in diesem Fall ist es das Root-Verzeichnis (`/`). Zu den wichtigsten Programmen aus der „Tabos Application Suite“ gehören:

- **`cat`** – zeigt eine Datei an. Verwendung: `./cat [dateiname]`
- **`cd`** – wechselt das aktuelle Verzeichnis. Verwendung: `./cd [verzeichnis]`
- **`cp`** – kopiert eine Datei in einen anders Verzeichnis. Verwendung: `./cp [datei1]`

[datei2]“

- **kill** – beendet einen aktiven Prozess: Verwendung: „**kill [pid]**“. Die PID bekommt man z.B. über das Programm ps
- **ls** – zeigt das aktuelle Verzeichnis an. Verwendung: „**ls**“, optional ein [verzeichnis]
- **mkdir** – erstellt ein neues Verzeichnis. Verwendung: „**mkdir [verzeichnis]**“
- **mv** – bewegt eine Datei an einen anderen Ort. Verwendung: „**mv [datei1] [datei2]**“
- **pwd** – zeigt den aktuellen Verzeichnisnamen an. Verwendung: „**pwd**“
- **rm** – löscht eine vorhandene Datei. Verwendung: „**rm [datei]**“
- **rmdir** – löscht ein vorhandenes Verzeichnis. „**rmdir [verzeichnis]**“
- **touch** – erstellt eine neue leere Datei. Verwendung: „**touch [datei]**“
- **cdplayer** – CD Player. Verwendung: „**cdplayer [track]**“
- **fdisk** – Festplatten-Partitionierungs Programm. Verwendung: „**fdisk [fesplatte]**“
- **fortune** – Der Sprücheklopfer. Verwendung: „**fortune**“
- **hm** – Hangman-Spiel. Verwendung: „**hm**“
- **ttt** – TicTacToe-Spiel. Verwendung: „**ttt**“
- **mkfmt** – formatiert eine Diskette. Verwendung: „**mkfmt [diskette]**“
- **insmod** – fügt ein Modul in das System ein. Verwendung: „**insmod [module]**“
- **rmmod** – entfernt ein Module aus dem System. Verwendung: „**rmmod [module]**“
- **mount** – mountet ein bestimmtes Gerät ins VFS. Verwendung: „**mount [gerät] [verzeichnis] [fstype]**“
- **ps** – zeigt alle laufenden Prozesse an. Verwendung: „**ps**“
- **top** – zeigt fortlaufend alle laufenden Prozesse + MM Informationen. Verwendung: „**top**“
- **umount** – entfernt ein Gerät aus dem VFS. Verwendung: „**umount [verzeichnis]**“
- **w8378** – Systemmonitor für WinBond8378 Chipsätze. Verwendung: „**w8378**“
- **wavplay** – Wav-Player. Verwendung: „**wavplay [wavdatei]**“

5. Das Verzeichnis Layout

```

|-apps
|-boot
| |-grub
|-dev
|-etc
| |-init
|-home
|-lib
|-mnt
|-system
| |-keymaps
| |-modules

```

In Zukunft werden alle Programme unter dem Verzeichnis /apps in den jeweiligen Unterverzeichnissen gespeichert. Die Verzeichnisse /system und /etc sind dabei Systemverzeichnisse, in denen nur systemrelevante Daten gespeichert werden. /boot enthält lediglich Bootanweisungen für den Systemloader GRUB, damit Tabos gestartet werden kann. Jeder registrierte Benutzer bekommt ein eigenes Unterverzeichniss im Verzeichnis /home, in dem er die Möglichkeit hat seine Daten

013/14

zu sichern. Das Verzeichnis /mnt wird hier, wie vorher schon erwähnt, für weitere Datenträger benutzt.

6. Kompilierung des Systems

Um das System vom Quellcode in einen ausführbaren Code umzuwandeln, benötigt man ein funktionierendes Linux oder ähnliches System.

Notwendige Programme:

- gcc (GNU C Compiler)
- make (make Programm)
- ld (Linker)
- mtools (FAT Dateisystem Tools)

7. Literaturverzeichnis

- [1] Rüdiger Brause
Betriebssysteme, Grundlagen und Konzepte
2. überarbeitete Auflage Springer-Verlag, 1998-2001
- [2] Christian Maurer
Grundzüge der nichtsequentiellen Programmierung
Springer-Verlag, 1999
- [3] www.nondot.org/sabre/os/articles
- [4] www.intel.com/design/pro/manuals