Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Entwicklung eines Mikrokernelsystems zur Unterstützung der Lehre

Leonard König
Matrikelnummer: 4963906
leonard.koenig@fu-berlin.de

Eingereicht bei: Barry Linnert
Zweitgutachter: Prof. Dr.-Ing. Jochen Schiller

Berlin, 13. Februar 2019

Diese Arbeit beschäftigt sich mit der Konzipierung eines für die Lehre geeigneten Betriebssystemkernels. In diesem Kontext liegt der Fokus nicht auf breite Einsetzbarkeit des entwickelten Systems auf echten Rechnern, sondern auf die Einbettung in das vorhandene didaktische Konzept des Kurses „Betriebssysteme“. Diese anderen Anforderungen und der eingeschränkte Use-Case erzwingen aber auch ermöglichen einige radikale Designentscheidungen die in dieser Arbeit behandelt und erläutert werden. Darüber hinaus wird die im Kurs eingesetzte Virtualisierungslösung QEMU auf einen aktuellen Stand gebracht.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

13. Februar 2019

Leonard König

Inhaltsverzeichnis

1. Einführung **7**
1.1. Aufbau des Kurses „Betriebssysteme“ 7
1.2. Bisherige Arbeiten und Abgrenzung 10

2. Entwurf **12**
2.1. Struktur 12
2.1.1. Build-System 13
2.2. Hardware-Abstraction-Layer 16
2.2.1. Hardware-zu-Software-Interface 18
2.2.2. Software-zu-Hardware-Interface 19
2.3. System 20
2.4. QEMU 22

3. Evaluation und Fazit **23**
3.1. Ausblick 24

A. Begrifflichkeiten **25**

1. Einführung

„No. 1: The Larch. *The Larch. THE. LARCH*“ (Monty Python, Flying Circus S1E3 [24])

In dieser Arbeit werden zwei voneinander getrennte, aber ineinandergreifende, Projekte behandelt, wobei beide zur Unterstützung der Lehre im Kurs „Betriebssysteme“ [19] an der Freien Universität dienen sollen. Der besondere Fokus, der sich durch Beide ziehen soll, ist hierbei ein didaktischer: Ziel ist die Erstellung eines qualitativ für die Lehre geeigneten Systems. Dazu gehören geringer Pflegeaufwand der Codebasis, aussagekräftige Dokumentation, einfache Verständlichkeit und Anwendung genereller „best practices“ bei der Entwicklung.

Zu Beginn möchte ich in das Thema einleiten, indem ich motivierend das Szenario des zu unterstützenden Kurs schildere. Danach gehe ich detaillierter auf die eigentliche Problematik ein, welche dann in die umgesetzten Lösungen in den Projekten überleitet. Abschließend wird evaluiert ob das Ziel erreicht werden konnte und ein Ausblick auf weitere Änderungen und Verbesserungen gegeben. Ich werde hierbei das Verständnis einiger Begriffe voraussetzen, und verweise bei Unklarheiten auf die Begriffserläuterungen im Appendix.

1.1. Aufbau des Kurses „Betriebssysteme“

Um die Anforderungen zu verstehen, ist es sinnvoll den Aufbau des Kurses zu kennen. Innerhalb dieser Veranstaltung des Masterstudiengangs Informatik sollen die grundlegenden Konzepte des Design von Betriebssystemen vermittelt werden. Hierzu gibt es einen theoretischen Teil mit einer Vorlesung und ein praktisches Seminar, in dessen Rahmen die Studierenden auf der – bereits schlicht vorhandenen – Entwicklungsplattform PORTUX-920T einen eigenen Betriebssystemkernel entwickeln sollen.

Um an die eigentliche Entwicklung heranzuführen, sind sieben Übungsaufgaben auf das Semester aufgeteilt zu bearbeiten. Diese sind hierbei zwar in sich abgeschlossen und behandeln immer ein konkretes Thema, bauen aber durchaus aufeinander auf. Für den Fall, dass es bei einem Übungszettel Probleme gab wird nach jeder Aufgabe eine Beispiellösung zur Verfügung gestellt, an Hand derer die Teilnehmenden entweder ihre eigene Lösung verbessern können oder diese direkt als Basis für die nächste Übung nutzen dürfen. Im ersten Teil dieser Arbeit wird genau so eine Implementierung vorgestellt, jedoch mit dem Fokus auf Eignung im Einsatz in der Lehre, weniger auf Effizienz oder Sicherheit.

Eine Übersicht der einzelnen Aufgabenabschnitte die bearbeitet werden sollen sei hier gegeben:

1. Serielle Schnittstelle: Um einfache textuelle Ein- und Ausgabe zu ermöglichen, soll ein simpler Treiber für die serielle Schnittstelle der Debug Unit „DBGU“ geschrieben werden.

   Hierbei soll sowohl mit der Hardware bekannt gemacht (und hoffentlich die Angst vor dieser genommen) als auch eine Grundlage geschaffen werden, um Fehlerausgaben zu ermöglichen, die die weitere Entwicklung sehr vereinfachen.

2. Ausnahmebehandlung: Ein ARM-Prozessor wie der im PORTUX eingesetzte bietet Ausnahmebehandlung über eine Interrupt-Vector-Table (IVT). Ausnahmen sind in diesem Zusammenhang bspw. Zugriff auf nicht vorhandene Daten, oder unbekannte

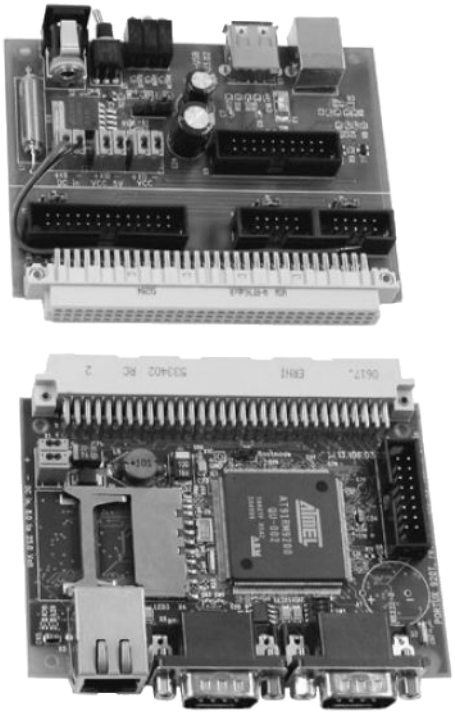

Abbildung 1: Portux 920T [31] – „The machine that goes ‚ping'" [25]

Befehle. Diese Ausnahmen sollen abgefangen und sinnvolle Fehler auf der DBGU ausgegeben werden.

In dieser Aufgabe werden damit Kenntnisse über die eingesetzte Technologie vertieft und ein, später sehr wichtiges, Instrument der Hardware/Software-Schnittstelle angebunden.

3. Externe Interrupts: Ebenfalls über die Ausnahmebehandlung werden externe Interrupts behandelt. Diese werden durch Peripherie-Geräte ausgelöst um eine Unterbrechung der normalen Ausführung anzufordern. Ein solches Gerät ist der System Timer (ST), welcher in regelmäßigen Abständen eine Unterbrechung auslöst. Damit können bestimmte Aufgaben wiederkehrend ausgeführt werden. Die Aufgabe ist, dies eben so zu konfigurieren und die ankommenden Unterbrechungen auf der Konsole auszugeben um die Funktionsweise zu demonstrieren.

   Hierzu muss – aufbauend auf der vorherigen Aufgabe – die IVT erweitert und sonst lediglich bereits bekanntes Wissen über Treiberprogrammierung angewandt werden. Sobald die Möglichkeit besteht, auf externe Ereignisse zu reagieren und damit auch zusätzlich zeitabhängige Aktionen auszuführen, ist die Grundlage geschaffen, um sich in der Abstraktionshierarchie nach oben zu arbeiten.

4. Multitasking: Bisher kann das Betriebssystem nur Aufgaben hintereinander ausführen. Ziel ist nun, mehrere Aufgaben quasi-parallel zu erledigen, durch Nutzung des STs, mit dessen Hilfe in regelmäßigen Zeitabständen zwischen den Aufgaben gewechselt wird. Hierbei ist die Vorarbeit aus den vorherigen Aufgaben essentiell.

Der sog. Scheduler soll regelmäßig aufgerufen werden, dieser die aktuelle Task unterbrechen um die nächste abarbeiten zu lassen. Hiermit wird eine Abstraktionsebene geschaffen und sich langsam von der physischen Ressource CPU (bzw. verfügbare CPU-Rechenzeit) gelöst und diese in Zeitscheiben aufgeteilt.

5. System-Aufrufe: Um dem Betriebssystem Funktionalität hinzuzufügen muss noch den Kern anpasst werden. Ziel dieser Übung ist es eine Schnittstelle zu definieren, sodass abstraktere Dienste außerhalb des Kerns laufen können. Solche Anwendungen müssen aber trotzdem auf die vom Kern verwalteten Ressourcen zugreifen. Dies soll nun über besagte Schnittstelle passieren. Diese Systemaufrufe werden über sog. Software-Interrupts umgesetzt, welche über einen dafür vorgesehenen Befehl im Prozessor ausgelöst werden. Wird die Ausführung damit unterbrochen, kann das Betriebssystem die Anfrage bearbeiten.

   In dieser Aufgabe soll nun eine logische Trennung zwischen Kern und Anwendung definiert werden, dafür benötigen wir also eine Systemaufrufschnittstelle. Das ist die Schnittstelle, die unterschiedliche Kernel aus Anwendungssicht unterscheidet. Dieses Interface soll einerseits viel der Funktionalität des Kernels abstrahieren, andererseits definiert es die Anwendungsseite der Kommunikation mit dem Kernel und somit auch durchaus das Design innerhalb unseres Systems.

6. Virtueller Adressraum: Bisher liegen Kern und Anwendung trotz ihrer logischen Trennung durch den letzten Zettel im gleichen Adressraum. Somit könnte theoretisch eine Anwendung auch „direkt“ auf das System zugreifen, in dem es schlicht die dem Kern zugewiesenen Teile des Arbeitsspeichers liest und schreibt. Um diese Zugriffe auf Speicher an Adressen die dem Kern zugeteilt sind zu unterbinden – und aus vielen anderen Gründen – implementieren wir virtuelle Adressen. Hierbei wird auf den Speicher nicht mehr direkt über die physikalischen Adressen zugegriffen, sondern über virtuelle (auch logische) Adressen. In einer weiteren Komponente des Systems, der Memory-Management-Unit (MMU), liegt eine Tabelle vor, die diese virtuellen Adressen in physikalische Adressen übersetzt. Dbaei können für unterschiedliche Bereiche (bspw. Kern und Anwendung) verschiedene Tabellen angelegt werden, sodass Zugriffe auf Kern-Speicher aus der Anwendungen heraus ins Leere laufen und eine entsprechende Ausnahme auslösen, wie in Aufgabe 2 erwähnt.

7. Prozesse: Zwar haben ist bereits der Kernel vor Zugriff durch die Anwendungen geschützt, eine Abschottung zwischen den einzelnen laufenden Threads gibt es aber noch nicht, sie haben bisher nur alle ihre eigenen Stacks. Nun sollen echte Prozesse mit eigenen Adressräumen implementiert werden.

   Mit dieser Aufgabe wird der Kurs abgeschlossen. Das System setzt – auf grundlegenster Ebene – eine Trennung von Kernel und Anwendung um, indem es in ersterem die physischen Hardware-Ressourcen wegabstrahiert um sie als logische Ressourcen zu verwalten, die die Anwendungen wiederum anfragen können.

Ziel ist es, den Teilnehmenden mit diesen Aufgaben vor Augen zu führen, welche Abwägungen gemacht und wo Kompromisse getroffen werden müssen. Aus theoretischer Sicht sollten alle diese Aufgabenbereiche klar und der Sinn und Zweck der Techniken bereits bekannt sein. Was es jedoch tatsächlich bedeutet bspw. ein Syscall-Interface zu schreiben und welche Gedanken man sich dabei machen muss (oder sollte), ist mit

dem Bereich der Softwaretechnik verknüpft. In diesen Zusammenhang gehört auch der wohl größte Streit bei der Konzeptionierung eines Betriebssystems: Die Debatte über Microkernel vs. monolithische Kernel, welche genau an dieser Stelle einhakt und deren enorme Relevanz innerhalb der Aufgaben klar werden sollte. Somit wird natürlich auch diese Problematik in dieser Arbeit aufgegriffen werden müssen.

Jedoch steht leider für den Kurs an sich die Entwickungsplattform nicht in ausreichender Menge zur Verfügung. Stattdessen wird die Virtualisierungssoftware QEMU eingesetzt; sie löst von der Notwendigkeit die Hardware in hinreichender Menge zur Verfügung stellen zu müssen und bietet zudem eine sehr bequeme Art der Entwicklung. Vorhanden ist eine für den Kurs angepasste Version von QEMU in einer älteren Version, deren Aktualisierung und Instandsetzung ebenfalls in dieser Arbeit übernommen wurde.

1.2. Bisherige Arbeiten und Abgrenzung

„UNIX is simple and coherent.“ (Dennis Ritchie [27])
„GNU's Not UNIX.“ (Richard Stallman [28])

Meine Arbeit reiht sich mit ihrem lehrenden Anspruch dabei in eine Tradition ein, die von John Lions mit dem populären „Lions' Commentary on UNIX 6th Edition, with Source Code“ [20] eingeleitet wurde. Dieses Buch hat auf der Grundlage des für Lehre freigegebenen Quellcodes von UNIX V6 die fundamentalen Konzepte von Betriebssystemdesign und Implementierung behandelt. Auch wenn das Betriebssystem an sich nicht primär für die Lehre entwickelt wurde, hat es sich durch die Qualität, die Konsistenz und schlicht auch die Verfügbarkeit angeboten, basierend darauf zu lehren. Mit UNIX V7 wurde jedoch die akademische Nutzung durch Lizenznehmer nicht mehr gestattet, UNIX-ähnliche Systeme waren aber nicht mehr aus dem akademischen Lehrgebrauch herauszudenken.

In der Zeit darauf wurde durch Andrew S. Tanenbaum mit MINIX [29] ein System eigens nur für die Lehre geschaffen, im Gegensatz zu UNIX jedoch mit einem Mikrokernel-Design und wieder begleitet mit einem „klassischen“ Buch, an Stelle eines mit Erläuterungen versehenem Quellcodes.

Mit xv6 [7] wurde vom MIT ein UNIX V6 Klon für x86 gebaut, zudem es – ganz im Stile von Lions – eine kommentierte Version mit Querreferenzen gibt.

Alle diese Systeme sind jedoch General-Purpose-Betriebssysteme und setzen schlussendlich viele Kenntnisse voraus, die in dem Kurs „Betriebssysteme“ nicht gegeben sind. Für viele der Teilnehmenden ist das Schreiben eines Kernels das erste größere richtige Projekt, indem *alles* von Hand gemacht werden muss – erst recht das erste mal in C. Während es also sehr viel Literatur gibt zu dem rein theoretischen Design von Betriebssystemen, sind die Anforderungen hier trotzdem andere.

In dem Kurs sollen die Studierenden ihr erstes eigenes Betriebssystem schreiben, da sind die Probleme oft viel grundlegender: Wie schreibt man Makefiles? Wie funktioniert eigentlich ein C-Compiler mit Präprozessor, eigentlichem Compiler, Assembler und Linker? Und wie greift man auf Memory-Mapped-I/O zu? Die Kenntnisse die man voraussetzen kann sind größtenteils theoretischer Natur – und hier soll diese Arbeit ansetzen.

In dem Programmcode, des in diesem Zusammenhang entwickeltem Betriebssystems, sollen also die gewählten Lösungsstrategien klar erkennbar sein. Zudem sollte es als Vorlage dienen können, um die Teilnehmenden bei ihren Übungen auch aus praktischer Sicht zu unterstützen.

Was sind nun aber die Anforderungen an ein System, dessen Programmdesign für den „Lesenden“ schnell greifbar ist? Hierzu gibt es einige Punkte zu beachten:

1. Geringe Komplexität,
2. Eindeutige Trennung von unabhängigen Komponenten und
3. klare Schnittstellen zwischen diesen.

Traditionell steht der Punkt 1 im Gegensatz zu den anderen, da zwar eine sinnvolle Trennung für ein aufgeräumteres System sorgt, aber auch für mehr Aufwand bei der Kommunikation und Kooperation, bzw. überhaupt die Notwendigkeit von Informationsaustausch. Es handelt sich an dieser Stelle somit um eine gewisse Gratwanderung, wie viel Trennung ist sinnvoll – und wie viel schon zu viel?

Am Ende soll das entstandene Betriebssystem genau so wie die bisherige Lösung eingesetzt werden können, im Spezifischen aber als Beispiel für gelungenes Kernel-Design stehen. In besonderem Maße spiegelt sich das in der Anforderung wieder, dass die unterschiedlichen Lösungen der Aufgaben, genauso wie die Aufgaben selbst, aufeinander aufbauen sollen – sprich im Verlauf möglichst nur ergänzender Programmcode geschrieben, Schnittstellen nicht verändert und Logik nicht überarbeitet werden. Gleichzeitig soll das System minimal und nicht unnötig komplex sein, damit es für die Teilnehmenden als Basis für weitere Bearbeitung der Aufgaben dienen kann. Das Ergebnis soll auch zum Lesen der Arbeit und des Programmcodes einladen, um anhand dessen den Lesenden durch die ersten Entwicklungsstufen eines Betriebssystemkernels zu leiten, dabei dem Projekt zu unterstüzen und die Panik zu nehmen.

2. Entwurf

„Plans are useless, but planning is essential“ (Dwight D. Eisenhower [8])

Eine der ersten Fragen, die man sich als Entwickler stellt, wenn man die Aufgabe bekommt eine Software zu entwickeln, ist nach den Werkzeugen, wie Programmiersprache, Build-System, etc. Optimalerweise schaut man sich hierzu einerseits bestehende Projekte und deren Begründung bei ihren Entscheidungen an, und vergleicht die eigenen Anforderungen mit diesen Projekten. In diesem Fall ist die Wahl jedoch recht einfach. Seit UNIX in Version 4 in C geschrieben wurde [10], hat es viele Betriebssysteme gegeben, die diesem Beispiel gefolgt sind. Neben Linux, aber auch großen Teilen des Windows NT Kernels [17], den anderen UNIX-Systemen wie macOS/Darwin, den BSD-Distributionen oder Solaris sind auch „neuere“ Betriebssysteme – darunter RiotOS, FreeRTOS, etc. – zu vielen Teilen in dieser Programmiersprache entwickelt worden. Das liegt unter anderem daran, dass C genau mit der Entwicklung von Betriebssystemen (spezifischer UNIX) im Kopf entworfen wurde [26]. Natürlich hat diese Programmiersprache auch eine ganze Reihe an Nachteilen, eine Abwägung mit den Vorteilen erübrigt sich für uns, da wir durch den Rahmen des Kurses bereits diesbezüglich eingeschränkt sind: Es ist die Sprache, auf Basis derer gelehrt wird, recht „simpel“ im Umfang (bspw. verglichen mit C++ oder Rust) und es kann ein gewisses Grundwissen vorausgesetzt werden, zudem sind viele Ressourcen verfügbar. Aus ähnlich konservativen Gründen werden als Build-System Makefiles genutzt, um genau zu sein GNU/Makefiles [11].

2.1. Struktur

Mein Ziel es ist eine klare Linie bei der Entwicklung zu bewahren und an jeder Stelle die Designentscheidungen klar zu machen und zu betonen. Das fängt schon bei der Ordnerstruktur und Quellcodeorganisation an; der Kernel ist in drei Teile geteilt:

Hardware-Abstraction-Layer Der Ordner `hw` stellt für unterschiedliche Geräte die Treiber bereit sowie für gesamte Rechner „Boards“ die Beschreibungen der verwendeten Komponenten etc.

Systemdienste Enthält ausschließlich hardwareunabhängige Initialisierungsroutinen und Dienste wie Scheduler, Inter-Process-Communication usw. Schreibt eine Schnittstelle für die Kommunikation vor, die von der `hw`-Schicht implementiert werden muss.

Kernel-Applikationen In einem Microkernel eigentlich ein leerer Ordner, denn hier werden Kernel-Programme abgelegt, die „richtige Arbeit“ verrichten. In diesem Microkernel werden dort die Programme abgelegt, die die Funktionsweise des Kernels für die einzelnen Übungszettel demonstrieren sollen, da diese – zumindest solange es keine Systemaufrufe gibt – im Kernel lagern müssen.

Auf diese Art und Weise wurden direkt von Beginn an zwei fundamentale Konzepte umgesetzt: Die besagte Trennung von Mikrokernel-Diensten und eigentlichen Programmen sowie Ansätze des „Sanduhr-Modells“. Die Sanduhr besteht hierbei aus der breiten Unterstützung verschiedenster Hardware und der Möglichkeit unterschiedlichste Software auszuführen, wobei das System nur eine „dünne“ Schnittstelle dazwischen ist.

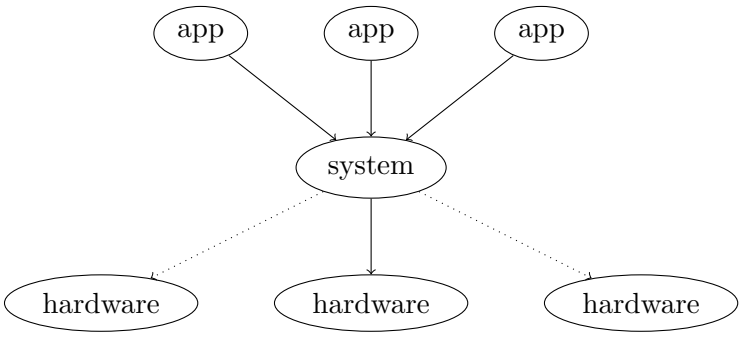

Abbildung 2: Aufbau des Betriebssystems: „Sanduhr“

2.1.1. Build-System

Bevor ich nun auf die einzelnen Teile genauer eingehe, soll kurz die Struktur des Build-Systems erläutert werden, um die Funktionsweise und die Gedanken dahinter bewusst zu machen. Makefiles als Build-System bieten eine durch POSIX [13] standardisierte Sprache, um deklarativ Targets zu definieren, die mittels Regeln aus ihren Voraussetzungen erstellt werden. Das ist ein wenig vorstellbar wie ein Kochrezept, wobei die Targets das Gericht mit jedem seiner Zwischenschritte, die Regeln die Anweisungen, und die Voraussetzungen entweder vorhergehende Zwischenschritte oder bereits vorhandene Zutaten sind. Am Ende bildet dann das „Rezept“ zum Erstellen eines Targets einen Baum aus Zwischentargets als Knoten, wobei jeweils die Kinder eines Knotens die Vorbedingungen des selbigen sind.

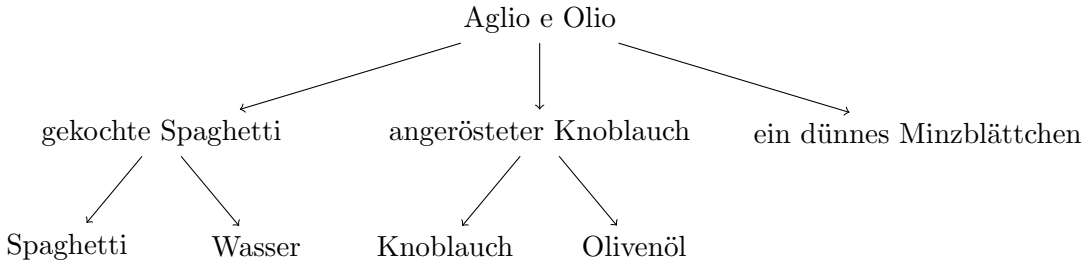

Abbildung 3: Beispielhafter Abhängigkeitsbaum

Diese Regel- und Abhängigkeitshierarchie kann sich nun natürlich immer weiter spinnen, an unterster Stelle muss aber etwas stehen, was man definitiv hat – bei einem Softwareprodukt ist das der Quellcode. Ein Beispiel für ein Makefile zum Erstellen eines Programmes ist im Codebeispiel 1 gegeben.

In diesem Rezept gibt es die sog. automatischen Variablen `$@`, `$^` und `$<`. Erstere wird hierbei durch das Target (vor dem Doppelpunkt) und die folgenden durch alle – respektive nur die Erste – der Vorbedingungen (nach dem Doppelpunkt) ersetzt werden.

Die Regel benutzt die Variablen für den C-Compiler (standardmäßig `cc`), den C Präprozessor, Kompilations- und Linkereinstellungen sowie weitere Bibliotheken (standardmäßig leer). Existieren also in dem gleichen Verzeichnis wie die Datei `Makefile` die benötigten Dateien `foo.c` und `libbar.a`, dann kann mit einem Aufruf von `make` das Programm gebaut werden:

```
> make
```

```
# Tgt: Pre1 Pre2
# |------>Rule
# |------>Rule

# first rule = default rule, usually called all
all: foo

foo: foo.o libbar.a
	$(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
foo.o: foo.c
	$(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
```

Codebeispiel 1: Einfaches Makefile mit automatischen Variablen

```
all: foo

foo: foo.o libbar.a
	$(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
%.o: %.c
	$(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<

clean:
	$(RM) *.o foo

.PHONY: clean all
```

Codebeispiel 2: Makefile mit Pattern-Regeln

```
cc -c   -o foo.o foo.c
cc  -o foo foo.o libbar.a
```

Nun soll häufig jedes Objekt `a.o` aus einer zugehörigen Quellcodedatei `a.c` gebaut werden. Um das zu tun, unterstützt GNU/Make Pattern-Regeln, mit denen vereinfacht sich das Beispiel wie im Codebeispiel 2 zu sehen. Zudem wurde dort ein weiteres Target hinzugefügt, um Kompilationsprodukte zu bereinigen. Eine weitere Besonderheit ist das besondere „unechte“ `.PHONY`-Target, welches keinerlei Regeln hat, aber dessen Prerequisites selbst dann „erstellt“ werden sollen, wenn bereits Dateien vorliegen, die den Namen des Targets tragen. So erstellen weder die Regeln zu `all` noch `clean` Dateien mit diesen Namen und sollen, selbst wenn es solche Dateien gäbe, ausgeführt werden.

An sich ist damit ein einfaches aber funktionstüchtiges Makefile fertig und somit möchte ich jetzt zu dem tatsächlich eingesetzten kommen. Der größte Trick ist, die Liste an Objekten, die am Ende zu dem Kernel zusammengelinkt wird, in eine Variable auszulagern mit dem vorerst verwirrenden Namen `obj-y`. Es gibt nun in jedem Ordner in dem Quellcodedateien liegen, sowie in dem entsprechenden übergeordneten Ordner, eine Datei `Makefile.objs`. Jede dieser bindet alle `Makefile.objs` in den Unterordnern und wird ebenfalls selber rekursiv von dem Wurzel-Makefile eingebunden. Diese Makefiles befüllen erst die besagte Variable, wie zum Beispiel in diesem Auszug aus `system/Makefile.objs`:

```
obj-y += system/init.o
```

Es gibt jedoch auch Objekte, die nicht immer in den Kernel gelinkt werden sollen. Dies wird über Variablen geregelt, die Namen tragen wie `CONFIG_AT91DBGU` und entweder den Wert `y` oder eben einen anderen haben. Darüber wird bestimmt, ob das Objekt der Liste `obj-y` oder einer anderen (unbenutzten) angehängt wird. Als ein solches Beispiel ist hier ein Auszug aus `hw/uart/Makefile.objs` gegeben:

```
obj-$(CONFIG_AT91DBGU) += hw/uart/at91dbgu.o
```

Soll nun der Treiber für die Debug-Unit im Kernel benutzt werden, wird die Variable auf „yes“ gesetzt und somit der Liste angehängt. Um dieses modulare System zu vervollständigen benötigt das Haupt-Makefile eine `config.mak` genannte Datei, welche die genannten Variablen entsprechend setzt. Für das eingesetzte Board ist bereits eine Datei `portux_config.mak` gestellt, welche schlicht mit einer symbolischen Verknüpfung „aktiviert“ werden kann.

```
./
├─ Makefile – enthält Regeln
├─ config.mak (portux_config.mak) – konfiguriert Features
├─ app/
├─ hw/ – Hardware-Abstraction-Layer
│  ├─ Makefile.objs – bindet Unterordner ein
│  └─ uart/
│     ├─ Makefile.objs – fügt at91dbgu.o der Objektliste hinzu
│     ├─ at91dbgu.c
│     └─ at91dbgu.h
└─ system/
   ├─ Makefile.objs
   └─ system.c
```

Abbildung 4: Auszug Makefile-Hierarchie

Der letzte Trick, der noch angewandt wird, ist eine automatische Generierung von Abhängigkeitsinformationen. Was sich sehr komplex anhört, ist zumindest als Problem recht simpel: Ein Objekt muss dann rekompiliert werden, wenn der Quellcode sich ändert. Dies kann Make schon von Haus aus, indem es die Modifikationsdaten des Targets mit denen seiner Voraussetzungen vergleicht. Ist eine der Voraussetzungen aktueller als die Zieldatei, wird neu gebaut. In C gibt es aber eine kleine Unklarheit, nämlich besteht dort der „Quellcode“ der Objektdatei nicht nur aus ihrer eigentlichen Quelldatei, sondern auch aus all den inkludierten Headern, die durch den C-Präprozessor beim Aufruf des Compilers in die Kompilationseinheit kopiert werden. Man muss also beim Bauen von `a.o` aus `a.c` nicht nur diese beiden Dateien bzw. die letzten Schreibzugriffe auf diese vergleichen, sondern auch die von `a.c` benutzten Header. Diese können sich aber wiederum ändern, weshalb bei jedem Kompilationsvorgang die Abhängigkeitsinformationen aktualisiert werden müssen. Dies unterstützen die verwendeten Compiler, sie generieren tatsächlich Makefiles der Form:

```
a.o: a.c foo.h bar.h baz.h
```

```
struct dbgu {
        uint32_t __unused[5];

        uint32_t du_sr;
        uint32_t du_rhr;
        uint32_t du_thr;

        uint32_t __unused2[9];
        uint32_t __reserved[45];
        uint32_t __unused3[9];
};

unsigned char dbgu_getc(volatile struct dbgu *const dbgu);
void dbgu_putc(volatile struct dbgu *const dbgu, unsigned char c);
```

Codebeispiel 3: API bereitgestellt durch Gerätetreiber

```
#define DBGU_BASE 0xFFFFF200
static volatile struct dbgu *const dbgu = (struct dbgu*)DBGU_BASE;
void serial_putc(unsigned char c)
{
        dbgu_putc(dbgu, c);
}
unsigned char serial_getc(void)
{
        return dbgu_getc(dbgu);
}
```

Codebeispiel 4: Nutzung des Treibers bei der Implementierung der Board-API

Diese werden zusätzlich in den `Makefile.objs` eingebunden und somit in das Repertoire an Zielen mit ihren Voraussetzungen aufgenommen.

Ein Problem bleibt jedoch: Wird konditionale Kompilation mittels Präprozessor genutzt, so kann es sein, dass mal einen Header eingebunden wird und mal nicht. Wird nun die dazugehörige Variable nicht durch eine bereits als Abhängigkeit erfasste Datei gesetzt, sondern direkt auf der Konsole, kann nicht mehr komplett für die Funktionsweise des Systems garantiert werden. Abhilfe ist kaum noch elegant möglich ohne Make als Build-System neu zu erfinden, was mit Mk [15] von Plan 9 [16] oder ninja [12] von Google auch schon mehrfach getan wurde.

2.2. Hardware-Abstraction-Layer

„Hardware: the part of the computer ‚that you can kick.'" (Grace Hopper [32])

Nun gehe ich auf die einzelnen Komponenten des Kernels ein, zuerst auf den hardwarespezifischen Teil. Das sind neben den Treibern auch die Initialisierungsroutine des konkreten „Boards".

Die Treiber für die Komponenten sollen unabhängig von den Boards sein, sprich sie stellen eine gerätespezifische API bereit, wie Funktionen um auf Register zuzugreifen. Nur das Board jedoch enthält beispielsweise die Adressen der Speicherbereiche für den Zugriff. Beispielhaft sei hierfür in den Codebeispielen 3 und 4 die konkrete Umsetzung des Treibers für die DBGU der AT91-Serie gezeigt, welcher in der ersten Aufgabe des Kurses geschrieben werden muss.

Die Kapselung ermöglicht es, die Treiber in kompletter Isolation zu schreiben und erst mit der Gesamtheit des Boards zusammenzuführen; auf die Implementierung der Board-System-API wird aber später noch mal zurückgekommen.

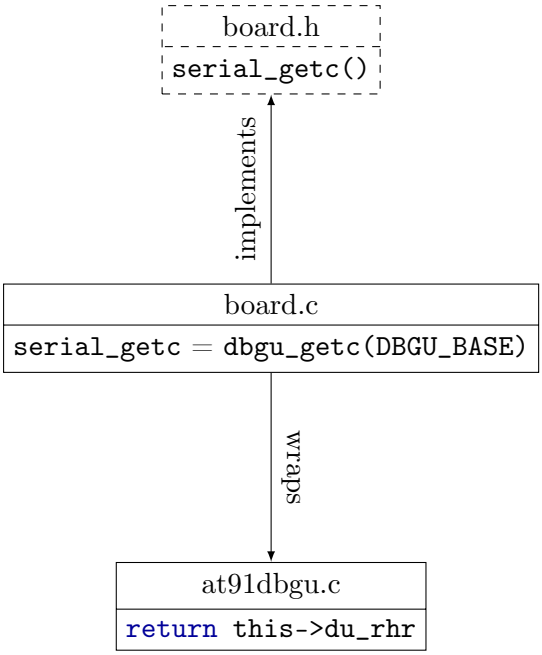

Abbildung 5: Beispiel der Abstraktion Anhand des DBGU-Treibers

Gleichzeitig wäre jedoch eine noch stärkere Abstraktion möglich. Denkbar wäre zum Beispiel die einheitliche Definition einer Schnittstelle, die die jeweilige Kategorie von Peripherie wie UART oder Timer implementieren muss. Dies hätte einige Vorteile, so wäre die Implementierung der Board-API nichts weiter als eine Beschreibung der Peripherie und könnte gegebenenfalls sogar komplett automatisiert generiert werden. Es wäre sogar denkbar, dem Betriebssystem an sich viel mehr Zugriff direkt auf die Hardware zu geben, solange das über dieses abstrakte Interface geschieht. Somit könnte mehr Abstraktion in den hardwareunabhängigen Teil des Kernels wandern.

Trotzdem habe ich mich gegen diesen Ansatz entschieden. Einer der Hauptgründe ist schlicht die verfügbare Zeit: Es müssten für alle Arten von Komponenten Treiber-Interfaces geschrieben werden, die die schwierige Balance halten, nicht zu viel zu spezifizieren und somit von nur einem Bruchteil der Hardware implementiert werden könnten und gleichzeitig nicht zu wenig, sodass das Interface unbrauchbar wird.

Diese Problematik wird generell durch die sehr heterogene Hardwarelandschaft erschwert. Manche Plattformen unterstützen das Austauschen der Komponenten zur Laufzeit und manche gar nicht. Am Ende wird es sehr schwierig allen unterschiedlichen Architekturen und Konfigurationsmöglichkeiten in einem Interface gerecht zu werden, auch wenn dies bei besonders simplen Geräten wie LEDs möglich wäre. Aber schon etwas komplexere Schnittstellen wie UART können zu einem Problem werden, möchte man diese bspw. mittels eines Direct-Memory-Interface-Controllers (DMI) ansprechen. Plötzlich müssen Komponenten auf tiefster Ebene miteinander arbeiten und kommunizieren können, so muss der DMI-Controller genau die Speicheradresse der entsprechenden Memory-Mapped-Register der UART kennen. Dies ist aber nicht nur boardspezifisch, sondern auch von der konkret eingesetzten UART abhängig. Das Interface müsste also

Möglichkeiten bieten, um Hardwaredetails aus dem System heraus abzufragen, und damit effektiv hardware-abhängige Logik in das eigentlich abstrakte System tragen und unser Modell damit komplett zunichtemachen. Die einzige Möglichkeit, die im Designprozess noch theoretisch zumindest umsetzbar erschien, war ein Einsetzen von einem objektorientierten Programmierstil mit gemischter Laufzeit- und Compilezeit-Polymorphie und den Zugriff auf die Geräte über „Opaque Handles“. Ein solches System würde jedoch sowohl das Zeitlimit für die Implementation überschreiten, als auch das Ziel der angestrebten Minimalität verfehlen. Die Gefahr ist ein komplexes System durch eine so komplexe Lösung zu ersetzen, dass die Lösung selbst wieder zum Problem wird: Over-Engineering.

Stattdessen wird die komplexe Problematik, durch eine Schnittstelle jede Hardwarekomponente abfangen zu wollen, reduziert auf die vielen einfachen kleinen Probleme, die in boardspezifischen Code ausgelagert werden. Dieser wiederum fasst nicht nur die Geräte zusammen, sondern auch die Kommunikation unter diesen und definiert die Schnittstelle zu dem hardwareunabhängigen Betriebssystem. Damit ist man eigentlich bereits zum Herzen des Kernels in dem Umfang, wie dieser hier implementiert wird vorgedrungen: Der Schnittstelle zwischen der Gesamtheit der eingesetzten Hardware als Plattform und der Betriebssystemfunktionalität, sprich den „Diensten“ die ein mikrokernelartiges System bereitstellt.

2.2.1. Hardware-zu-Software-Interface

Das Interface das von der Hardware bereitgestellt wird, soll flexibel genug sein, um alle angestrebten Nutzungsszenarios zu unterstützen und gleichzeitig so klar, dass in dem hardwareunabhängigen Teil auch keine hardwarespezifischen Abfragen sein müssen.

Hierzu müssen zuerst die Nutzungsszenarien angesehen werden um daraus abzuleiten, welche Hardware unterstützt werden soll bzw. welche Annahmen über diese gemacht werden können – und welche „Art“ von Betriebssystem am Ende das Ergebnis darstellen soll. Bei der Hardware limitiere ich mich auf recht „feste“ Systeme, schränke mich also bei dem hardwaremäßigen Austausch/Ersetzen von Peripherie vorerst ein. Es ist nicht möglich eine andere Hardwareinitialisierungsroutine im Kernel zur Laufzeit auszuführen – dies erspart viel Komplexität. Werden also Komponenten ausgetauscht, muss entweder der Kernel ersetzt werden oder die Komponente muss so unkritisch für das System sein, dass ihr Treiber im Userspace laufen kann. Der Kerneltreiber unterstützt dann nur die Hardwareschnittstelle und propagiert diese nach außen über das System an den Userspace. Des Weiteren treffe ich viele, zumindest scheinbar grundsätzliche, Annahmen, wie die registerbasierten CPUs oder Speicher mit Speicheradressen – schon weil die Programmiersprache C diese Annahmen macht – aber auch habe ich Funktionen in dem Interface festgeschrieben, die diese Voraussetzungen aus ihrer Schnittstellenspezifikation heraus benötigen.

Durch die Einschränkungen wäre dieses System nicht mehr ohne Weiteres einsetzbar auf normalen IBM-PCs, wo durchaus jede Komponente wie CPU oder Mainboard mit Chipset ausgetauscht, die Betriebssystem-Installation aber beibehalten wird. Es wäre also nötig auf Basis des geringsten gemeinsamen Nenners aller x86 (oder zumindest IA32 und AMD64) Plattformen zu programmieren und alle Funktionalität darüber hinaus in dem für den Userspace bereitgestellten Interface ansprechbar und steuerbar zu machen.

Somit bleibt primär der Embedded-Bereich, in dem sich das eingesetzte PORTUX-920T-System ohnehin bewegt. Es kommt jedoch eine mögliche weitere Plattform hinzu: Es wäre denkbar das Hardware-Interface basierend auf einem Host-Betriebssystem wie Linux zu schreiben, sodass man das Embedded-System außerhalb einer virtuellen Maschine „direkt“

als Programm in Linux ausführen könnte. Das hat mehrere Vorteile, vom Testen der dafür geschriebenen Applikationen bis zum Testen der gesamten hardwareunabhängigen Funktionalität selber.

2.2.2. Software-zu-Hardware-Interface

Zusätzlich zu dem Interface der Hardware die das System verwalten möchte, muss es noch ein weiteres geben: Die Schnittstelle für die Hardware um Systemfunktionen auszuführen. Die kritischste dieser Funktion ist `system_init()`, denn sobald die hardwarespezifische Initialisierung abgeschlossen ist, muss eine Übergabe an das eigentliche Betriebssystem stattfinden. Darüber hinaus jedoch gibt es noch weiteres an Systemfunktionalität, die in diesem Interface definiert werden muss. Diese Schnittstelle ist in vielerlei Hinsicht eine besondere Herausforderung. Während normalerweise das Betriebssystem den Zugriff auf die Hardware verwaltet und somit die Kontrolle über den Programmfluss hat, ist diese Schnittstelle dazu gedacht, der Hardware (bzw. den Treibern) die Möglichkeit zugeben „von sich aus“ System-Funktionalität auszuführen. Das bringt viele Probleme mit sich.

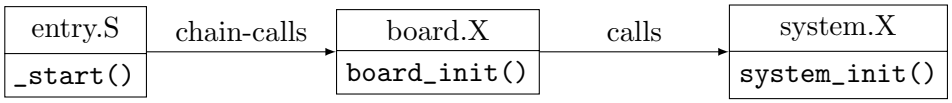

Abbildung 6: Systemstart: Zugriff von Hardware auf Software-Interface

Doch leider ist so eine Schnittstelle nötig, denn anders ist das Benutzen von Interrupts außerhalb des hardwarespezifischen Teils nicht möglich: Bei einem Software-Interrupt muss die entsprechende Syscall-Routine im Kernel ausgeführt werden, bei einem Data-Abort gegebenenfalls das zugehörige Nutzerprogramm beendet werden, usw. usf.

Möchte man darüber hinaus bei einem Absturz des Kernels („Kernel Panic“) auch noch hardwarespezifische Informationen wie Registerinhalte ausgeben, muss diese Funktion aus dem Hardware-Bereich auf die Funktion `printf()` aus dem System-Bereich zugreifen. Es kann damit sogar sein, dass die Ausführung der Funktion unterbrochen wird und diese dann wieder darin verschachtelt aufgerufen wird. Das ist so lange kein Problem, wie `printf()` auf keine geteilten Ressourcen zugreift, oder nur auf eine Art und Weise, dass es nicht zu Race-Conditions kommt. Ein gepuffertes Lesen und Schreiben kommt damit kaum in Frage, denn hier sicherzustellen, dass es zu gegenseitigem Ausschluss kommt und es gleichzeitig keinen Deadlock über alle Ebenen hinweg (Treiber, System-Abstraktion, Formatierungsfunktion) gibt, würde die Komplexität des Systems enorm erhöhen. Der gewählte Ansatz ist stattdessen, Kernel-Ausgaben sowieso möglichst zu minimieren und Aufrufe von Systemfunktionen aus der Hardware heraus – soweit es geht – zu eliminieren.

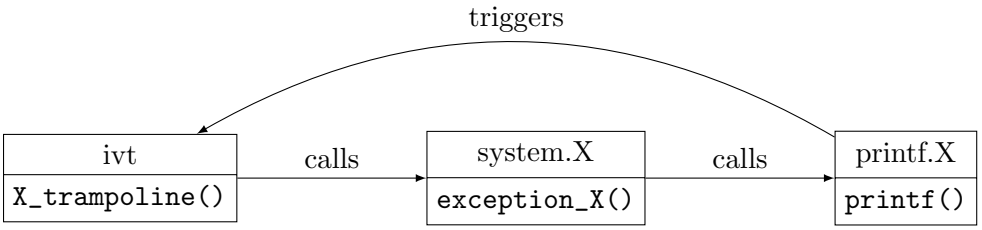

Abbildung 7: Zugriff einer ISR auf Systemfkt. kann zu unendl. Rekursionen o. Ä. führen

2.3. System

„Software: the part of the computer that you want to kick.“

Ich möchte nun zu der Systemfunktionalität an sich kommen, denn es bräuchte keine Kommunikation zwischen Hardware und System, wäre Letzteres nicht vorhanden. Es wurde schließlich der Aufwand betrieben um Funktionalität auszulagern; die Frage ist jetzt: Was konnte als hardwareunabhängige Logik herauskristallisiert werden?

Das System besteht eigentlich nicht aus viel mehr als Initialisierungsroutinen für die Subsysteme im Kernel und diesen Subsystemen selber. Hinzu kommt eine (kleine) Bibliothek an nützlichen Funktionen, die im Kernel bereitgestellt werden sollen.

Das Subsystem, welches zum aktuellen Zeitpunkt am ausgereiftesten ist, ist das Gegenstück zu dem, dessen Problematik in den Abschnitten zum hardwarespezifischen Teil geschildert wurde: die Ausnahmenbehandlung. In dieser wird – wie vorher angedeutet – bei Ausnahmen wie einem Data Abort das System angehalten und ausführliche Debuginformationen ausgegeben. Dieser Kernel-Panic ist aber nicht immer das Mittel der Wahl. Und hier zeigen sich langsam die Vorteile der Trennung der Welten: Die Schnittstelle erlaubt einem stattdessen ggf. nur den Thread, der die Ausnahme ausgelöst haben könnte, zu beenden, sollte Threading implementiert sein. Es wäre möglich ein komplett anderes (und „vollständigeres“) System zu bauen, komplett unabhängig von der Hardware, welches aber die gleichen Schnittstellen benutzt – wie hier vorgestellt. Dieses müsste sogar nicht mal Microkernel-Prinzipien verfolgen.

Wenn jetzt schon so viel Aufwand getrieben wurde den hardwareabhängige Teil von dem Rest zu trennen, dann ergibt es durchaus Sinn diese Aufgabentrennung weiterzudenken, d. h. jetzt auch eine klare Trennung von Kernelmodus und Nutzermodus zu schaffen und wieder möglichst viel Logik auszulagern.

Die Ausmaße, die das annehmen kann, soll die Implementierung von einer analogen Funktion zu dem UNIX-Syscall `read()` [13] (auf `stdin`) zeigen. Hierzu sollte der Treiber für die serielle Schnittstelle auf interruptgesteuerte Arbeit umgestellt werden. Das bedeutet, dass die konkrete Hardwarekomponente die CPU bei jedem Zeichen die CPU in ihrer Ausführung unterbricht. Die CPU führt nun eine Interrupt Service Routine aus, die das Zeichen liest und in der Regel in eine Art Puffer schreibt, wo es bei Bedarf ausgelesen werden kann. Klassischerweise ist dieser Puffer im Treiber oder zumindest im Kernel implementiert – obwohl eigentlich solch eine abstrakte Datenstruktur durchaus ein Nutzerprozess sein kann. Der Ansatz ist klar: Ein `read()` ist in diesem System kein Syscall, sondern eine Usermode-Funktion, die aus einem ebensolchen Puffer liest. Das Schöne dabei ist, dass dadurch an dieser Stelle sogar Overhead reduziert werden kann, da kein Wechsel aus dem Nutzermodus heraus nötig ist. Einen Syscall `getc()` gibt es ebenfalls nicht und möchte man mehr Zeichen lesen als es gibt, wird auch nur im Nutzermodus gewartet, bis durch den Kernel entsprechend viele Zeichen im Puffer abgelegt wurden.

Aber es wäre nicht Betriebssystemdesign, würde dieser Ansatz nicht ebenfalls Probleme bergen. Gibt es mehrere serielle Schnittstellen, muss irgendwo festgelegt werden, welche die Schnittstelle ist, die für die Konsole benutzt werden soll – und welche eine andere Verwendung hat, bspw. die Kommunikation mit anderer Peripherie. In den meisten Fällen ist diese Flexibilität auf höherer Ebene aber nicht nötig, sondern kann im Board implementiert werden, da solche über UART angeschlossenen Geräte fest verdrahtet sind. Hierbei verdeutlicht sich ein weiterer Vorteil der Designentscheidung: Ob Daten zwischengepuffert werden sollen, sollte nicht im Treiber der UART entschieden werden,

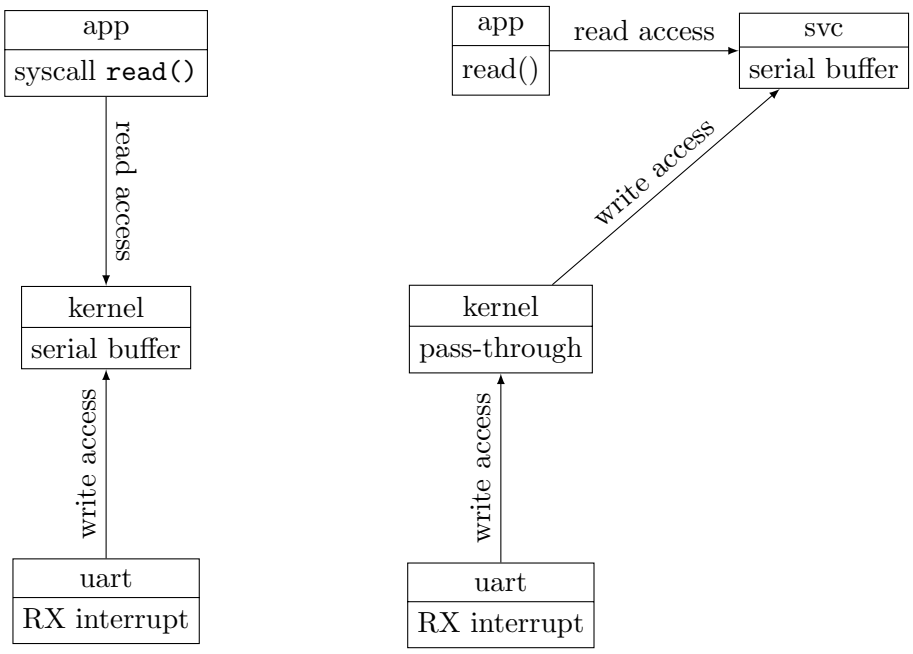

Abbildung 8: Read als Syscall in UNIX vs. als Aufruf eines User-Mode Dienstes

sondern bei der eigentlichen Verwendung, sprich mindestens im System-Level, besser noch im Nutzer-Level.

Etwas schwerwiegender ist aber ein anderer Nachteil: Der Kernel schreibt weiterhin ungepuffert und auch noch auf das gleiche Interface wie der Nutzer. Aber auch diese Schwierigkeit ist keine, die allzu große Kopfschmerzen bereiten sollte, schließlich ist der Kernel so klein, dass – sollte er auf die Konsole schreiben – es sich in den meisten Fällen um Debug-Informationen oder gar einem kompletten Kernel-Panic halten dürfte. Wenn die Ausgabe des Nutzers mittendrin unterbrochen wird, dürfte dies nicht weiter tragisch sein. Essenziell ist lediglich eines: Die Nutzermodus-Implementierung von `read()` (und noch wichtiger `write()`) sollte möglichst keine Nutzermodus-Fehler werfen die abgefangen werden müssen, denn wie sollten diese dann ausgegeben werden? Bei einem so eingeschränkten Interface ist das jedoch eine machbare Aufgabe.

Durchaus relevant bei einer solchen Designentscheidung ist, dass dies plötzlich in einem komplett umgekehrten Interface im Vergleich zu einem klassischen UNIX resultiert. Während UNIX eigentlich nur einmal eine Funktion im Nutzermodus aufruft (den init-Prozess) und sonst keinerlei „Ansprüche“ an das Userland hat, benötigt dieses System eine deutlich umfangreicheres Interface, welches von dem Userland implementiert werden muss, damit das Gesamtsystem funktionieren kann. Interessanterweise ähnelt dieses wiederum UNIX – nur eben, dass nicht der Nutzer `read()` und `write()` im Kernel aufruft, sondern, dass andersherum der Kernel einzelne Zeichen im Userland ablegt und abfragt.

In-Kernel Applikationen Nach dieser ausführlichen Beschreibung des Kerneldesigns möchte ich die Beschreibung des Gesamtsystems abschließen, und zwar mit Applikationen, die im Kernel laufen – also komplett konträr zu dem eigentlichen Ziel der Trennung von Aufgaben. Wie jedoch am Anfang des Abschnitts 2.1 auf Seite 12 beschrieben wurde, besteht der Kernel-Quellcode auch aus einem Ordner „app“, welcher eben diese enthält

– oder leer ist bei einem eigentlichen Microkernel. Der Grund für die Existenz dieses Ordners ist schlicht dem Aufbau des Kurses „Betriebssysteme“ geschuldet, bzw. den Aufgabenstellungen, die vor dem Aufbau eines Syscall-Interfaces kommen und somit in diesem Ordner landen müssen. Es handelt sich also hierbei eigentlich nur um „Demos“ der Low-Level-Funktionalität des Kernels und sie sind aus dem Anspruch geboren, das System möglichst inkrementell aufzubauen: sprich Logik aus den vorherigen Aufgabenstellungen nur zu erweitern. Während das bei den Treibern und dem System an sich durch sauberes Design gelöst werden kann, sind die Demos in sich abgeschlossene Funktionseinheiten, welche sich nicht ergänzen können. Somit dürfen diese nebeneinander in diesem Ordner existieren und können dann zur Kompilierzeit ausgewählt werden, um diesem Teil der Arbeit gerecht zu werden.

2.4. QEMU

„And Now for Something Completely Different...“ (Flying Circus S1E2 [24])

Abschließend möchte ich noch über einen ganz anderen Gegenstand dieser Arbeit reden, nämlich der eingesetzten Virtualisierungs- und Emulationslösung, die in diesem Rahmen überarbeitet und auf einen aktuellen Stand gebracht wurde. Der mir vorliegende Zustand war eine gepatchte Version von QEMU 2.1.2 [3], bei der die Unterstützung für das Board hinzugefügt wurde; das Ziel war eine aktuelle Version von QEMU mit äquivalentem Funktionsumfang. Hierbei wurde zuerst das Git-Repository von QEMU [22] heruntergeladen um dann auf einer separaten Branch den Patch anzuwenden. Daraufhin wurden die einzelnen Änderungen gesichtet, lokal dokumentiert und in logische Abschnitte unterteilt. Diese wurden dann wiederum einzeln der aktuellen Entwicklungsversion hinzugefügt und die dabei angefallene Portierungsarbeit in kleinen Stücken geleistet, bis alles übertragen wurde.

Problematisch war bei dieser Arbeit war, dass in dem betreffenden Zeitraum viel Refactoring stattfand, bei dem interne APIs komplett umgestellt wurden, so wurde QDev (2009) [2] durch QAPI (2011) [4] ergänzt und auf Basis dss QObjectModels (QOM, ebenfalls 2011) [6] reimplementiert [14]. Die Dokumentation dieser (vielen kleinen) Umstellungen wurde aber nur temporär und unvollständig vorgehalten. Da nun dieser Umstieg abgeschlossen war, war es nötig Patches von vergleichbaren Boards zu sichten, die eben diese Umstellungen mitgemacht haben und die (oft nur in den Commit-Nachrichten vorhandenen) Beschreibungen der Veränderungen zu lesen.

Sobald jedoch das Prinzip dieser Änderungen verstanden war, war die eigentliche Arbeit größtenteils Anpassungen an vorhandenen Code und nicht weiter bedeutend. An einigen Stellen wurde die Arbeit sogar einfacher gemacht, da bestimmte APIs vorher nicht existierten und deshalb durch den Patch tief greifende Änderungen in ganz anderen Teilen des Quellcodes notwendig waren, die nun wegfielen. Aber an anderer Stelle blieb es leider bei Workarounds, da QEMU standardmäßig bestimmte Register nicht emuliert, die aber für den Anwendungsfall im Kurs notwendig sind. Darüber hinaus hat die aktuelle Version von QEMU Unterstützung für neuere ARM-Architekturen eingepflegt, welche sog. Exception-Levels [21] definieren. Diese werden intern auch nun für ältere Architekturen genutzt, was zu fehlerhaftem Verhalten führt wird ein Data Abort ausgelöst. Hier ist noch Ausbesserungsarbeit nötig.

3. Evaluation und Fazit

„Es könnte schöner sein“ (Faber [9])

Ich möchte nun rüblickend betrachten, ob die Eingangs gelisteten Ziele eingehalten werden konnten. Das hier konstruierte Betriebssystem sollte möglichst verständlich aufgebaut sein und von vornherein die Trennung von Zuständigkeiten der einzelnen Quellcode-Bereiche klar machen. Die Lösungen sollte iterativ aufeinander aufbauen und nicht bei jeder Änderung eines Teils des Systems Änderungen an anderen Teilen erzwingen. Der hier gesetzte Fokus lag auf der Vermeidung von unnötiger Komplexität – da es nicht darum geht den performantesten oder sichersten Kernel zu schreiben, sondern ein System zu schaffen, an dem man lernen kann.

Hierzu wurden einige Designentscheidungen getroffen, um genau diese Anforderungen zu erfüllen. Die radikalste Entscheidung war die, den hardwareabhängigen Teil auf unterster Ebene komplett in ein gemeinsames Interface wegzuabstrahieren. Dies erfüllt nicht nur die Bedingung der Trennung von Zuständigkeiten, sondern führt auch direkt dazu, dass der Hardware-Abstraction-Layer selbst minimal sein muss, gerade so komplex um seine Aufgabe zu erfüllen; es darf darin keine weitere Funktionalität implementiert werden. Als Beispiel dafür wurde die Auslagerung des Pufferns von seriellen Ein- und Ausgaben in das System herein ausführlich beschrieben. Damit bestehen die Treiber für die einzelnen Peripheriegeräte fast nur aus Spezifikationen ihrer Register und Gettern und Settern. Die Zusammenführung der Treiber zu einem Board wird danach auf ein simples Stecksystem reduziert, indem die einzelnen benötigten Komponenten aktiviert und verbunden werden, bzw. mit kleinen Wrapper-Funktionen nach außen als Teil des Interfaces bereitgestellt. Dies ermöglicht es auch, einzelne Komponenten elegant und unabhängig von einander zu entwickeln – spezifisch auch Aufgaben zu implementieren, ohne tiefgreifende Veränderungen vorzunehmen, außer dem Hinzufügen der benötigten neuen Logik.

Auch die Entscheidung, die „Anwendungslogik“ für die Demonstration der einzelnen Übungsaufgaben aus dem eigentlichen System zu trennen, hat sehr geholfen dieses Ziel zu erreichen. Dadurch blieb das eigentliche System frei von „echter Arbeit“ und die Demos können bei der Kompilationszeit ausgewählt werden, funktionieren also losgelöst von dem Kernel an sich.

Ein geschicktes Design des Interfaces und Wahl der Prämissen war dazu kritisch und keineswegs einfach. Wie schon angedeutet, mussten auch hier Kompromisse gemacht werden. So ist das entwickelte System keineswegs für alle Plattformen geeignet, trotz seiner sehr abstrahierten Implementierung, da es bspw. das Wissen um die eingesetzte Hardware zur Kompilierzeit voraussetzt. Diese Kompromisse ermöglichen es, ein etwas weniger flexibles aber deutlich simpleres Abstraktionskonzept einzusetzen. Da unsere eigentliche Plattform sogar nur einen spezifischen Kleinrechner umfasst, ist die Abwägung hier zwischen Schlichtheit und Flexibilität zugunsten der einfachen Lösung ausgefallen.

Trotzdem wurde nicht alles so erreicht wie ursprünglich gedacht. So ist der Umfang der Funktionalität zum Zeitpunkt der Ausarbeitung dieser Arbeit deutlich unter dem angestrebten Ziel einer komplett als Ersatz einsetzbaren Lösung für den Kurs „Betriebssysteme“, da schlicht nicht alle Aufgaben unter diesen zusätzlichen Voraussetzungen implementiert werden konnten. Jedoch war absehbar, dass eine Ergänzung zu dem dafür benötigten Umfang schlicht viel mehr Zeit gebraucht und den Rahmen dieser Arbeit gesprengt hätte. So wurde sich dafür entschieden, das verknüpfte Softwareprojekt in diesem

Zutand zu pausieren und die generelle Problematik mit den gefundenen Lösungsansätzen zu schildern.

Es wurden jedoch die Grundsteine für ein System gelegt, welches in der Lehre explizit eingesetzt werden kann, um die Problematiken und Lösungsansätze der Kernel-Programmierung beizubringen. Zwar sind lange nicht alle Anforderungen an ein „komplettes“ System erfüllt, aber dies kann nun ergänzt werden, da eine saubere Platform geschaffen wurde.

3.1. Ausblick

Wie bald das System aber abgeschlossen und einsatzfähig ist, muss sich noch zeigen. Es wäre denkbar, auf Basis der Interface-Definitionen auch andere Hardware-Abstraction-Layer in einer anderen Sprache (z. B. Rust [23] oder auch SPARK [1]) zu schreiben und davon unabhängig das System weiterzuentwickeln. Da die Interfaces so gehalten wurden, dass bei Änderungen der Implementierung sich die ABI nicht ändert, könnten hier theoretisch komplett unterschiedliche Entwicklungen weitergetrieben werden. Dies ermöglicht unterschiedlichste Anwendungs- und Erweiterungsszenarien. So können einzelne Komponenten des Systems mit Unit-Tests auf Korrektheit gegen das Interface getestet werden, ohne ein vollständiges System zu benötigen. Spezifisch wäre es möglich einen Hardware-Abstraction-Layer zu programmieren, der die Ausführung des „Betriebssystems“ als schlichte Anwendung in einem bereits vorhandenen Betriebsysstem ermöglicht.

Dies setzt aber voraus, dass das Interface immer gleich bleibt, was wiederum aber nur möglich ist, sobald diese Schnittstelle hinreichend viele Anforderungen erfüllt, sodass alle angestrebten Anwendungs- und Einsatzzwecke abgedeckt sind. An dieser Stelle muss die eigentliche Entwicklung ansetzen: Es muss dass Einsatzgebiet klar eingeschränkt und die damit verbundene benötigte Funktionalität in der Schnittstelle wiedergeben werden – aber nicht mehr.

Hier lohnt es sich, andere Betriebssystem-Projekte mit in Teilen vergleichbaren Zielen zu analysieren um ggf. aus Fehlern zu lernen und sich auch abzugrenzen.

A. Begrifflichkeiten

In der Arbeit wird von Virtualisierungsumgebungen und Betriebssystemen gesprochen. Um Unklarheiten vorzubeugen sollen diese Begriffe im Folgenden definiert und von möglichen anderen Definitionen abgegrenzt werden und eine Einordnung der Funktionalität in den Computer gegeben werden.

Betriebssystem und Kernel Das Betriebssystem ist eine Schicht, die zwischen der Hardware und den Anwenderprogrammen liegt. Theoretisch ist diese nicht nötig, schließlich könnte man auch versuchen die Nutzerprogramme direkt auf der Hardware laufen zu lassen. Diese Art der Ausführung hat jedoch den Nachteil, dass bei schon kleinsten Änderungen das Programm angepasst werden muss. Hinzukommen weitere Probleme wie Komfort, keine Möglichkeit mehrere Programme gleichzeitig laufen zu lassen etc. Um genau diese Funktionalität aber zu leisten, brauchen wir ein Programm – das Betriebssystem – welches zwischen der Hardware und den Anwendungsprogrammen sitzt. Dieses verwaltet eben diese eigentlichen Anwendungen und stellt eine abstraktere Sicht auf die Hardware her. Während das Betriebssystem weiterhin für jede Hardware angepasst wird, können die Anwendungen auf Basis dieses Abstraktionskonzeptes arbeiten. Das heißt, das Betriebssystem verwaltet die Ressourcen: Es stellt nach außen abstrakte virtuelle Ressourcen zur Verfügung, die wiederum intern realen zugeordnet werden. Optimalerweise ist die Schnittstelle, die das Betriebssystem damit zu den Programmen darstellt, so geartet, dass die Programme zur Ausführungszeit aus ihrer Sicht wie auf einem Batch-Processing-System laufen: ohne Unterbrechung und ohne sich darum zu kümmern, was sonst noch auf dem System existiert oder laufen soll. Dafür wird eine komplette Aufgabentrennung zwischen Anwendung und System geschaffen – wie stark aber diese Trennung vorgenommen wird und wie viel Funktionalität tatsächlich nur im Kern des Betriebssystemes liegt, teilt man in die Kategorien Mikrokernel und monolitische Kernel ein.

Während in einem Mikrokernel nur die Anwendungsprogramme tatsächliche Aufgaben übernehmen, fungiert das Betriebssystem lediglich als Dienstleister und Mittelmann zwischen den Programmen und der Hardware: „A microkernel does no real work“ (Jochen Liedtke [18]). Bei monolithischen Kerneln wird einiges an Funktionalität, die eigentlich aus logischer Sicht als Anwendungsprogramm bezeichnet werden muss, doch mit in den Kernel an sich geladen – häufig aus Geschwindigkeitsgründen. Funktional unterscheiden sich hierbei beide Ansätze nicht. Microkernelsysteme lagern schlicht einige Aufgaben in eine höhere Abstraktionsschicht als Anwendungsprogramme aus. Dies bedeutet zwangsläufig einen gewissen „Umweg“ bzw. „Overhead“.

Am Ende ist es sehr schwierig exakt zu definieren, was ein Betriebssystem ist, um Andrew S. Tanenbaum zu zitieren [30]:

> It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true. Part of the problem is that operating systems perform two essentially unrelated functions: providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources.

Virtualisierung und Emulation Als eine Virtualisierungssoftware in dem Sinne dieser Arbeit und wie sie eingesetzt wird, wird hier ganz allgemein eine Software bezeichnet, die

es ermöglicht Programme in einer durch Software definierten Umgebung laufen zu lassen, d. h. unabhängig von existierender Hardware. Für uns relevant ist eine vollständige Virtualisierung des gesamten Rechners, und ist somit streng genommen eine Emulation. Die hier eingesetzte Software QEMU ist genau solch eine Virtualisierungs- und Emulationssoftware.

Literatur

[1] AdaCore. *SPARK Pro*. URL: https://www.adacore.com/sparkpro/ (besucht am 2019-02-06).

[2] Markus Armbruster. *QEMU's new device model qdev*. KVM Forum, 2010. URL: https://www.linux-kvm.org/images/f/fe/2010-forum-armbru-qdev.pdf (besucht am 2019-02-06).

[3] Technische Universität Berlin. *Toolchain – Betriebssysteme Praktikum*. 2014. URL: https://svn.kbs.tu-berlin.de/svn/BSPrak/2014w/public/toolchain/ (besucht am 2019-02-06).

[4] Eric Blake. *QAPI Interface Design: Best Practices*. KVM Forum, 2013. URL: https://www.linux-kvm.org/images/d/d8/Kvm-forum-2013-qapi-design.pdf (besucht am 2019-02-06).

[5] Jérôme Bon. CC BY 2.0. Wikimedia Commons. URL: https://upload.wikimedia.org/wikipedia/commons/8/80/Larix_decidua_Embrun1.jpg.

[6] Paolo Bonzini. *QOM exegesis and apocalypse*. KVM Forum, 2014. URL: https://www.linux-kvm.org/images/9/90/Kvmforum14-qom.pdf (besucht am 2019-02-06).

[7] R. Cox, F. Kaashoek und R. Morris. *xv6*. 2017-08-29. URL: https://pdos.csail.mit.edu/6.828/2018/xv6.html (besucht am 2019-01-27).

[8] Dwight D. Eisenhower. „Remarks at National Defense Executive Reserve Conference“. In: *Public Papers of the Presidents of the United States, Dwight D. Eisenhower, 1957*. Hrsg. von USA Federal Register Division. 1958, S. 817–820.

[9] Faber. *Es könnte schöner sein*. URL: https://www.youtube.com/watch?v=15KwOgHG8jY (besucht am 2019-02-06).

[10] D. Fiedler. „The History of Unix“. In: *BYTE magazine* (1983), S. 188. URL: https://archive.org/stream/byte-magazine-1983-08/1983_08_BYTE_08-08_The_C_Language#page/n189/mode/2up (besucht am 2019-01-27).

[11] *GNU make Documentation*. URL: https://www.gnu.org/software/make/manual/make.html (besucht am 2019-01-27).

[12] Google. *Ninja*. Build System. URL: https://ninja-build.org/ (besucht am 2019-01-27).

[13] The Open Group. *Base Specifications*. Version 7. 2018. URL: http://pubs.opengroup.org/onlinepubs/9699919799/nframe.html (besucht am 2019-02-06).

[14] Eduardo Habkost. *QEMU internal APIs. How abstractions inside QEMU (don't) work together*. FOSDEM, 2017. URL: https://habkost.net/talks/fosdem-2017/ (besucht am 2019-02-06).

[15] A. G. Hume und B. Flandrena. *Maintaining Files on Plan 9 with Mk*. URL: http://9p.io/sys/doc/mk.html (besucht am 2019-01-27).

[16] Bell Labs. *Plan 9 from Bell Labs*. Operating System. Version 4. 2014. URL: https://9p.io/plan9/index.html (besucht am 2019-01-27).

[17] *Lesson 2 - Windows NT System Overview*. 2014. URL: https://docs.microsoft.com/en-us/previous-versions//cc767881(v=technet.10).

[18] J. Liedtke. „On Micro-kernel Construction“. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. SOSP ’95. Copper Mountain, Colorado, USA: ACM, 1995, S. 237–250. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224075. URL: http://doi.acm.org/10.1145/224056.224075.

[19] B. Linnert. *Betriebssysteme*. Kurs. 2018. URL: http://www.inf.fu-berlin.de/w/SE/VorlesungBetriebssysteme2018 (besucht am 2019-01-27).

[20] J. Lions. *Lions’ Commentary on UNIX 6th Edition with Source Code*. Computer classics revisited. Peer-to-Peer Communications, 1996. ISBN: 9781573980135. URL: https://books.google.de/books?id=OlZ3QgAACAAJ.

[21] ARM Ltd. *Exception Levels*. URL: https://developer.arm.com/docs/100069/0602/overview-of-aarch64-state/exception-levels (besucht am 2019-02-06).

[22] QEMU Project. *QEMU Git Repository Mirror*. URL: https://github.com/qemu/qemu (besucht am 2019-02-06).

[23] Rust Project. *Rust*. URL: https://www.rust-lang.org/ (besucht am 2019-02-06).

[24] Monty Python. *Flying Circus*. TV-Series. Comedy. 1969. URL: http://www.ibras.dk/montypython/justthewords.htm (besucht am 2019-01-27).

[25] Monty Python. *Meaning Of Life*. Movie. Comedy. 1983. URL: http://www.montypython.net/meaningmm1.php (besucht am 2019-02-09).

[26] D. M. Ritchie. „The Development of the C Language“. In: *The 2nd ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: ACM, 1993, S. 201–208. ISBN: 0-89791-570-4. DOI: 10.1145/154766.155580. URL: https://www.bell-labs.com/usr/dmr/www/chist.html (besucht am 2019-01-27).

[27] D. M. Ritchie. „UNIX – A Dialectic“. In: *Proceedings USENIX Association Winter Conference*. Washington, DC, 1987.

[28] R. M. Stallman. *new Unix implementation*. 1983-09-27. URL: https://www.gnu.org/gnu/initial-announcement.html (besucht am 2019-01-27).

[29] A. S. Tanenbaum und A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall Software Series. Pearson Prentice Hall, 2006. ISBN: 9780131429383. URL: https://books.google.co.uk/books?id=RYU%5C_AQAAIAAJ.

[30] Andrew S. Tanenbaum und Herbert Bos. *Modern Operating Systems*. 4th. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN: 9780133591620.

[31] taskit. *Portux920T Starterkit*. URL: https://www.taskit.de/details/product/premium-starter-kit-portux920t.html (besucht am 2019-02-07).

[32] K. Williams. *Grace Hopper: Admiral of the Cyber Sea*. Library of naval biography. Naval Institute Press, 2012. ISBN: 9781612512655.

Abbildung 9: The Larch. [24][5]