Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

Das Basekernel OS

1. Kernel
2. Anwenderprogrammme
3. Standardbibliothek

Kernel

Der Kernel des Basesystem OS hat eine klassische monolithische Kernelarchitektur. Wesentliche Bestandteile der **innersten ersten Schicht** und Basisdienste sind:

1. Kernel Startup (aka. "main" Funktion)
   - Assembler `kernelcore.S`
   - C `main.c`
2. Speichermanagement
   - `page.c` (VMM)
   - `pagetable.c` (VMM)
   - `kmalloc.c`
3. Prozessmanagement
   - `process.c` (Programmausführung)
   - `elf.c` (Programmformat)
4. Systemaufruf Schnittstelle (für Anwendungsprogramme)
   - `syscall_handler.c`
5. Ereignismanagement
   - `event.c`
   - `interrupt.c`

Kernel

Weitere Bestandteile der **zweiten Schicht** und Basisdienste sind:

6. Gerätemanagement
   - `device.c` (Gerätetreiberschnittstelle)
   - `pipe.c` (Queues zwischen Prozessen)
7. Dateisystem
   - `diskfs.c`
   - `fs.c` (allgemeine Dateisystemschnittstelle)
   - `cdromfs.c`
8. (Block) Cache
   - `bcache.c`
9. Ein- und Ausgabe (eher minimale Standardblibliothek für den Kernel)
   - `console.c`
   - `graphics.c'
   - `printf.c`
   - `string.c`

Kernel

Weitere Bestandteile der **nullten Schicht** sind hardwarenahe Komponenten:

10. Gerätetreiber
    - `console.c`
    - `graphics.c`
    - `pic.c` (Interruptcontroller)
    - `ata.c` (Parallele Geräteschnittstelle für Blockgeräte, HDD, CDROM)
    - `clock.c` (Timer)
    - `keyboard.c` (Tastatur)
    - `serical.c` (Serielle UART Schnittstelle)
    - `rtc.c` (Real Time Clock ⇒ Systemuhr)
11. Datenstrukturen
    - `list.c`
    - `hash_set.c`

Kernel

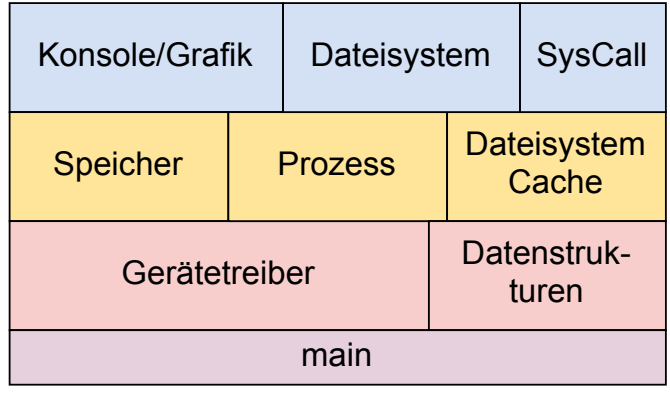

Abb. 1. Schichtenaufbau des monolithischen Basekernel

Standardbibliothek

Die Standardbibliothek implementiert häufig vorkommende Funktionen für:

- Ein- und Ausgabe
- Dateioperationen
- Speichermanagement (oder eine Schnittstelle dazu)
- Systemaufrufe

Sie wird von allen Nutzerprogrammen (user space) verwendet. Funktionen für Netzwerkzugriff (Sockets) und vieles mehr kann noch enthalten sein.

Standardbibliothek

1. Systemaufrufschnittstelle
   - C `syscalls.s`
   - Assembler `syscall.S`
2. Speichermanagement (USer space)
   - `malloc.c`
3. Ein- und Ausgabe
   - `stdio.c`
4. Sonstiges
   - `user_start-c` (Start eines Benutzerprogramms)
   - `errno.c` (Fehlerbehandlung)

Standardbibliothek

5. Zeichenketten
    - `string.c`
6. Allgemeine Funktionen
    - `stdlib.c` (**Programmausführung**, Zeit, Systeminformation usw.)
7. Konsole und Grafiksystem (Fenster)
    - `nwindow.c`

Benutzerpogramme

Im wesentlichen gibt es nur eine Shell. Die Aufgabe einer Shell ist:

1. Interaktive eingegebene Befehle von einer Kommandozeile unmittelbar ausführen ⇒ **Nutzerbetrieb**
2. Shellskripte (kleine Programme als Sammlung von Shellbefehlen) ausführen ⇒ **Stapelbetrieb**

Dabei sollen zwei primäre Schnittstellen zum Betriebssystem auf programmatische und interaktive Weise zur Verfügung gestellt werden (Operationalität):

A. Dateisystem, Textdatein

B. Prozesse und Programme

C. Ein-und Ausgabe (Terminal)

Shell

Die meisten gängigen Unix-Shells bieten die folgenden Funktionen:

[Wikipedia]

- Starten von Kommandos
- Dateinamen-Wildcards (globs) als Kommandoargumente
- Bedingungen (if, case) und Schleifen (while, for)
- interne Kommandos (cd, read)
- interne Variablen ($HOME)
- Manipulation der Umgebungsvariablen für die neuen Prozesse
- Ein-/Ausgabeumlenkung
- Starten mehrerer Prozesse, Verkettung über Pipes
- Starten von Prozessen im Hintergrund

Shell

Moderne Shells können darüber hinaus:

- Vervollständigung von Kommandos, Dateinamen und Variablen (completion system)
- Editieren der Kommandozeile (command line editing)
- Wiederholung und Editieren früherer Kommandos (command history)
- Stoppen und erneutes Starten von Prozessen (job control)
- Verschieben von Prozessen aus dem Vordergrund in den Hintergrund und umgekehrt (job control)
- Eingebautes Kommando zur Durchführung von Berechnungen ($((2+2)))
- Eingebautes Kommando zum Testen von Dateieigenschaften (test)

Shell

Skripte

- Shellskripte sind Text-Dateien und stellen kleine Programme dar.
  - Sie werden vom Anwender geschrieben und beim Aufruf von der Shell gelesen und ausgeführt.
  - Muss man z. B. immer wieder in seinem Arbeitsalltag fünf Befehle nacheinander in die Shell eintippen, kann man das vereinfachen, indem man diese Befehle in einem Skript sammelt und dann nur noch das ganze Skript aufruft.
- Die Shell liest das Skript aus und führt die Befehle entsprechend aus. Zu beachten ist, dass man alle Befehle, die man „händisch“ in die Shell eingibt, auch über ein Skript ausführen kann und umgekehrt.

Dienstprogramme

0. Shell
   - `shell.c` (sehr einfacher Shell Intepreter, kein Skripting)
1. Prozess Manager
   - Nur Prozessinformation durch `procstat.c`
   - Steuerung von Prozessausführung (stoppen) gibt es hier nicht
2. Systeminformationen
   - `systats.c`
3. Dateisystemedienste
   - `livestat.c` (Information über Gerätetreiber/Gerät und Cache)
   - `copy.c` (Datei kopieren)
4. Fenstermanager
   - `manager.c` (Grafiksystem, Verwaltung von Fenstern)

Der Erzeugungsprozess

Es müssen folgende Komponenten von Quellcode (C/Assembler) in binäre Programmdaten übersetzt werden:

1. Der Kernel (`kernel`) und Bootloader (`bootblock`)
2. Die Standardbibliothek (`baselib.a`)
   - Hier nur als Archiv (a), nicht als geteilte (shared) Bibliothek (dll/so) `baselib.a`
   - Alle Programme werden statisch (d.h. vollständig enthaltend) kompiliert und erzeugt.
3. Die Nutzerprogramme

Für den gesamtn Übersetzungsprozess wird ein GCC Kompiler verwendet. Dieser enthält neben dem Präprozessor *cpp*, den eigentlichen Kompiler *cc* noch einen Assembler *as* und einen Linker *ld*. Weitere Hilfsprogramme vor allem für das ELF Format werden benutzt (*objdump*, *objcopy*, *ar*).

Der Erzeugungsprozess

Es muss GCC mit 32 Bit x86 ELF (Executable Link Format) verwendet werden. Andere Prrogrammformate und Prozessorarchitekturen führen zu einem nicht funktionieren Basekernl OS (sowohl Kernel als auch Nutzerprogramme).

- Der gesamte Übersetzungsprozess wird mit dem *make* Programm gesteuert.

make (englisch für machen, erstellen) ist ein Build-Management-Tool, das Kommandos in Abhängigkeit von Bedingungen ausführt. Es wird hauptsächlich bei der Softwareentwicklung als Programmierwerkzeug eingesetzt.

Der Erzeugungsprozess

- *make* liest eine teils prozedurale und teils deklarative Anweisungsdatei (Makefile), in dem die Abhängigkeiten des Übersetzungsprozesses von Programmen formalisiert erfasst sind.
- Diese Formalisierung beschreibt, welche Quelltextdateien auf welche Weise durch den Compiler oder durch andere Programme zu welchen Objektdateien bzw. Ergebnissen verarbeitet werden, bzw. welche Objektdateien vom Linker zu Programmbibliotheken oder ausführbaren Programmen verbunden werden.
- Alle Schritte erfolgen unter Beachtung der im Makefile erfassten Abhängigkeiten.[2]
- Wenn das Makefile vom make-Programm abgearbeitet wird, wird eine Umwandlung etwa einer Quelldatei in eine Objektdatei nur dann vorgenommen, wenn die Quelldatei **neuer** als die bereits vorliegende Version der Ergebnisdatei ist, bzw. wenn keine Ergebnisdatei vorhanden ist.
- Bei der Entwicklung großer Programmpakete mit vielen Quelldateien und vielen ausführbaren Programmen entfällt so die Notwendigkeit, bei einer Reihe kleiner Veränderungen alle Kompilierungen erneut durchzuführen.

Makefile (main)

```
include Makefile.config

LIBRARY_SOURCES=$(wildcard library/*.c)
LIBRARY_HEADERS=$(wildcard library/*.h)
USER_SOURCES=$(wildcard user/*.c)
USER_PROGRAMS=$(USER_SOURCES:c=exe)
KERNEL_SOURCES=$(wildcard kernel/*.[chS])
WORDS=/usr/share/dict/words

.PHONY: build-kernel build-library build-userspace build-cdrom-image

all: build-cdrom-image

build-kernel: kernel/basekernel.img

build-library: library/baselib.a

build-userspace: $(USER_PROGRAMS)

build-cdrom-image: basekernel.iso

kernel/basekernel.img: $(KERNEL_SOURCES) $(LIBRARY_HEADERS)
    cd kernel && make

library/baselib.a: $(LIBRARY_SOURCES) $(LIBRARY_HEADERS)
```

Makefile (config)

```
KERNEL_CCFLAGS=-Wall -c -ffreestanding -fno-pie -g -std=gnu99

# These settings select the native compiler,
# which is likely to work on native linux-x86.
#
CC=gcc -m32
LD=ld -melf_i386
#LD=ld -arch i386
AR=ar
OBJCOPY=objcopy -R .note.gnu.property
ISOGEN=genisoimage

# If you are compiling from another platform,
# then use the script build-cross-compiler.sh
# add cross/bin to your path, and uncomment these lines:
#CC=i686-elf-gcc
#LD=i686-elf-ld
#AR=i686-elf-ar
#OBJCOPY=i686-elf-objcopy

# If building on OSX, then install mkisofs via ports or brew
# and uncomment this:
#ISOGEN=mkisofs
```

Makefile (kernel)

```
include ../Makefile.config

KERNEL_OBJECTS=kernelcore.o main.o console.o page.o keyboard.o mouse.o event_queue.o clock.o interrupt

basekernel.img: bootblock kernel
    cat bootblock kernel /dev/zero | head -c 1474560 > basekernel.img

kernel: kernel.elf
    ${OBJCOPY} -O binary $< $@

bootblock: bootblock.elf
    ${OBJCOPY} -O binary $< $@

kernel.elf: ${KERNEL_OBJECTS}
    ${LD} ${KERNEL_LDFLAGS} -Ttext 0x10000 ${KERNEL_OBJECTS} -o $@

bootblock.elf: bootblock.o
    ${LD} ${KERNEL_LDFLAGS} -Ttext 0 $< -o $@

%.o: %.c
    ${CC} ${KERNEL_CCFLAGS} -I ../include $< -o $@

%.o: %.S
    ${CC} ${KERNEL_CCFLAGS} -I ../include $< -o $@
```

make it all

```
cd kernel && make
make[1]: Entering directory '/home/sbosse/proj/vm42/src/basekernel/kernel'
gcc -m32 -Wall -c -ffreestanding -fno-pie -g -std=gnu99 -I ../include bootblock.S -o bootblock.o
ld -melf_i386  -Ttext 0 bootblock.o -o bootblock.elf
objcopy -R .note.gnu.property -O binary bootblock.elf bootblock
gcc -m32 -Wall -c -ffreestanding -fno-pie -g -std=gnu99 -I ../include kernelcore.S -o kernelcore.o
gcc -m32 -Wall -c -ffreestanding -fno-pie -g -std=gnu99 -I ../include main.c -o main.o
gcc -m32 -Wall -c -ffreestanding -fno-pie -g -std=gnu99 -I ../include console.c -o console.o
gcc -m32 -Wall -c -ffreestanding -fno-pie -g -std=gnu99 -I ../include page.c -o page.o
gcc -m32 -Wall -c -ffreestanding -fno-pie -g -std=gnu99 -I ../include keyboard.c -o keyboard.o
In file included from keyboard.c:42:
keymap.de.pc.c:5:7: warning: multi-character character constant [-Wmultichar]
 {'3', '§', '3', 0},
       ^~~~
keymap.de.pc.c:5:7: warning: overflow in conversion from ‘int’ to ‘char’ changes value from ‘49831’ to
keymap.de.pc.c:13:2: warning: multi-character character constant [-Wmultichar]
 {'ß', '?', 'ß','\\'},
  ^~~~
keymap.de.pc.c:13:2: warning: overflow in conversion from ‘int’ to ‘char’ changes value from ‘50079’ t
keymap.de.pc.c:13:13: warning: multi-character character constant [-Wmultichar]
 {'ß', '?', 'ß','\\'},
             ^~~~
keymap.de.pc.c:13:13: warning: overflow in conversion from ‘int’ to ‘char’ changes value from ‘50079’
keymap.de.pc.c:14:2: warning: multi-character character constant [-Wmultichar]
 {'´', '`', '´',0},
  ^~~~
keymap.de.pc.c:14:2: warning: overflow in conversion from ‘int’ to ‘char’ changes value from ‘49844’ t
keymap.de.pc.c:14:13: warning: multi-character character constant [-Wmultichar]
 {'´', '`', '´',0},
             ^~~~
```

TODO