Großer Beleg
Kerndebugger für den Mikrokern FIASCO

Jan Glauber
Technische Universität Dresden

19. April 2001

Inhaltsverzeichnis

1 Einleitung

Seit Software existiert, gibt es auch fehlerhafte Software. Die Wahrscheinlichkeit für das Vorhandensein von Programmierfehlern (sogenannte Bugs) steigt dabei mit der Komplexität eines Programmes. Vor allem während der Entwicklungsphase eines Softwareprojektes treten häufig Fehler im Programm auf. Ein hilfreiches Werkzeug bei der Suche und Analyse solcher Fehler ist der Debugger. Betriebssysteme sind auch "nur" Programme, weshalb auch hier zur Fehlersuche Debugger verwendet werden. An der TU Dresden wird an einem Betriebssystemprojekt namens DROPS[1] geforscht, welches auf einem L4-kompatiblen[2] Mikrokern basiert. Im Rahmen dieses Projektes entstand auch eine eigene Implementation der L4 Schnittstelle, namens Fiasco. Für Fiasco gab es bisher nur einen rudimentären Kerndebugger. Diesen Debugger um noch fehlende Funktionalität zu erweitern, war das Ziel dieser Arbeit.

2 Grundlagen

In diesem Kapitel werden einige Aspekte von Betriebssystemen betrachtet, die für den Debugger von Interesse sind.

2.1 Debugger

Softwarefehler[3] können in drei verschiedene Kategorien eingeordnet werden: Fehler in der Syntax, Laufzeitfehler und logische Fehler.

Während Syntaxfehler beim Übersetzen eines Programmes vom Compiler erkannt werden, sind Fehler in fertig übersetzten Programmen schwerer zu finden. Laufzeitfehler entstehen, wenn ein Programm gültige Anweisungen enthält, diese aber bei der Ausführung Fehler verursachen (z. B. Division durch 0). Logische Fehler sind Fehler im Entwurf und der Implementierung, sie äußern sich durch falsche Resultate oder unerwartetes Verhalten. Da Laufzeit- und logische Fehler nicht vom Compiler erkannt werden, muß die Stelle, an der der Fehler im Programm auftritt, selbst gefunden werden. Das kann insbesondere bei nicht deterministisch auftretenden Fehlern kompliziert sein. Um solche Fehler einfacher lokalisieren zu können und ihre Auswirkungen zu beobachten, verwendet man einen Debugger.

Ein Debugger ist also ein Werkzeug, welches bei der Fehlersuche hilft. Debugger werden hauptsächlich während der Entwicklungsphase eingesetzt.

[1] Dresden Realtime Operating System

[2] siehe Kapitel 2.3

[3] Der Begriff Fehler bezieht sich im Folgenden immer auf Fehler, die in der Software auftreten. Hardwarefehler (z. B. defekter Arbeitsspeicher) sind im Rahmen dieser Arbeit nicht von Interesse.

Von einem Debugger erwartet man mindestens:

- Informationen über den Systemzustand
- Programmabarbeitung in einzelnen Schritten
- Variablen- und Speicherinhalt anzeigen
- Unterbrechungsbedingungen und Haltepunkte im Programm festlegen

Aktiviert wird ein Debugger, indem man das zu untersuchende Programm von der Debug-Umgebung aus lädt und startet. Alternativ kann der Debugger auch im System integriert sein und bei definierten Ereignissen, zum Beispiel durch eine Tastenkombination oder bei auftretenden Fehlern (Programmabsturz) aktiviert werden. Eine weitere Möglichkeit einen Debugger zu verwenden ist das Debuggen über eine serielle Schnittstelle. Dadurch kann der Debugger auf einem anderen Rechner als das zu untersuchende Programm laufen. Das hat den Vorteil, daß an Stelle des gesamten Debuggers nur ein Treiber für die serielle Schnittstelle auf dem Rechner mit dem zu untersuchenden Programm vorhanden sein muß.

2.2 Betriebssysteme

Um Betriebssysteme und Anwendungen vor fehlerhaften oder bösartigen Nutzerprogrammen zu schützen, verfügen Computer heutzutage über verschiedene Sicherheitskonzepte. Einen grundlegender Schutzmechanismus der von der Hardware unterstützt wird stellen Privilegierungsstufen dar, auf denen Programme ausgeführt werden können.

Zwei verschiedene Privilegstufen sind dabei notwendig und ausreichend, um Schutz zu gewähren. Das Betriebssystem als erstes gestartetes Programm läuft dabei auf der höchsten Privilegstufe und kontrolliert im Folgenden alle Anwendungen als zentraler Ressourcenverwalter. Nutzerprogramme laufen auf einer niederprioren Stufe, von welcher der Zugriff auf Prozeduren und Betriebsmittel der höheren Stufe nur über eine wohldefinierte, überwachte Schnittstelle möglich ist.

Auf der Intel x86-Plattform sind vier verschiedene Privilegstufen vorhanden. Diesen vier Stufen liegt die Überlegung zu Grunde, daß die Funktionalität eines Betriebssystemes in mehrere Teile getrennt werden kann. Betriebssysteme erbringen Dienste für Nutzerprogramme, welche über Systemaufrufe benutzt werden. Diese Dienstprozeduren verwenden wiederum Hilfsprozeduren, die gemeinsam benutzte Funktionen als Bibliothek zur Verfügung stellen. Die mögliche Aufteilung eines Betriebssystemes könnte also wie in Abbildung 1 aussehen.

Auf der am höchsten privilegierten Stufe läuft das Hauptprogramm, im folgenden als Kern bezeichnet, welches die Dienste aufruft. Aus Effektivitätsgründen (jeder Wechsel der Privilegstufen kostet Zeit) werden meist nur zwei Stufen, eine für das Betriebssystem und eine für Anwendungen, benutzt.

Es wird also zwischen Kern- und Nutzermodus unterschieden.

Ein Debugger, den man nutzt, um das Betriebssystem selbst zu Debuggen, benötigt zur Darstellung von Kerndaten Zugriff auf die Kernstrukturen und sollte sich deshalb auf der gleichen Privilegstufe

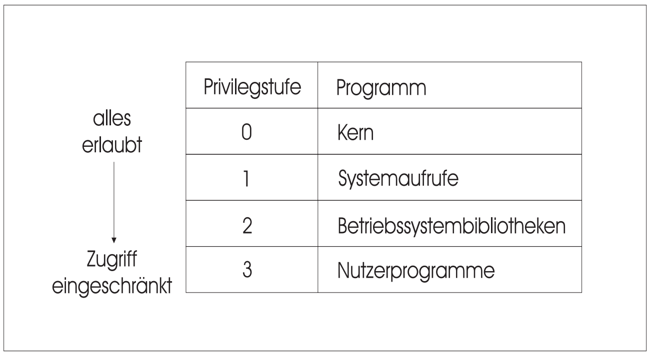

Abbildung 1: Aufteilung eines Betriebssystemes und Benutzung von Privilegstufen

wie der Kern selbst befinden.

Da die Kerndatenstrukturen betriebssystemspezifisch sind, ist es nötig, den Debugger an das System anzupassen. Deshalb werden diese Debugger auch als Kerndebugger bezeichnet.

Debugger für Anwendungsprogramme können aus diesem Grund nur teilweise zum Debuggen von Betriebssystemen genutzt werden. Während es beim Debuggen eines Anwendungsprogrammes in einem laufenden System keinen Grund gibt, in das Betriebssystem einzugreifen, stoppen Kerndebugger das System. Das ist notwendig, um konsistent Daten des Kernes zu untersuchen und damit den Kern selbst Debuggen zu können.

	Kerndebugger	Anwendungsdebugger
Datenstrukturen	Register, Seitentabellen, etc.	Nutzer-Register, Programmdaten
Speicherzugriff	gesamter	nur von Anwendung
Umgebung	Standalone	normale Anwendung
	integriert	später gestartet
Systemzustand	angehalten	laufend

Tabelle 1: Unterschiede zwischen Kern- und Anwendungsdebugger

2.3 Mikrokerne

Auch Betriebssysteme bleiben leider nicht von Programmierfehlern verschont. Mit ständig steigenden Anforderungen und Leistungsumfang (durch aufwendigere Benutzeroberflächen, Multimedia- und Netzwerkunterstützung) steigt die Komplexität und damit auch die Wahrscheinlichkeit von Programmierfehlern.

Hier soll das Mikrokern-Paradigma weiterhelfen, indem nur ein minimaler Betriebssytemteil im Kernel-Mode läuft und alle weitergehende Funktionalität von Komponenten (Server) im User-Mode erbracht wird. Durch Schutzmechanismen (siehe Abschnitt 2.2) kann verhindert werden, daß Anwendungen im User-Mode Schaden an Kernel-Mode Programmen verursachen. Dieser Systemaufbau verlagert das Problem zwar in erster Linie nur, da dadurch die Fehlerfreiheit der Server natürlich auch nicht gewährleistet werden kann. Der Vorteil liegt aber darin, daß auch wenn User-Level Betriebssystemteile versagen der Mikrokern stabil bleibt und nicht blockiert wird. Dieses Verhalten ermöglicht es immer noch kontrolliert zu reagieren, zum Beispiel Server zu beenden oder neu zu starten und damit einen totalen Systemausfall zu vermeiden.

Da selbst ein Mikrokern relativ komplex und in der Entwicklungsphase sicherlich nicht fehlerfrei ist, ist auch für diesen ein Debugger wünschenswert.

Im Folgenden werden die im Rahmen von DROPS relevanten Mikrokerne kurz beschrieben. Die Grundlage von DROPS bildet L4 - ein Mikrokern der 2. Generation. L4 unterscheidet sich von den Kernen der 1. Generation durch ein absolutes Minimum von Funktionalität im Kern. Da alle weiteren Funktionen von Servern erbracht werden, ist ein effizienter Kommunikationsmechanismus zwischen Adreßräumen Voraussetzung für ein performantes System. Dieser ist in L4 durch IPC (Interprozeskommunikation) realisiert.

Neben L4, welches nicht frei verfügbar ist, entstanden am IBM Watson das ebenfalls unter einer kommerziellen Lizenz stehende Lava Nucleus (LN) sowie an der Universität Karlsruhe L4KA, als Neuimplementation von L4.

Fiasco ist ein L4-kompatibler Mikrokern für die Intel Plattform, der zum größten Teil in C++ geschrieben ist. Ziel der Fiasco-Implementation ist es, im Gegensatz zu anderen L4-Implementationen, die meist in Assembler realisiert wurden, einen leicht wartbaren, freien (Fiasco steht unter der GPL-Lizenz) und im Quelltext leicht verständlichen Mikrokern zu erstellen.

3 Der Fiasco Kerndebugger

3.1 Entwurf

An Hand der Anforderungen, die an einen Debugger gestellt wurden, leitet sich ab, daß ein Debugger eine Sammlung von Funktionen (z. B. Breakpoints verwalten, Speicher anzeigen) darstellt. Diese sind voneinander wenig bis gar nicht abhängig. Neben der eigentlichen Funktionalität gibt es Ein- und Ausgabefunktionen sowie eine zentrale Instanz, in der die einzelnen Funktionen aufgerufen werden.

Der Debugger kann also in drei Teile aufgeteilt werden: die Benutzerschnittstelle, die Kommandoschleife sowie die einzelnen Funktionsmodule.

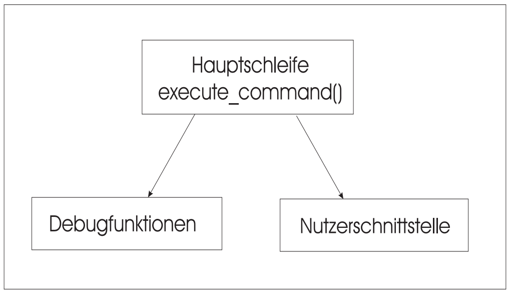

Abbildung 2: Struktur des Kerndebuggers

Die Nutzerschnittstelle des Fiasco Kerndebuggers ist an das Interface des LN Kerndebuggers angelehnt, was L4/LN Benutzern die Bedienung erleichtern soll.

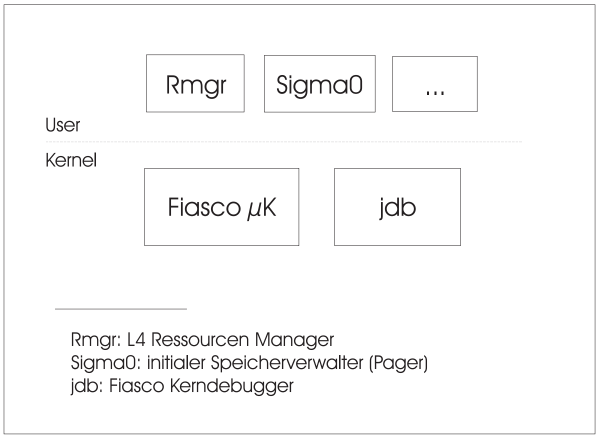

Abbildung 3: Systemaufbau

Wie im vorangegangenen Kapitel bereits erläutert, läuft der Kerndebugger auf der höchsten Privilegstufe als einziges Programm. Bei der Aktivierung des Debuggers wird der Rechner angehalten und der komplette Systemzustand gesichert. Alle Interrupts, einschließlich des Timer-Interrupts, werden gesperrt.

Der Kerndebugger benutzt also weder Funktionalität des Fiasco Kernes, noch andere Gerätetreiber. Daraus ergibt sich, daß der Debugger selbst Treiber für alle benötigten Geräte (minimal: Tastatur, Grafik) implementieren muß. Der Kerndebugger ist ein optionaler Teil von Fiasco. Das bedeutet, daß der Kerndebugger nicht erforderlich ist, um Fiasco zu benutzen (siehe Abbildung 3).

Im Folgenden wird ein Überblick über die verschiedenen Debugfunktionen und Entwurfsentscheidungen gegeben. Die Funktionen, die der Kerndebugger aufweisen sollte, orientieren sich an denen bekannter Debugger (wie: gdb[4]) und an Fiasco-spezifischen Eigenschaften (wie: Threadkontrollblöcke, IPC).

3.1.1 Speicherdarstellung

Die Anzeige von Speicherinhalten kann über physische oder virtuelle Adressen erfolgen. Virtueller Speicher ist dabei Prozessen (in Fiasco: Tasks) zugeordnet und kann zwischen zwei Prozessen unterschiedliche Inhalte aufweisen. Im Gegensatz dazu, ist die Darstellung des physischen Speichers zwischen allen Prozessen gleich.
Da alle Programme mit virtuellen Adressen arbeiten, erschien die Darstellung virtuellen Speichers am geeignetsten. Diese Entscheidung stellt keine Einschrän-kung dar, da der gesamte physische Speicher in Fiasco in den virtuellen Adreßraum eingeblendet wird (siehe Abbildung 4).

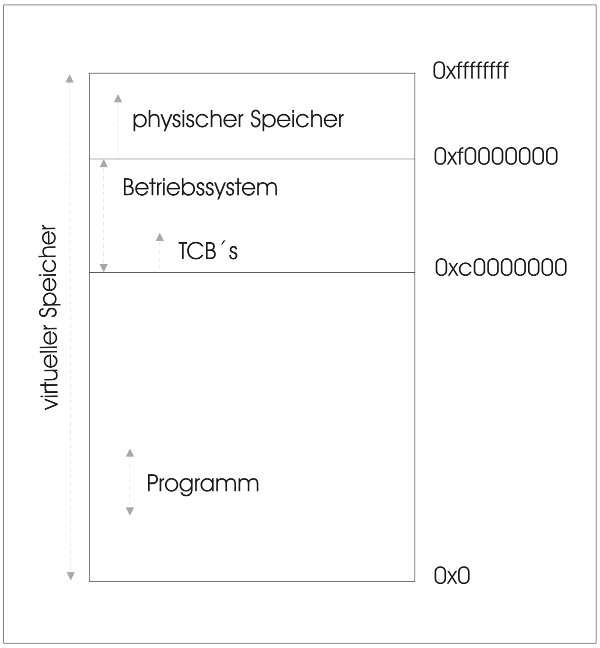

Abbildung 4: Speicherlayout unter Fiasco

Bei der Darstellung soll neben der Angabe einer Startadresse damit auch die Angabe der Task, deren Adreßraum dargestellt werden soll, möglich sein. Innerhalb der Darstellung soll es Möglichkeiten zur Navigation geben.

[4] auf Unix Systemen weit verbreiteter Debugger

3.1.2 Seitentabellen

Die Zuordnung von virtuellem zu physischem Speicher erfolgt durch sogenannte Seitentabellen. Diese realisieren den Adreßraum eines Prozesses. Bei einem Prozeßwechsel wird zur Seitentabelle des neuen Prozesses umgeschalten.
Daraus folgt, daß die Seitentabellen im Fiasco Kerndebugger pro Task anzeigbar sein sollen. Da Seitentabellen mehrstufig sind und Referenzen auf Speicherseiten darstellen, soll ein "Durchwandern" der Stufen bis zum referenzierten Speicherinhalt möglich sein.

3.1.3 Threadkontrollblöcke

Eine wichtige Datenstruktur stellt in Fiasco der Threadkontrollblock (Thread Control Block - TCB) dar, den jeder Thread besitzt. In ihm befinden sich alle Informationen über den Zustand des Threads, Zeiger auf andere TCB's sowie der Kernstack des Threads. Der Debugger soll die Threadkontrollblöcke eines beliebigen Threads anzeigen können.
Dargestellt werden relevante Registerinhalte sowie der (kommentierte) Zustand. Außerdem soll der Inhalt des Threadkontrollblocks (als Speicherauszug) angezeigt werden.

3.1.4 Ereignisverfolgung

Der Fiasco Kerndebugger soll es ermöglichen, auf bestimmte Ereignisse im Kern

zu reagieren. Ereignisse entsprechen dabei dem Aufruf von Kernfunktionen. Dabei sind vor allem Systemaufrufe von Interesse. Diese werden, wie in Kapitel 2.2 beschrieben, von Nutzerprogrammen ausgelöst und haben einen Privilegstufenwechsel vom Nutzermodus in den Kernmodus zur Folge.

Auf der Intel x86-Plattform wird dies realisiert, indem eine Anzahl von Ausnahmen (siehe Abschnitt 3.2) zur Verfügung steht, denen jeweils eine Behandlungsroutine (Handler) zugeordnet wird. Diese Zuordnung ist Aufgabe des Betriebssystemes, d. h. diese Routinen sind austauschbar.

Der Aufruf einer Log-Funktion kann erfolgen, indem nach Wechsel in den Kernmodus, aber vor Aufruf der Kernbehandlungsroutine eine Abfrage erfolgt:

```
if (log_this_event)
  log_event();
handle_event();
```

Diese Methode ist am einfachsten zu realisieren, allerdings verschlechtert sich durch die zusätzliche Abfrage einer Bedingung die Performance auch wenn die Ereignisverfolgung nicht aktiviert ist.

Besser ist es, zur Ereignisverfolgung die oben beschriebene Möglichkeit des Austauschs von Behandlungsroutinen zu benutzten. Wird die Ereignisverfolgung aktiviert, kann der Handler der Ausnahme durch die Log-Funktion ersetzt werden, die nach ihrer Abarbeitung die ursprüngliche Funktion aufruft. Damit sind keine Änderungen am Kerncode erforderlich und die Performance verschlechtert sich nicht.

Der Kerndebugger soll die Ereignisse Seitenfehler und IPC verfolgen können. Die Aktivierung der Ereignisverfolgung erfolgt durch die Kommandozeile. Es soll möglich sein, systembezogene Restriktionen für Ereignisse zu setzen, zum Beispiel die Beschränkung auf einen Thread oder einen Adreßbereich.

Neben der Möglichkeit, Ereignisse direkt auf dem Bildschirm anzuzeigen, ist es wünschenswert, Ereignisse auch nach ihrem Auftreten untersuchen zu können. Dazu wird ein sogenannter Tracebuffer verwendet. In ihm werden Ereignisse zusammen mit der Zeit ihres Auftretens sowie einer Ereignisnummer gespeichert. Bei der nächsten Aktivierung des Debuggers soll es möglich sein, diesen Buffer anzuzeigen.

3.1.5 Backtracing

Durch Untersuchung des Stackrahmens eines Threads ist es möglich, die Aufrufhierarchie eines Threads, das heißt Funktionen, die in einem Thread aufgerufen wurden und noch nicht beendet sind, darzustellen. Diese Möglichkeit wird von vielen Debuggern angeboten und ist auch als Backtrace bekannt.

Backtracing wird realisiert, indem die Befehlszeiger, beginnend vom Aktuellen (EIP Register) dargestellt werden und versucht wird, diesen Adressen Funktionen zuzuordnen. Voraussetzung dafür ist, das sogenannte Framepointer (EBP Register) benutzt werden, die korrekt miteinander verkettet sind.

Es soll nicht nur möglich sein, den Backtrace des gerade aktiven Threads, sondern den Backtrace eines beliebigen Threads zu untersuchen. Neben dem Nutzer-Stack eines Threads besitzt jeder Thread noch einen Kernelstack. Auch von diesem soll der Backtrace ermittelt werden können.

3.1.6 Mapping Datenbank

In L4-Systemen ist es möglich eine Speicherseite zwischen mehreren Prozessen gemeinsam zu benutzen (sharing). Mit dem L4-Systemaufruf *l4_fpage_unmap*, kann eine Seite, die ein Prozeß an einen anderen weitergegeben hat, wieder entzogen werden. Um diesen Systemaufruf zu realisieren ist es notwendig, Buch über die Weitergabe von Seiten (Mappings) zu führen. Dazu dient die Mapping Datenbank.

Der Debugger soll eine Visualisierug der Mapping-DB von Fiasco ermöglichen. Dargestellt werden alle Mappings einer physikalischen Seite.

3.1.7 Breakpoints, Watchpoints

Einen grundlegenden Mechanismus zum Debuggen stellen sogenannte Breakpoints dar. Nachdem ein Breakpoint gesetzt wurde, wird bei Erreichen der Breakpointadresse (Instruction Breakpoint) der Debugger aktiviert. Neben Instruction Breakpoints werden auch Breakpoints beim lesenden oder schreibenden Zugriff auf eine Speicheradresse sowie Ein-/Ausgabe Breakpoints unterstützt.

Neben herkömmlichen Breakpoints soll auch die Angabe von Bedingungen, unter denen ein Breakpoint gültig ist (sogenannte Watchpoints), möglich sein. Dabei kann festgelegt werden, daß ein Breakpoint nur durch einen bestimmten Thread ausgelöst wird. Weiterhin kann ein Wert bzw. ein Gültigkeitsintervall für ein Register oder eine Speicheradresse angegeben werden.

3.1.8 Symbole

Als Synonym für Adressen sollen die Funktionsnamen aus der Symboltabelle, welche beim Übersetzten des Kernes erzeugt wird, und die Namen der im Kern verwendeten Funktionen beinhaltet, verwendet werden können. Dazu ist es ausreichend, die Eingabefunktionen so zu erweitern, daß Symbole an Stelle von Adressen angegeben werden können.

3.2 Implementation

Bei der Implementation konnte auf einen rudimentär schon vorhandenen Kerndebugger aufgebaut werden. Die benötigten Ein- und Ausgabetreiber (Tastatur, Grafik, serielle Schnittstelle) wurden dem OSKit[5] entnommen.

Wie Fiasco selbst, ist der Kerndebugger in C++ implementiert. Der Zugriff des Debuggers auf (private) Kerndatenstrukturen wird erlaubt, indem der Debugger als *friend* der entsprechenden Klasse Zugriff auf ihre Daten erhält.

Wie im Kapitel “Entwurf” beschrieben, wurde versucht, die Benutzerschnittstelle von der Funktionalität des Debuggers zu trennen. Um dies zu erreichen, wurden Teile des Debuggers in eigenen Klassen oder Modulen implementiert, wenn die Nutzerschnittstelle von der Funktionalität trennbar war. Funktionen, die eigene Datenstrukturen umfassen wurden dabei als Klassen implementiert (siehe Abbildung 5).

Der Debugger wird immer durch eine Ausnahme in Form einer sogenannten Trap aufgerufen. Eine Ausnahme führt zur sofortigen Programmunterbrechung und dem Aufruf einer speziellen Prozedur, die auf die Ausnahme reagiert. Das Programm wird nach dem Befehl, der die Trap auslöst, unterbrochen. Die Programmausführung kann nach Behandlung der Ausnahme verlustfrei mit dem nächsten Maschinenbefehl fortgesetzt werden.
Nach Auftreten der Trap, rettet der Prozessor automatisch den gesamten Maschinenzustand (Trap state). Das hat zur Folge, daß für den Thread, der die Trap ausgelöst hat, alle Register bekannt sind.

3.2.1 Speicher- und Seitentabellenanzeige

Virtueller Speicher wird in der x86-Architektur durch die Verwendung von 2-stufigen Seitentabellen realisiert. Es existiert ein Seitentabellenbasisregister, welches bei einem Taskwechsel mit einem Zeiger auf die Seitentabellen der Task geladen wird. Dadurch wäre bei der Verwendung virtueller Adressen zur Speicherdarstellung diese auf den virtuellen Speicher der gerade aktiven Task beschränkt.

[5] Sammlung von Bibliotheken zur Unterstützung der Entwicklung von Betriebssystemen, siehe [8]

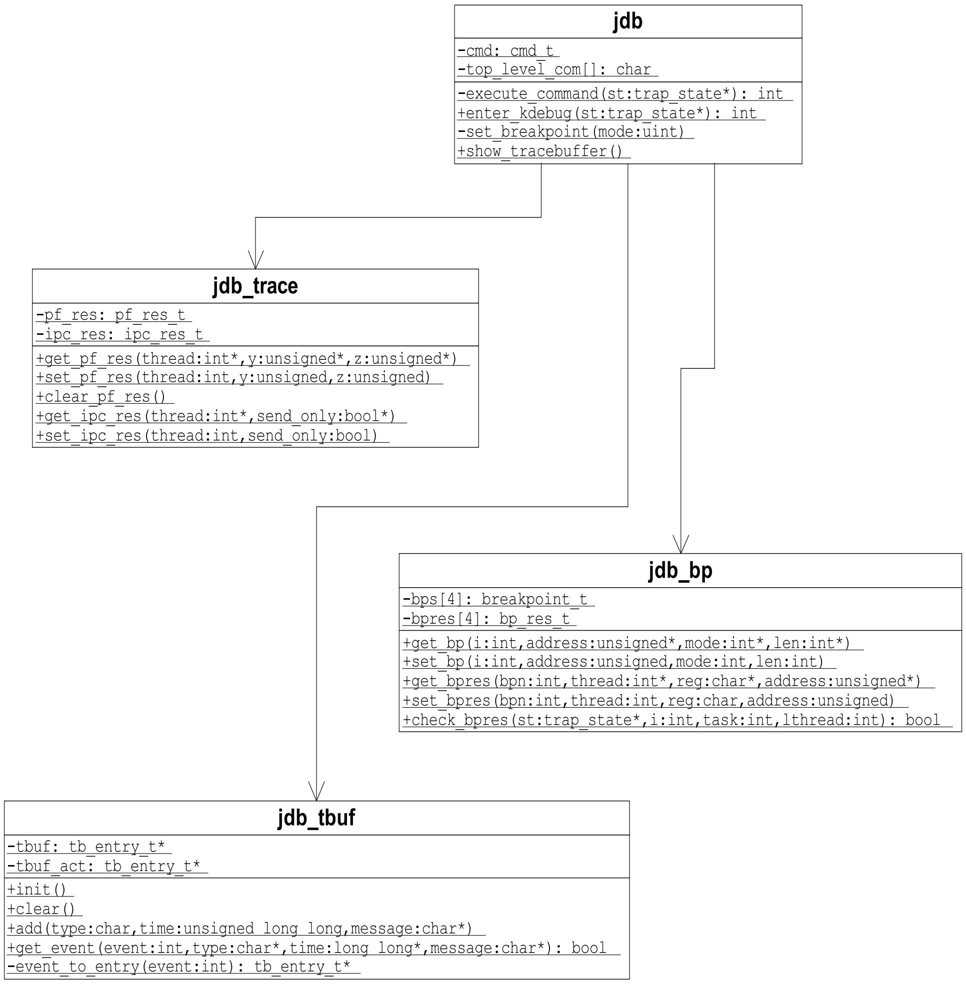

Abbildung 5: Klassendiagramm des Kerndebuggers (Funktionen und Parameter nicht vollständig)

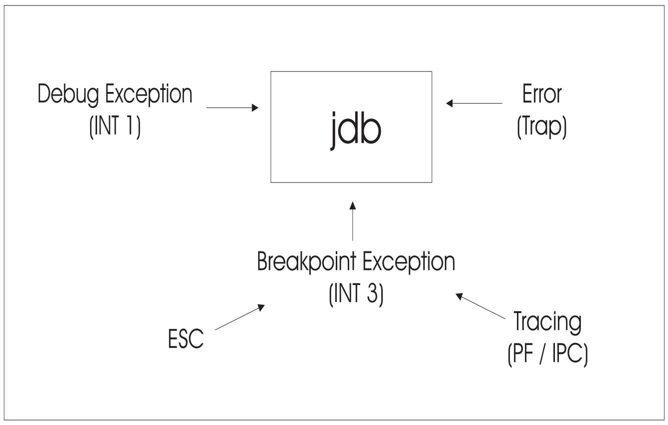

Abbildung 6: Aktivierung des Kerndebuggers

Da ein Kontextwechsel (switch context) nicht, beziehungsweise nur mit größerem Aufwand möglich ist, war es sinnvoller, zur Speicherdarstellung eine Umrechnung von virtuellen Adressen in physische vorzunehmen. Bei dieser Übersetzung war es leicht möglich, die Seitentabellen eines beliebigen Prozesses zu verwenden und damit Zugriff auf virtuellen Speicher aller Tasks zu erlangen.
Da die Seitentabellen selbst im Speicher liegen und Speicher- sowie Seitentabellenanzeige fast gleiche Funktionalität benötigen, erschien es sinnvoll, beide in einem Modul zu vereinheitlichen.
Dadurch ist es auch möglich, einen beliebigen Speicherbereich als Seitentabelle zu interpretieren.

Zusätzlich zur LN-Funktionalität ist es möglich, im physischen Speicher nach einem Wert zu suchen und Speicherinhalt zu verändern.

3.2.2 Threadkontrollblöcke

Bei der Ausgabe der Registerinhalte ist zu beachten, daß der Inhalt der sogenannten General-Purpose Register (EAX-EDX, ESI, EDI, EBP, ESP) nur vom aktuellen Thread bekannt ist. Für ihn werden bei der Aktivierung des Debuggers alle Register gerettet (Trap state). Andere, nicht aktive Threads, müssen (abhängig von ihrem Zustand) nicht alle ihre Register auf dem Stack gespeichert haben.

3.2.3 Ereignisverfolgung

Die Ereignisverfolgung (Tracing) dient der Überwachung von Systemaufrufen. Dabei werden ähnlich, wie bei Watchpoints, Bedingungen angegeben, unter denen der Debugger aktiviert wird. Die Ereignisverfolgung muß an der Kommandozeile aktiviert werden. Danach wird zum Beispiel bei Auftreten eines Seitenfehlers, *bevor* das Ereignis im Kern behandelt wird, das System angehalten und es besteht die Möglichkeit, den Debugger zu aktivieren. Nach Abarbeitung des Tracing-Codes erfolgt der Aufruf der ursprünglichen Funktion.

Bei der Implementation der Seitenfehlerüberwachung stellte sich heraus, das nach Austausch der Behandlungsroutine (siehe Abschnitt 3.1.4) und vor Aufruf der eigentlichen Log-Funktion noch verschiedene Low-Level Operationen (Register retten, etc.) nötig sind. Diese unterscheiden sich von denen ohne Ereignisverfolgung nur durch den Aufruf der Log-Funktion.

Die Implementation der IPC-Überwachung konnte eleganter durch die Verwendung von Funktionszeigern erfolgen. Da Fiasco die Funktionen für Systemaufrufe über eine Tabelle mit Funktionszeiger ermittelt, konnte statt Austausch der Ausnahmebehandlungsroutine dieser Zeiger benutzt werden. Dieses Vorgehen hat den Vorteil, daß keine "eigene" Low-Level Behandlung erforderlich ist, sondern bei Aktivierung der IPC-Überwachung einfach der Zeiger des IPC Systemaufrufes manipuliert werden kann. Die Änderungen am Kerncode fallen dadurch geringer aus als bei der Verwendung einer eigenen Behandlungsroutine.

Der Tracebuffer, in dem Ereignisse aufgezeichnet werden können, wurde als Ringbuffer implementiert (siehe Abbildung 7). Gespeichert werden, abhängig von seiner Größe, die letzten aufgetretenen Ereignisse.
Jedes Ereignis wird mit einem 64-Bit Zeitstempel versehen, wofür das Prozessor-interne Zeitstempelregister (TSC) benutzt wird.

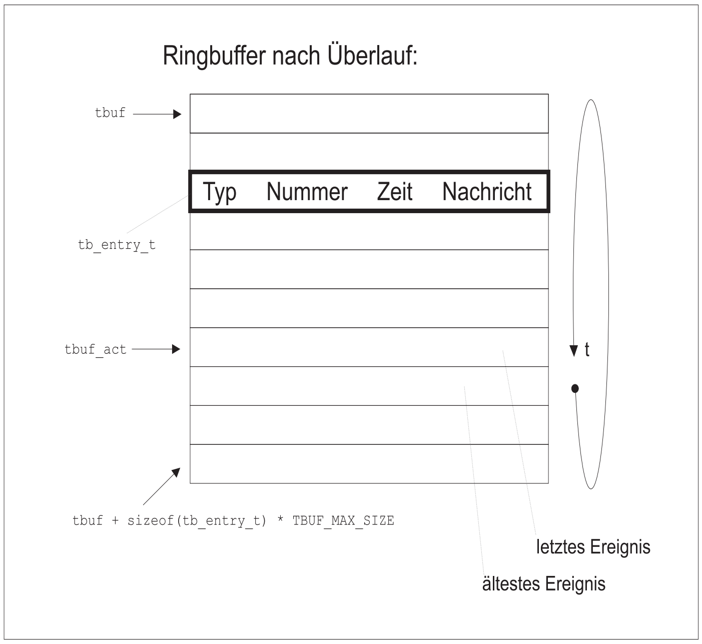

Abbildung 7: Aufbau des Tracebuffers

3.2.4 Mapping Datenbank

Verwendet wurde hier hauptsächlich Programmcode von Lukas Grützmacher, der die Mapping Datenbank implementiert hat [5]. Möglich ist die Anzeige aller Mappings für eine (physische) Seite oder die Angabe einer Adresse, die als Zeiger auf einen Eintrag der Mapping Datenbank interpretiert wird. Neben der virtuellen Adresse (und der zugehörigen Task) an die eine Seite gemappt ist, wird die Tiefe des Mappings (wie oft die Seite weitergegeben wurde) sowie die Seitengröße angezeigt.

3.2.5 Symbole

Zur Zeit werden nur die Symbole des Fiasco-Kerns unterstützt. Dazu muß die Symboltabelle in den Speicher geladen werden. Ihre Adresse wird durch den L4 Ressourcen Manager (Rmgr) in die Kernel Info Page[6] eingetragen.

3.2.6 Breakpoints, Watchpoints

Um Breakpoints effizient implementieren zu können, ist Hardwareunterstützung notwendig. Auf Intel Pentium Prozessoren werden derzeit vier Hardwarebreakpoints unterstützt.

Watchpoints wurden, als um Bedingungen erweiterte Breakpoints, implementiert. Da sie nicht durch Hardware unterstützt werden, verschlechtert sich bei Benutzung die Performance, was jedoch nicht

[6]Speicherseite, die Informationen über den Kern enthält (wie: Versionsnummern, Parameter)

zu vermeiden ist. Die Watchpoints sind mit den Breakpoints gekoppelt, das heißt nur wenn ein Breakpoint auftritt, werden die Restriktionen untersucht. Watchpoints ohne (Hardware-) Breakpoints zu unterstützen, ist nur mit Hilfe von Single-Stepping (Einzelschrittmodus) möglich, dadurch aber sehr performancelastig und wurde deshalb als nicht sinnvoll erachtet.

3.2.7 Backtracing

Um den Nutzer-Backtrace anzuzeigen, wird der Wert des EBP Registers, des entsprechenden Threads sowie der aktuelle Befehlszeiger (EIP) benötigt.
Für den aktuellen Thread, welcher vor der Aktivierung des Debuggers lief, sind diese Register bekannt. Vor Eintritt in den Kern schreiben Threads ihre Arbeitsregister (CS + EIP, SS + ESP, EFlags) auf ihren Kernstack zurück. Da alle Threads die nicht aktiv sind sich im Kern befinden, ist es möglich, den Befehlszeiger eines beliebigen Threads zu ermitteln.

Schwieriger ist es, die Position des Framepointers (EBP) herauszufinden, da dieser nicht an einem festen Platz gespeichert wird, sondern sich abhängig vom Threadzustand irgendwo auf dem Stack befindet. Dieses Verhalten entsteht durch die Benutzung des EBP-Registers als Parameter für Systemaufrufe.

Für das Kern-Backtracing gilt, daß EIP und EBP am Anfang des aktuellen Kernstacks gespeichert sind. Problematisch ist, daß diese Framepointer während eines Systemaufrufs nicht korrekt verkettet sind. Dadurch wird die Darstellung der Kernbefehlszeiger abgebrochen, wenn ein ungültiger Framepointer gefunden wird.
Liegen die ermittelten Adressen im Bereich des Kernes, so wird die Adresse mit Hilfe der Symboltabelle um den Funktionsnamen ergänzt.

A Dokumentation

Als Dokumentation der Benutzerschnittstelle des Debuggers entstand das Fiasco Kerndebugger Manual, auf Grundlage des LN Kerndebugger Manuals [3].

B Glossar

Adreßraum: Schutzmechanismus, jeder Prozeß hat seinen eigenen Adreßraum und kann damit nicht auf andere Adreßräume zugreifen

Ausnahme (Exception): vom Prozessor unterstützte Unterbrechung, stopt die Ausführung und ruft eine spezielle Prozedur auf, die auf die Ausnahme reagiert

Debugger: Programm zur Fehlersuche und Analyse

Framepointer: Inhalt des EBP-Registers, ergibt mit Stackpointer (ESP) zusammen einen Stackframe (obere und untere Grenze des aktuellen Stacks)

Haltepunkt (Breakpoint): Unterbrechungsbedingung, die ein Programm stopt, wenn sie erfüllt ist

Interrupt: siehe Ausnahme

IPC: Inter Process Communication - grundlegender (und einziger) Kommunikationsmechanismus von L4, Kommunikations-Infrastruktur für Nutzerprogramme um untereinander und mit dem Kern zu kommunizieren, wird für die Kommunikation über Adreßraumgrenzen genutzt

Kernmodus, Kernel mode: höchste Privilegstufe, uneingeschränkter Zugriff auf alle Betriebsmittel, nur das Betriebssystem sollte auf dieser Stufe laufen

Kerndebugger: an ein Betriebssystem angepaßter Debugger, der dessen Datenstrukturen kennt

Mikrokern: realisiert ein minimales Betriebssystem, stellt eine Schnittstelle (API) zur Verfügung die die Auslagerung weiterer Funktionalität in User-Level Server ermöglicht

L4, L4KA, LavaNucleus: L4-kompatible Mikrokerne, die die L4 API implementieren

Mapping: bezeichnet: ein Eintrag in einer Seitentabelle (Zuordnung von virtuellem Speicher zu physischem) - oder - ein Eintrag in der Mapping Datenbank

Mapping Datenbank: speichert alle Mappings von Seiten, notwendig um den L4-Systemaufruf fpage_unmap zu realisieren

Prozeß (Task): in L4 ein Adreßraum, in dem mindestens ein Thread vorhanden ist

Rmgr: L4 Ressourcen Manager, verwaltet Betriebsmittel

Seite (Page): Verwaltungseinheit des virtuellen Speichers

Seitenfehler (Pagefault): Mechanismus durch den eine virtuelle Seite eine physische Speicherseite zugeordnet bekommt oder ein unerlaubter Speicherzugriff

Sigma0: initialer Pager, behandelt Seitenfehler

Symbole: symbolische Namen für Adressen, z. B. Funktionsnamen

Symboltabelle: enthält alle Symbole eines Programmes

Systemaufruf: Schnittstelle zum Aufruf von Kernfunktionen durch Nutzerprogramme

Thread: kleinste Einheit eines Programmes, ein Ablaufpfad, mehrere Threads können in einer Task auf einen gemeinsamen Adreßraum zugreifen

Threadkontrollblock (TCB): beinhaltet alle Informationen über einen Thread

Trap: siehe Ausnahme, Prozessor speichert automatisch den gesamten Zustand und restauriert diesen nach Behandlung der Trap

Tracing: Verfolgen von Ereignissen

User mode, User level: niedrigste Privilegsstufe, Nutzerprogramme sollten auf dieser Stufe laufen

Watchpoint: um Bedingungen (z. B. Registerinhalt) erweiterter Breakpoint

Literatur

[1] Andrew S. Tanenbaum, *Moderne Betriebssysteme*, Prentice Hall, 1995

[2] Michael Hohmuth, *The Fiasco Kernel: System Architecture*, 2000

[3] Jochen Liedtke, *LN Kdebug Manual*, 1997

[4] Jochen Liedtke, *L4 Reference Manual*, 1996

[5] Lukas Grützmacher, *Mapping Datenbank für FIASCO*, 1998

[6] Bjarne Stroustrup, *Die C++ Programmiersprache*, Addison-Wesley, 1998

[7] Hans-Peter Messmer, *PC-Hardwarebuch*, Addison-Wesley, 1995

[8] Bryan Ford and Flux Project Members, *The Flux Operating System Toolkit*
http://www.cs.utah.edu/flux/oskit

[9] Intel: Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture
http://developer.intel.com/design/PentiumIII/manuals

[10] Intel: Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference
http://developer.intel.com/design/PentiumIII/manuals

[11] Intel: Intel Architecture Software Developer's Manual, Volume 3: System Programming
http://developer.intel.com/design/PentiumIII/manuals