Flux: Rekursive Virtuelle Maschinen

Hauptseminar im WS 1997/98

Bearbeitet von: Annamaria Christina Andres

Zusammenfassung

Es wird eine neue, von der Utah Universität entwickelte Methode für modulare und erweitbare Betriebssystemen vorgestellt.

Die Entwickler haben versucht, die Eigenschaften einer virtuellen Maschinen und eines Mikrokerns im eigenen System zu übernehmen. Um verstehen zu können wie sich dieses Konzept von den vorherigen unterscheidet, wird anhand des von IBM entwickelte System VM/370 beschrieben, wie eine traditionelle virtuelle Maschine funktioniert. Da die virtuelle Maschine von Java auch sehr bekannt ist, wird der Unterschied der letzten zwei genannten kurz erläutert. Im zweiten Abschnitt, werden die Ziele von Flux und die notwendigen Eigenschaften diese Ziele zu erreichen vorgestellt.Im drritten Abschnitt folgt die Beschreibung des Aufbaus von Flux zusammen mir den Unterschieden zu der traditionellen virtuellen Maschine. Durch einige Beispiele wird das Konzept veranschaulicht.

1 Die traditonelle virtuelle Maschine

Die traditionelle virtuelle Maschine wird anhand des von IBM entwickelten Systems VM/370 erklärt.

"Eine virtuelle Maschine ist eine Illusion der wirklichen Maschine "[2]. Auf einer wirklichen Maschine läuft ein Programm, das durch Exportieren der Hardwarearchitektur mehrere parallel laufende Maschinen mit derselben oder sogar mit höherer Kapazität als die der wirklichen Maschine laufen lassen kann. Dieses Programm heißt *virtual machine monitor.* Es gibt aber auch virtuelle Maschinen, die softwarebasiert sind, da sie nicht durch das Exportieren von Hardware enstehen, sondern von einem Programm implementiert werden. In diesem Fall, heißt das Programm *Nester.* Alle Prozesse und Befehle der virtuellen Maschine werden simuliert. Wenn eine virtuelle Maschine eine andere laufen lassen kann, dann heißen diese ineinandergeschachtelten virtuellen Maschinen rekursive virtuelle Maschinen. Für das Funktionieren des Systems und Entstehen der virtuellen Maschinen ist im VM/370 der *Control Program* zuständig. Dieses Thema wird weiter unten ausführlich untersucht. Eines der Ziele, die durch virtuelle Maschinen erreicht werden soll, ist die Benutzung einer Maschine gleichzeitig von mehreren Benutzern, so daß diese den Eindruck haben, allein alle Ressourcen einer Maschine zur Verfügung zu haben. Deswegen können die virtuellen Maschinen nur einen einzigen Benutzer zu einem gewissen Zeitpunkt haben.

1.1 Das Control Program (CP)

Das Control Program (CP) ist der wichtigste Bestandteil. Es erstellt die Umgebung, in welcher die virtuellen Maschinen laufen können.

Das CP läuft auf der wirklichen Maschine im *supervisor state* und hat daher Zugang zu allen privilegierten Maschinenbefehle. Es enthält in einem Hauptblock für jede virtuelle Maschine die virtuellen Register, den Zustand (supervisor oder problem/user) und den Befehlszähler. Alle virtuellen Maschinen werden im *problem/user state* der realen Maschine laufen gelassen. Jedes Mal wenn die virtuelle Maschine einen privilegierten Befehl ausführen will, erzeugt sie einen Interrupt, der von der CP aufgenommen wird und weiterverarbeitet wird. Diese Interrupts sind der Schlüssel der Kommunikation zwischen dem CP und der virtuellen Maschine. Das CP schaut nach, ob die virtuelle Maschine sich im supervisor state befindet, wenn ja, simuliert es das Durchführen des privilegierten Befehls, ansonsten (d.h. die virtuelle Maschine befindet sich im problem state) wird das generierte Exeption-Signal weitergeleitet. Das CP schaut auch nach, was die virtuelle Maschine braucht und ermöglicht die Durchführung. Zum Beispiel, falls ein Ein/Ausgabe Befehl durchgeführt werden muß, erstellt es die notwendigeVerbindung zwischen den virtuellen und den wirklichen Ein/Ausgabe Mechanismen.Das alles verhält sich so als ob der ganze Prozeß auf der virtuellen Maschine abgelaufen wäre.

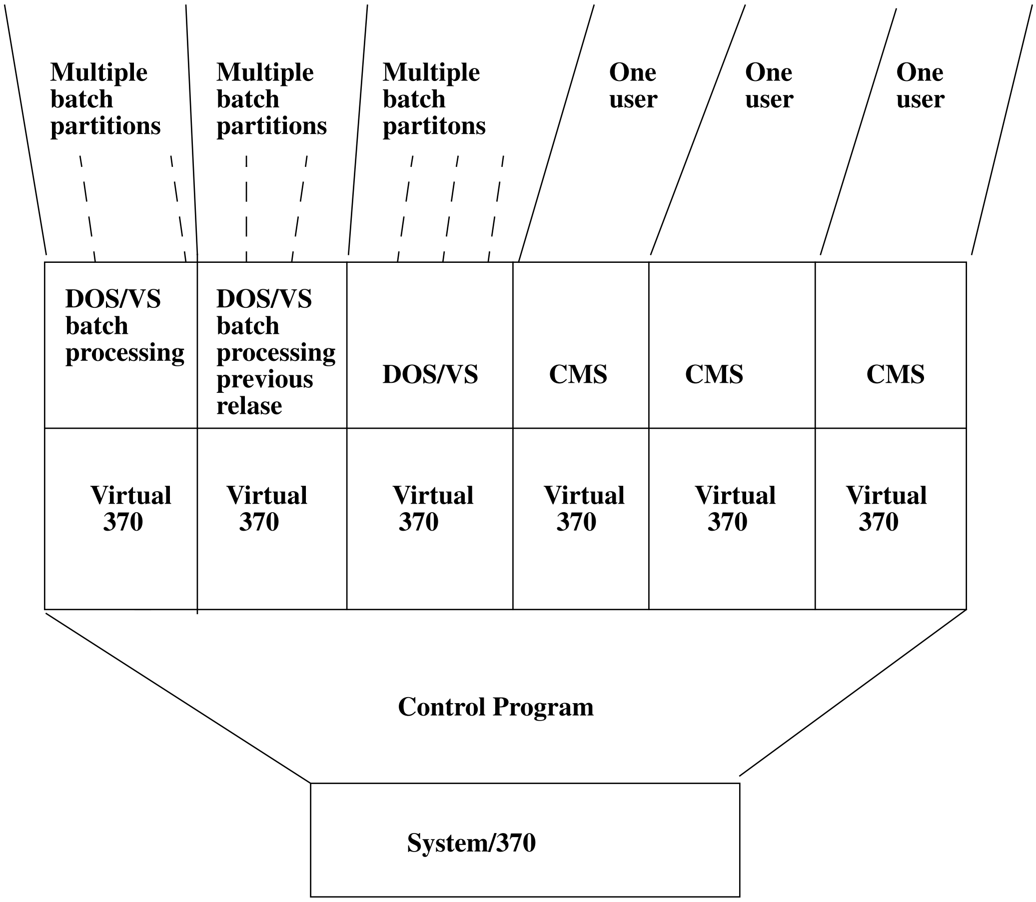

Abb. 1.1 Entstehen von virtuellen Maschinen

1.2 Funktionen von CP

- **Demand Paging.** CP hat nur einen kleinen Anteil seines Speichers im reellen Speicher, der Rest, wird "gepaged". Die Konsequenzen, sind sehr interessant, besonders wenn auf der virtuellen Maschine ein Betriebsystem läuft das selber paged. Dann haben wir zwei Stufen von paging und so entstehen Verzögerungen im "paging enviroment". Die erste Stufe wird von CP wie normale Ein/Ausgaben behandelt und die von CP erzeugte ist logisch transparent für die virtuelle Maschine dargestellt. Die beste Möglichkeit diese Probleme zu entfernen ist durch *hand shaking.*
- **Minidisks.** DASD (Direct Acces Storage Device) kann sowohl an einer einzigen, wie auch mehreren virtuellen Maschinen zugeordnet werden. CP teilt die Platte in mehrere Miniplatten (Minidisks) ein und sorgt dafür, daß sie eindeutig auf je eine virtuelle Maschine verteilt werden. Eine virtuelle Maschine darf nicht auf den Speicherplatz einer anderen zugreifen, daher auch nicht auf der anderen Minidisk. Die virtuellen Zylindernummern sind verschieden von den eigentlichen. Wenn zum Beispiel eine virtuelle Maschine die virtuelle Nunerierung von 0 .. 300 hat, kann es sein, daß sie in Wirklichkeit auf den Nummern 160 .. 360

läuft. Die Speicherplatzverteilung auf jeder Minidisk wird von dem gerade auf der virtuellen Maschine laufenden Betriebsystem bestimmt, CP teilt nur die Platte auf und überwacht die virtuellen Maschinen, daß nicht auf dieMinidisk einer anderen zugegriffen wird.

- **Console Management.** Jede virtuelle Maschine hat eine virtuelle Konsole. So können die Benutzer der virtuellen Maschine genauso handeln, als ob sie mit einer physikalischen zu tun hätten.
- **CP User Privilege Clases.** Da eine VM im Normalfall mehrere Benutzer hat, ist es nicht möglich, daß sie dieselben Zugriffsrechte haben. Daher wird jeder Benutzer einer *CP User Privilege Class* zugeordnet. Es kann genau bestimmt werden was welchem Benutzer erlaubt ist, da er mit Benutzername und Passwort identifiziert werden kann und damit einer bestimmten Klasse zugeordet wird. CP selbst wird auch als Benutzer betrachtet und gehört der *supervisor class* an, so daß es überall anerkannt wird.
- **VM Directory.** VM Directory ist eine Datei, die alle nützlichen Beschreibungen für alle möglich definierbaren virtuellen Maschinen enthält. Sie besteht aus dem Benutzernamen, Passwort und Klasse eines Benutzers, wie auch anderen Daten die die virtuelle Maschine des Benutzers definieren

1.3 Eigenschaften virtueller Maschinen

In diesem Abschnitt werden ein paar der Eigenschaften der virtuellen Maschinen beschrieben, wie auch die Unterschiede des bis jetzt beschriebenen Systems zur Java Virtual Machine (JVM).

(1) Es wird ermöglicht, daß mehrere Benutzer auf eine Maschine zugreifen können, und den Eindruck haben eine Maschine alleine selber zu bedienen;

(2) Es können unterschiedliche Betriebsysteme gleichzeitig auf einer Maschine laufen, ohne daß sie einander beeinträchtigen. Das ist besonders nützlich, wenn eines der Systeme noch unter Testlauf steht, oder wenn man gerne auch ältere Versionen behalten möchte, um nicht die alten Programme aktualisieren zu müssen;

(3) Keine virtuelle Maschine kann auf dem Speicherbereich der anderen zugreifen, das CP überwacht die Zugriffe aller virtuellen Maschinen;

(4) Schutz der Daten wird gesichert, durch Abfrage des Passworts beim Zugang ins System, wie auch beim Zugreifen auf verteilte Diskdateien;

(5) Ein ungewöhnlicher Endzustand einer virtuellen Maschiene wird vom CP bemerkt und teilt ihr und ihren Benutzern die Wirkungen dieses Zustandes mit.

Der wesentliche Unterschied zwischen dem vom IBM entwickelten System und der JVM besteht darin, daß der Java Intepreter ein Programm ist, der eine virtuelle Maschine implementiert, gesamt der Hardwarearchitektur auf der sie läuft. Dadurch, das die JVM alles simuliert wird (auch die normalen Befehle), müssen auch alle Befehle emuliert werden, im Gegensatz zu VM/370, wo nur die privilegierten Befehle emuliert werden müssen. Wenn man nicht das umdefinieren der Architektur braucht, ist die IBM-Version effizienter, weil die meisten Befehle direkt auf der reellen Maschine ausgeführt werden können, ohne mit Hilfe von Software erzeugt zu werden.

2 Ansätze/Motivation für Flux

In diesem Absatz wird erkärt, was die Entwickler von Flux dazu gebracht hat, eine neue Art von System zu entwerfen das virtuelle Maschinen erzeugen kann und wo sie die Schwerpunkte bei der Entwicklung gesetzt haben. Zuerst werden die Nachteile der hardwarebasierten virtuellen Maschinen vorgestellt und hinterher die Eigenschaften die bei der Entwicklung von Flux im Zentrum standen. Das Hauptziel ist, die höchstmögliche Leistung auf jeder Stufe im Inneren der virtuellen Maschine zu erhalten

2.1 Die Nachteile der traditionellen viruelle Maschine

Die Entwickler von Flux, haben als zentrale Eigenschaft ihres Systems nicht die Vervielfachung der Hardwarearchitektur betrachtet, sondern die Modularität, Flexibilität und Erweiterbarkeit des Systems. Die bis jetzt vorgestellte Maschine hat aus diesem Standpunkt einige Nachteile:

(1) Es ist nicht mit vertretbarem Aufwand möglich, alle Befehle gewisser CPUs in der virtuellen Maschine zur Verfügung zu stellen. Zum Beispiel kann es “Problem Mode” Befehle geben, die es ermöglichen, die Prioritätsebene zu ermitteln. Zur Simulation müßte ebenso der Problem Mode simuliert werden.

(2) in jeder Rekursionstufe verringert sich die Leistung, weil auf jedem Niveau alle privilegierten Befehle des höheren empfangen und emuliert werden müssen. Auf der n-ten Stufe, muss der Befehl 2^{n-1} mal wiederholt werden.

(3) Es besteht nicht die Möglichkeit, daß Großvater- und Enkelprozeß miteinander kommunizieren, dies ist auf der Ebene Vater - Sohnprozeß beschränkt. Diese fehlende Eigenschaft wird *short circuit* genannt und ist bei der Enwicklung von Flux gefordert.

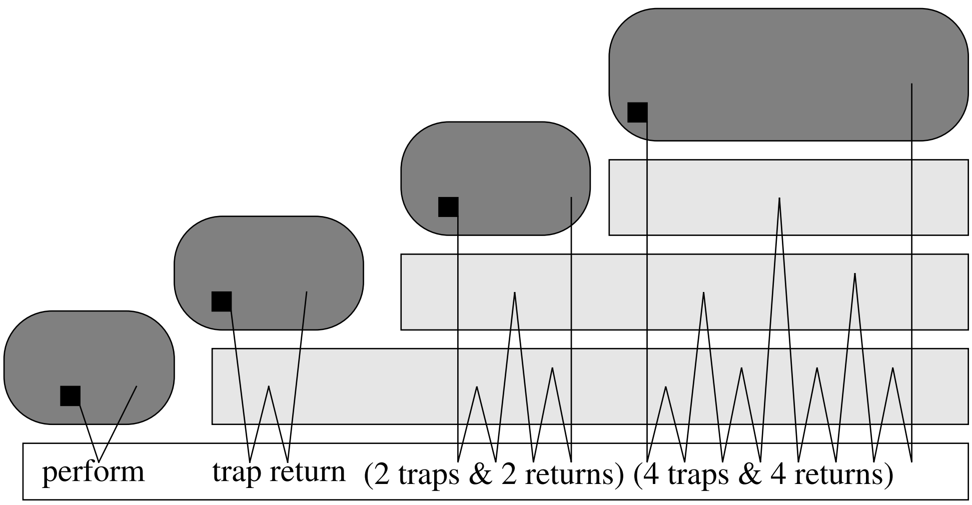

Abb. 2.1 Exponentieller Slowdown in der traditionellen virtuellen Maschine

Rekursive_Virtuelle_Maschinen: 02.12.97 20:22

0

2.2 Die zentralen Eigenschaften von Flux

Wegen den obengennanten Nachteile ist Flux nicht so wie die bis jetzt beschriebenen virtuellen Maschine aufgebaut, es bewahrt nur die nützlichen Eigenschaften. Damit ist gemeint, daß die exponentiellen Kosten auf jeder Stufe der Rekursivität vermieden werden, indem in jeder Stufe die ganze Umgebung der übergeordneten Stufe simuliert wird, und zwar mit allen Befehlen, Ein/Ausgabe Geräte und dem Zugriff auf dem Speicher. Dazu sind zwei Eigenschaften von zentraler Bedeutung: *State encapsulation* und *border control* .

2.2.1 State Encapsulation

State Encapsulation bedeutet die Fähigkeit einen Prozess in einem anderen so einzuschließen, daß der ganze Zustand des Sohnprozesses und dessen logischer Nachfolger eine Untermenge vom Zustand des Vaterprozesses sind. Das gibt ihm die Möglichkeit, seinen Sohn, wie eine Anhäufung von Daten zu betrachten und ihn auch dementsprechend zu behandeln.

Es gibt Systeme, in welchen die Lebensdauer der Söhne unabhängig von der der Vaterprozesse ist (z.B. in Unix). Das ist aber bei virtuellen Maschinen nicht möglich, da eine virtuelle Maschine ohne ihren Simulator nicht existieren kann, weil der Zustand der virtuellen Maschine ein Teil der Variablen des Simulatorprogramms ist. Deswegen muß es dem Vaterprozeß erlaubt sein, seine Söhne zerstören zu dürfen, gesamt ihrer Recourcen. Diese Fähigkeit wird hierarchischer Ressourcemanagement genannt.

Ein weiterer Aspekt ist die Fähigkeit den Zustand aller Unterprozesse identifizieren zu können (*state visibility*). Das ist verlangt für alle Anwendungen wie “debugging” oder “checkpointing” und für das letztere ist desweiteren notwendig, alle Zustände des Sohnprozesses zu kennen.

Letztens ist es wichtig, daß die Referenzen relativ sind. Damit sind sowohl die innerhalb eines Prozesses od. Vaterprozesses (*innere Referenzen*) gemeint, wie auch die, die auf ein anderes Objekt zeigen (*äußere Referenzen*). Die Relativität wird verlangt, wegen den Rechten des Vaterprozesses seiner Söhne gegenüber. Er darf sie wie normale Daten behandeln und normale Daten kann man ja von einer Stelle in einer anderen umsetzen. Der Vorteil dieser Fähigkeit ist, daß eine virtuelle Maschine jederzeit in einer anderen Umgebung eingesetzt werden kann und alle ihre Referenzen gültig bleiben.

2.2.2 Border Control

Die sogenannte *border control* Eigenschaft beruht auf Vater - Söhneprozesse Beziehungen. Alle Aktionen des Sohnes laufen über den Vater - er weiß bescheid, was sein Sohn tun will, und nur wenn er es erlaubt, kann dies auch ausgeführt werden. Diese Eigeschaft, zusammen mit der State Encapsulation sind besonders wichtig in Anwendungen, wie Sicherheitsmonitore, die totalen border control verlangen. Sie müssen die Fähigkeit besitzen die Aktivitäten aller Söhne und derer Nachfolger kontollieren zu können.

3 Aufbau von Flux

Wie schon erwähnt, unterscheidet sich Flux teilwese von den traditionellen Maschinen auch vom Aufbau her. Es basiert auf einer Softwarearchitektur (Fluke genannt), und besteht aus drei wesentlichen Teile:

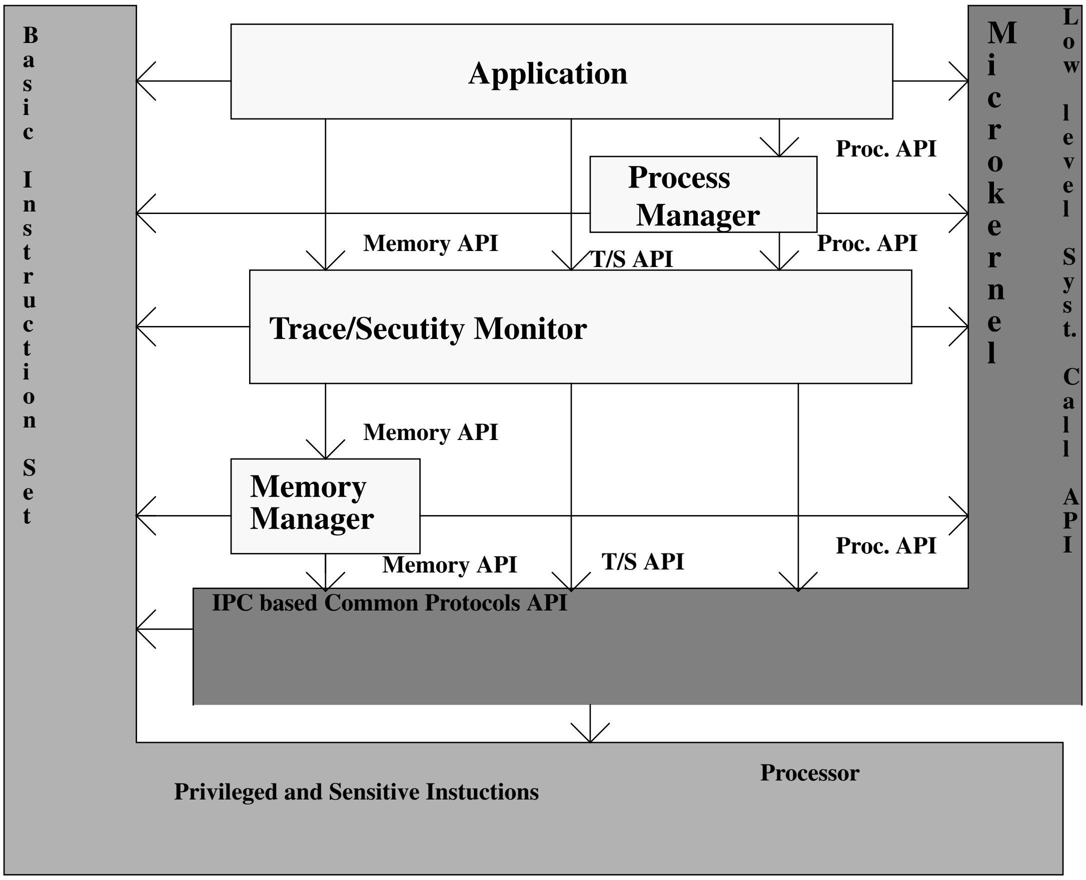

Abb. 3 Der dreiteilige Aufbau von Flux

3.1 Der Prozessor

Die CPU ist das am einfachsten gebaute Bestandteil des Systems. Die Konstruktion erlaubt allen Prozessen ohne Befehlsemulation zu laufen, ein sehr wichtiger Punkt, weil ansonsten der bei der Java virtuellen Maschine erwähnte Nachteil nicht zu vermeiden ist. Die Konstruktion beruht auf der Übernahme einer wohldefinierten Untermenge der x86 Architekur, so daß es möglich ist, diesen Punkt einzuhalten. Das heißt, alle unprivilegierte Instuktionen, die eine globale Auswirkung haben können, werden aus dieser Untermenge ausgeschlossen (z.B CPUID).

3.2 Der Mikrokern

Der Mikrokern ist der interessanteste Teil des vorgestellten Systems. Er läuft direkt auf einer x86 Architekur, ohne daß die Befehle emuliert werden müssen. Die Implementation sorgt dafür, daß es portabel und flexibel ist. Das meiste ist in rechnerunabhängigem C geschrieben und übernimmt die Vorteile vom Aufbau des *Flux OS Toolkit (Fluke* genannt*),* die später kurz erwähnt werden. Der Mikrokern implemeniert die *low-level API*, und initialisiert nur die *high level API (Common Protocol API).* Die letzteren werden im nächsten Abschnitt beschieben.

3.2.1 Fluke.

Flux OS Toolkit, ein "Baukasten" zur erstellung von Betriebsystemen. Es liefert eine Menge von Komponenten, mit denen diese gebaut können. Diese Komponenten sind modular implementiert, unabhängig voneinander und sehr gut dokumentiert. Auch wenn Fluke ursprünglich auf x86 Architekur läuft, ist es auf andere portierbar, da seine Schnittstellen architekturunabhängig sind. Die x86 spezifischen Implementationen sind in einer einzigen Komponente eingeschlossen, in der Low Level Unterstützungsbibliothek (support library).

Auf den low level Mechanismen werden high level Anwendungen aufgebaut, die architektur - unabhängig sind und deren Hauptziel die Implementation maximaler Flexibilität ist. Die Teile die am meisten Arbeit verlangt haben, waren die, die für ein Betriebsystemprogrammierer uniteressant sind und daher oberflächtlich implementiert werden.Daher sind diese für die nächste Version meistens unbenutzbar sind. Fluke wurden auch diese Teile sorgfältig programmiert, mit klar definierten Schnittstellen und reichlich dokumentiert. Ansonsten, wurden natürlich auch Teile von schon vorhandenen Systeme übernommen und nur erweitert.

Ein gutes Beispiel für die letzteren sind die Linux Drivers, FreeBSD Networking oder NetBSD File System. Diese Teile sind jeweils in einem Fluke Modul eingeschlossen, das die Bedienung dieser vereinfacht. Zum Beispiel bei den Linux Drivers werden die Einzelheiten des ursprünglichen Codes versteckt, und an deren Stellen gibt es einfachdefinierte Gerätschnittstellen. Diese können außerdem ohne weiteren Aufwand erweitert werden. Die weiteren Module werden jetzt nur aufgezählt, bei weiterem Interesse sollte man den Artikel darüber lesen[3]. Außer den bis jetzt erwähnten Bestandteilen, gibt es noch elf weitere Module. Zwei für der Unterstützung des Speichermanagementes, die list-based memory manager (LMM) für direkte und einfache Speicherallokierung von physikalischem oder virtuellem Speicher und die "adress map manager" (AMM) für die anderen Arten von Speichermöglichkeiten. Es gibt noch eine minimale C-Bibliothek, ein Modul für der Initialisierung und das Starten von mehreren Prozessen im Symetrik Multiprocessing (SMP). Es gibt noch Module zur Unterstützung der Partitionierung (Partitioning), ein Exec und ein Libm (mathematische Bilbiothek) Modul. Diese Module muß man natürlich nicht alle verwenden, wenn man "OS Toolkit" benutzt. Dieser wurde verwendet, um den Mikrokern von Flux zu implementieren, indem die meisten Bauteile übernommen wurden.

3.2.2 Low Level API.

Das low level API sind eine Menge von Mikrokernobjekte und Systemaufrufe. Diese sind alle vom Mikrokern implementiert, damit die Eigenschaften aus dem zweiten Kapitel erhalten bleiben. Die aufwendige Implementierung lohnt sich aber, da die Vaterprozesse dadurch nur noch die Sohnprozeßsystemaufrufe überwachen muss.

Die Speicherverwaltung ist sehr effizient implementiert. Es gibt keine Funktionen, die Speicher allokieren, der Speicher kann nur von einem Prozeß zu einem anderen (Vater- Sohn Prozesse) remapt werden. Das heißt, wenn der Sohnprozeß Speicher braucht, teilt er es dem Vater mit und der Vater gibt ihm welchen von seinem ab. Der Vaterprozeß kann kontrollieren, was mit diesem Speicherbereich geschieht, indem er z.B. seinem Sohn nur Leserechte darauf gibt. Wenn der Vaterprozeß den Speicher zurück haben will, kann er ihn zu jedem beliebigem Zeitpunkt von seinem Sohn zurücknehmen.

Alle low level API Aufrufe sind Operationen auf ein paar einfache Typen von Kernelobjekten wie zum Beispiel Threads. Allen Kernobjekten wird ein kleiner Anteil vom physischem Speicher zugeteilt. Es gibt Systemaufrufe, die ein Kernelobjekt finden können, wie auch welche, die dessen Zustand bestimmen und eventuell verändern können. Threadobjekte werden beschrieben, um die Kernelobjekte zu veranscaulichen, wie auch warum diese zusammen mit den oben vorgestellten Adreßraummechanismus “state encapsulation” und “border control” erlauben. Threadobjekte untrestützen Multithreadprozesse. Ist ein Objekt einmal initialisiert, wird es aktiv und führt Befehle durch in dem ihm zugewiesenem Adreßaum aus. State visibility erlaubt aber dem Vaterprozeß, alle Threads seines Sohnes zu stoppen, deren Zustand festzulegen, wenn nötig, sogar den Sohnprozeß samt seiner Objekte zu verschieben. Das muß nicht destruktiv geschehen, also hilft die remaping Funktion aus der Adreßraumverwaltung, indem die Objekte weiterhin auf demselben physischen Speicherraum laufen, auch wenn der virtuelle ein anderer ist.

Alle Referenzen zwischen Kernelobjekten heißen *Capabilities.* Das sind Fähigkeiten, die ein Objekt besitzen kann, und dadurch auch in einer gewissen Art handeln kann. Jeder Sohnprozeß bekommt von seinem Vater eine “Capability” zugewiesen, die seine Rechte bestimmt (z.B. kann der Vater dem Sohn nur Leserechte auf einem bestimmten Speicherbereich geben...). Der Vater kann sich außerdem zwischen dem Sohn und dessen “Capability” schieben und somit laufen alle Befehle des Sohnes über dem Vater. Dadurch weiß der Vater bescheid was sein Sohn macht oder machen will und kann seine Aktivitäten kontrollieren, indem er, wenn nötig, gewisse Befehle nicht zuläßt. Das stellt nichts anderes als die “border control”- Eigenschaft dar.

3.3 Die Common Protocols API (High Level Protocols)

Der Unterschied zwischen low und high level API besteht darin, daß die zweiten vom belibigen Abstraktionsebene implementiert werden können, im Gegensatz zur low-level API, die nur der Mikrokern implementieren kann.

Die Eigenschaften aus dem zweiten Kapitel (state encapsulation und border control) sind automatisch von den common protocols API erfüllt, weil sie auf einem Capabilitymodel beruhen, die Zwichenschiebung des Vaterprozesses erlaubt. Der Vater kann verteilt dem Sohn eine “Capability”, durch der er ihm seine Rechte zuteilt, und hat damit die Kontolle im Sinne der “state encapsulation”. Die Zwischenschiebung erlaubt es ihm sogar die Befehle des Sohnes manipulieren zu können, indem alle Befehle über ihn laufen müssen. Das ist was “border control” charakterisiert.

Es gibt Schnittstellen, die bereits implementiert sind und auf denen die “common protocoll” Programme (Nester) zugreifen können. Beispiele von Nester, also Anwendungen der high level protocols werden im nächsten Abschnitt vorgestellt.

4 Anwendungen

Wie man sich gut vorstellen kann finden die Anwendungen auf der high level Protocols Ebene statt, da diese für die Implementierung verschiedener Objekte außerhalb des Mikrokerns verantwortlich sind. Die Programme heißen alle Nester und auf alle wird zugegriffen über “interposing” auf der Vaterschnittstelle. Einige Beispiele:

4.1 C - Bibliothek

Wie schon erwähnt gibt es im Mikrokern die minimale C-Bibliothek, die alle primitiven Befehle enthält. Es gibt aber auch außerhalb eine in den Common Protocols API - d. h. als Nester, mit derer Hilfe richtige Unix-Anwendungen möglich sind. Diese ist wie alle Nester optional erreichbar.

4.2 Der Prozeßmanagement Nester

Dieses Programm ermöglicht POSIX-artigen Prozessen zu laufen. Das ist der einzige Nester im System der mehrere direkte Unterprozesse auf einmal verwaltet. Alle POSIX-Prozesse sind direkte Söhne vom Prozeßmanager, aber sie dürfen untereineander die normale Prozeßhierarchie behalten.
fork() und *exec()* Befehle sind erlaubt, und der Prozeßmanager verteilt jedem POSIX-Prozeß ein ID zu. Dieser Nester ist auch optional und wird zum Beispiel verwendet, wenn *fork()* Befehle benötigt werden.Hiermit sieht man, daß man mit einem solchem System die normale Prozeßhierarchie nachahmen kann, aber umgekehrt ist es eigentlich unmöglich.

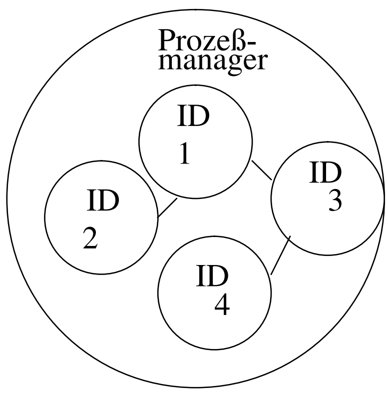

Abb.4
Verwaltung der POSIX-Prozesse

4.3 Der Checkpoint Nester

Der Checkpoiner wird wie ein “root”-Prozeß gleich direkt über dem Mikrokern gestartet, so daß ein komplett “checkpoint”-fähiges System entsteht. Auf den Nester wird über die “MemPool” Schnittstelle zugegriffen, um den Zustand des Sohnprozesses vollständig zu überwachen . Er verlangt die Rechte auf alle Adreßräume des Sohnprozesses und kann dadurch jede Aktivität (Prozesse, Threads, ...) des Sohnes erfrieren. Somit kann der aktuelle Zustand problemlos gespeichert werden. Hinterher werden die Adreßräume wieder für den Sohn freigegeben, so daß dessen Aktivitäten wieder ungestört weiter laufen können.

Der Checkpointer braucht alle Eigenschaften auf die Flux aufgebaut wurde. Mit Hilfe der “state encapsulation”, “state visibility” und der Relativität der Referenzen gelingt es ihm den Zustand des Sohnes festzuhalten, wie auch Kernobjekte zu speichern. Border control ermöglicht dem Nester auf jeder Ebene die er braucht zu agieren, um so selten wie möglich einen globalen Checkpoint zu machen. Er wird nur dort wo es notwendig ist aufgerufen. Dieses Konzept läßt sich auf beliebigen Umgebungen erweitern, auf kleinen wie einzelne Prozesse, oder aber auch auf großen wie Netzwerk “geclusterter” Maschinen.

Literaturliste

[1]H. M. Deitel: *Operating Systems;* second edition, Addison-Wesley Publishing Company.

[2] Ford, Hilber, Lepreau, Tullmann, Back, Clawson: *Microkernels Meet Recursive Virtual Machienes;* `http://www.cs.utah.edu/projects/flux/`.

[3] Ford, Van Maren, Lepreau, Clawson, Robinson, Turner: *The Flux OS Toolkit: Reusable Components for OS Implementation;* `http://www.cs.utah.edu/projects/flux.`

[4] Ford , Back, Benson, Lepreau, Lin, Shivers: *The Flux OS Kit: A Substrate for Kernel and Language Research;* `ftp://mancos.cs.utah.edu/papers/oskitsosp97.html`.