Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf

vorgelegt von
Diplom Informatiker

Wolfgang Schröder

Vom Fachbereich 20 Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs
genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr.-Ing. W. K. Giloi
Berichter:	Prof. Dr.-Ing. S. Schindler
Berichter:	Prof. Dr.-Ing. K.-P. Löhr, Freie Universität Berlin

Tag der wissenschaftlichen Aussprache: 8.12.1986

Berlin 1986
D 83

Zusammenfassung

Wolfgang Schröder: **Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf**

Mit MOOSE wird das Konzept einer Familie von Betriebssystemen vorgestellt, das es ermöglicht, applikationsspezifische Betriebssysteme entwerfen zu können, die jeweils auf gemeinsamen Entwurfsentscheidungen basieren. Ziel dieses Ansatzes ist es, einen gemeinsamen Bezugsrahmen für die verschiedenen UNIX-Dialekte (UNIX *look-alikes* und *work-alikes*) zu definieren.

Das Konzept ist so ausgelegt, daß insbesondere für den Mikroprozessor- und Mikro-Computer-Bereich jeweils optimale Systemunterstützung erreicht werden kann. Die zugrunde liegende Konzeptdiskussion motiviert ebenso die Anwendung des prozeßorientierten Systementwurfs zum Aufbau einer Betriebssystemfamilie, wie auch die auf den gegenseitigen Nachrichtenaustausch (*Message-Passing*) basierenden Mechanismen zur Interprozeßkommunikation. Zur Erarbeitung der einzelnen Motivationspunkte wird insbesondere auf Konzepte und Techniken zur Realisierung von DAS, FAMOS, MULTICS, UNIX und THOTH Bezug genommen.

In dem vorgestellten Systemmodell richtet sich die Identifikation von Prozessen nach den von ihnen zur Verfügung gestellten Diensten. Diese Dienste werden vom diensterbringenden Prozeß benannt und systemglobal bekannt gemacht. Dienstanfordernde Prozesse können auf Dienstnamen zurückgreifen, um den diensterbringenden Prozeß identifizieren und mit ihm kommunizieren zu können. Damit wird ein Prozeß logisch adressiert und kann unabhängig von einem zugrunde liegenden verteilten bzw. nichtverteilten System betrachtet werden.

Das zugrunde gelegte Identifikationsmodell ermöglicht die dynamische Rekonfiguration eines Systems. Prozesse adoptieren Dienste, wenn die Migration von Prozessen auf andere Prozessoren durchzuführen ist. Der Übergang von einem nichtverteilten zu einem verteilten (Betriebs-, Applikations-) System wird damit fließend gestaltet. Prozesse werden eingesetzt, um Traps und Interrupts zu adaptieren und dadurch deren Behandlung konsequent abzukapseln. Eine optimale Anpassung des Betriebssystems an gegebene Applikationssysteme wird dadurch zur Laufzeit ermöglicht.

Die Realisierung der erarbeiteten Konzepte wird exemplarisch anhand eines konkreten Mitglieds, AX, der MOOSE Betriebssystemfamilie vorgestellt. Hierzu werden verschiedene Vertreter des Betriebssystementwurfs herangezogen, die jeweils den "Stand der Kunst" in ihrer Sparte mit festlegen. Für den prozedurorientierten Entwurf stehen UNIX SYSTEM V und 4.2BSD Pate. Der prozeßorientierte Systementwurf wird durch QNX und V vertreten, wohingegen für den objektorientierten Systementwurf SOS das Vergleichssystem darstellt. Der prozeßorientierte und auf *Message-Passing* basierende Systementwurf zeichnet sich bei dieser vergleichenden Diskussion nicht nur durch seine klare Konzeption aus. Es zeigt sich ebenso, daß er die Konstruktion leistungsfähiger und flexibler Betriebssysteme in besonderem Maße unterstützt, womit die Eignung zum Aufbau einer Betriebssystemfamilie fundiert ist.

UNIX ist ein eingetragenes Warenzeichen von *AT&T Bell Laboratories*.

DANKSAGUNG

Von der Ausarbeitung der ersten Konzepte für MOOSE bis hin zur Realisierung des ersten voll einsatzfähigen Mitglieds der MOOSE-Betriebssystemfamilie, AX, sind ca. 3 Jahre intensiver Arbeit vergangen. Während dieser Zeit mußten viele Schwierigkeiten gemeistert werden, was ohne die hilfreiche Unterstützung von außen nicht möglich gewesen wäre.

In diesem Sinne sei besonders, Prof. Dr.-Ing. Sigram Schindler, gedankt, der mich oftmals auf den Boden der Realität zurückgeführt hatte, wenn meine Vorstellungen über das Konzept einer Betriebssystemfamilie zu weite und mystische Züge annahmen. Ihm sei ebenfalls für die Unterstützung des MOOSE-Projektes gedankt, indem Räumlichkeiten und Arbeitsplätze für die Mitarbeiter des MOOSE-Projektes zur Verfügung gestellt worden sind.

Aus dem großen Kreis der Mitarbeiter des MOOSE-Projektes sei Peter Dehne, Kai-Uwe Mehls und Andreas Stockmeier besonders gedankt. Ohne ihren vorbildlichen Einsatz für ein Projekt dieser Größenordnung, wäre die Realisierung von AX nicht möglich gewesen. Ebenso sei Gerhard Durchgraf in diesem Zusammenhang genannt, der die Portierung von AX mit großem Einsatz vorangetrieben hat und bei der Bewältigung der mc68451 allein auf weiter Flur stand. Seine Arbeit hat dazu beigetragen, Erkenntnisse zur Realisierung eines portablen Betriebssystems in den Entwurf von AX mit einfließen zu lassen.

Viele Diskussionen (wenn nicht sogar Streitgespräche) mit Bernd Oestmann über die Geheimnisse von UNIX und der UNIX-Welt, haben AX als *Advanced UNIX* in allen Punkten bestätigt. Ihm sei für seine "Streitfähigkeit" gedankt. Jörg Noltes Trap-/Interrupt-Monitor TIM war ein wesentliches Werkzeug, das half, Untiefen von QNX aufzudecken und damit zur schnellen Verwirklichung der Emulation von QNX beitrug. Auch Jörg sei für die schnelle Realisierung von TIM gedankt.

Die übrigen Mitarbeiter des MOOSE-Projektes, als da sind Ingo Donasch, Claus Dunkern, Andreas Illg, Claudia Kulessa, Birgit Panzram, Holger Pause, Thomas Patzelt, Klaus-Dieter Prasuhn, Michael Sander, Günther Tesch und James-Henry Weida, haben mit ihren Arbeiten die Grundlage für ein vollständig einsatzfähiges Betriebssystem der MOOSE-Betriebssystemfamilie gelegt. Ihnen allen sei dafür gedankt.

Meinen Kollegen Thomas Luckenbach, Ulrich Mett und Jürgen Schulze sei für ihre Unterstützung in der Überarbeitung verschiedener Versionen der vorliegenden Arbeit gedankt. Jürgen hat es immer wieder verstanden, aus einem Personal-Computer zwei zu machen und damit dem MOOSE-Projekt die dringend benötigte Hardware zur Verfügung zu stellen.

Schließlich möchte ich noch zwei Personen aus meinem direkten persönlichen Umfeld für ihre Unterstützung bei meiner Arbeit danken. Meine Freundin mußte viele eigene persönliche Einschränkungen auf sich nehmen, um mir genügend Freiraum während der vergangenen 3 Jahre offenzuhalten, MOOSE verwirklichen zu können. Mein Vater legte den Grundstein dafür, daß ich die Möglichkeit besaß, die vorliegende Arbeit habe durchführen zu können. Seine lange Krankheit und sein Tod haben die Endphase dieser Arbeit mit einem dunklen Schatten überzogen. Meinem Vater hätte sehr viel daran gelegen, den mit der vorliegenden Dissertation gegebenen weiteren Abschnitt in meinem Leben selbst miterleben zu können. Ihm war dies leider nicht vergönnt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Verbreitung von UNIX [1] hat gegenwärtig einen Stand erreicht, der dieses System zu einem Standard im Betriebssystembereich werden läßt. Mit fast jedem neu auf dem Markt angebotenen Rechnersystem wird gleichzeitig eine Portierung von UNIX vorgestellt. Software-Hersteller bemühen sich, ihre Produkte auch auf UNIX-Installationen anzubieten, um somit auch den UNIX-Anwenderkreis erreichen zu können. Im nationalen- wie auch internationalen Bereich haben sich Firmen herauskristallisiert, die sich allgemein mit der Portierung von UNIX bzw. speziell mit der Generierung von anwenderspezifischen UNIX-Installationen beschäftigen.

1.1. Der Status Quo

Der Erfolg von UNIX ist im wesentlichen durch zwei Faktoren bedingt. Die ersten, elementaren Versionen von UNIX, insbesondere Version 6 [Thompson, Ritchie 1974] und "Vanilla" Version 7 [Kernighan, McIlroy 1979], ermöglichten aufgrund ihrer einfachen und, im Vergleich zu anderen Betriebssystemen, klaren Systemstruktur, eine enorm kurze Einarbeitungsphase in das System. Hinzu kam, daß sich diese ersten Versionen besonders für Prozessoren der 16-Bit Mikroprozessortechnologie eigneten. UNIX Version 6 und Version 7 liefen anfänglich auf Maschinen der PDP/LSI-11 [2] Familie, womit der Schritt zu anderen Prozessoren vergleichbarer Architektur [3] recht geradlinig war. Es war ein Verdienst der geringen Komplexität von UNIX, auf allen 16-Bit Mikroprozessoren dieses Betriebssystem anbieten zu können. Damit war ein zusätzlicher Baustein für den Erfolg von UNIX gelegt. Fazit: für jeden Mikroprozessor der 16/32-Bit-Familie stehen prinzipiell UNIX-Versionen zur Verfügung. Engpässe ergeben sich nur in den Fällen, wo nicht genügend unterstützende Hardware um die jeweiligen Prozessoren herum angesiedelt ist.

Die andere wesentliche Tatsache für den Erfolg von UNIX lag in der firmenpolitischen Unterstützung durch die Bell Laboratories, in denen UNIX entwickelt worden ist. Diese Unterstützung wirkte sich in erster Linie dadurch aus, daß Universitäten (nationaler wie auch internationaler Herkunft) durch sehr günstige Konditionen insbesondere auf den Quellcode des UNIX-Kernels zurückgreifen konnten. Damit wurde die Ausbildung im Betriebssystembereich durch ausgiebige Fallstudien anhand von UNIX erheblich gefördert, so daß u.a. ausführliche Dokumentationen über die Struktur von UNIX entstanden. In diesem Zusammenhang sind

[1] UNIX ist ein eingetragenes Warenzeichen von *AT&T Bell Laboratories*.
[2] PDP und VMS sind ein eingetragenes Warenzeichen der *Digital Equipment Corporation*.
[3] Hierunter fallen insbesondere Systeme basierend auf dem mc68000 von Motorola, dem z8000 von Zilog, sowie dem i8086 von Intel.

[Lions 1977] und [Lions 1977b] als die Standardliteratur über UNIX für die Ausbildung im Betriebssystembereich anzusehen. Diese Arbeiten spiegeln direkt die kompakte und von einer Person allein überschaubare Systemstruktur der UNIX Version 6 wider und unterstützen u.a. damit auch gleichzeitig die Begründung für den Erfolg von UNIX im Ausbildungsbereich.

1.1.1. Der Kandidat für einen Industriestandard

Aufgrund der Verbreitung in den Universitäten galt UNIX lange Zeit als ein Betriebssystem, das industriellen Anforderungen nicht gerecht werden könnte. Dies war sicherlich nicht unbegründet, zumal auch heute noch bestimmte UNIX-Versionen für die Ausbildung im Betriebssystembereich prädestiniert sind und dort ein wesentliches Einsatzgebiet mit abdecken. Es ergab sich jedoch in "natürlicher" Weise, daß das Wissen über UNIX nach und nach in die Praxis übertragen wurde, und zwar in dem Maße, wie ein Wechsel des ausgebildeten Personals von den Universitäten in die Industrie erfolgte. Die Akzeptanz dieses Betriebssystems im kommerziellen- wie auch industriellen Bereich ließ jedoch noch lange Zeit auf sich warten. Es ist erst jetzt ein Punkt erreicht, an dem über UNIX als das Betriebssystem im Rahmen eines Industriestandards gesprochen wird.

Unabhängig von der allgemeinen Euphorie, die mit UNIX verbunden ist, darf nicht vergessen werden, daß die gegenwärtig in der Diskussion stehenden Systeme für den Betriebssystemstandard weit von den UNIX-Versionen entfernt sind, die diese Diskussion überhaupt erst ermöglicht haben. UNIX System V [Bell 1983] oder 4.2BSD [Joy et al. 1983] sind UNIX-Versionen, die von vornherein einen komplexen Hardware-Aufbau benötigen, der mit dem der sogenannten *Main Frames* verglichen werden muß. Die Komplexität ist allgemein angewachsen, insbesondere auch auf Kosten der Übersichtlichkeit und der Wartbarkeit des UNIX-Kernels [4]. Allein aufgrund dieser Tatsache werden die bislang dominierenden Betriebssysteme für den Mikro-Computer-Bereich, z.B. MS-DOS bzw. PC-DOS [5] [IBM 1984], ihren Platz weiter beibehalten. Im Zusammenhang mit den Anforderungen an die zugrunde liegende Hardware wird UNIX diese Betriebssysteme erst dann ablösen können, wenn die Kosten der entsprechenden Hardware-Komponenten auf ein Niveau abgesunken sind, wie es derzeit für die typischen Systeme im Personal-Computer-Bereich gilt. Eine Tendenz in diese Richtung ist jedoch eindeutig zu verzeichnen.

Der potentielle UNIX-Markt wird gegenwärtig in fünf Segmente unterteilt [6]. Diese Unterteilung richtet sich nach der Anzahl der Benutzer, die pro System jeweils unterstützt werden können. Hierbei stellt UNIX das dominante Betriebssystem für die "4-", "8-" und "16-Benutzer" Marktsegmente dar. Damit deckt UNIX einen Markt ab, in dem die typische Benutzeranzahl pro System fließend zwischen 2 und 20 liegt. Das "1-Benutzer" Marktsegment –und damit der "klassische" Personal-Computer-Bereich– wird von MS-DOS beherrscht. Ebenso ist UNIX für das "32-Benutzer" Marktsegment, mit einer typischen Benutzeranzahl

[4] Die Komplexität von UNIX-Versionen findet gegenwärtig ihren Höhepunkt in dem ca. 800 K-Bytes umfassenden und residenten Kernel von 4.2BSD.

[5] MS-DOS ist ein eingetragenes Warenzeichen der *Microsoft Incorporation*. IBM und PC-DOS sind eingetragene Warenzeichen der *International Business Machines*.

[6] *Mini Micro Systems/June 1984*, S. 191-193.

zwischen 24 und 100, gegenwärtig nicht dominierend. In diesem Bereich sind firmenspezifische Betriebssysteme wie VMS und AOS[7] führend.

Es ist zu erwarten, daß sich diese Markteinteilung zugunsten von UNIX verschieben wird. Hierfür sprechen gegenwärtig zwei wesentliche Aspekte. Zum einen die Marktstrategie von *AT&T*, die insbesondere auf einer standardisierten Version von UNIX System V basiert. Den Anwendern wird hiermit ein konkreter Bezugsrahmen garantiert, der es ihnen ermöglicht, auch zukünftige Anwenderprogramme mit den ihnen bekannten Dienstschnittstellen unterstützen zu können. Zum anderen die wachsende Anzahl von UNIX-orientierten Applikationsprogrammen. Insbesondere in diesem Zusammenhang ist die zentrale Aufgabe von "/usr/group" zu sehen. Diese Einrichtung ermöglicht es, einen direkten Überblick über zur Verfügung stehende Applikationspakete für UNIX zu geben. Die Verbreitung von UNIX-orientierten Applikationsprogrammen wird damit bedeutend gefördert.

Die Konzeption und die Komplexität der als "Standard" in Betracht gezogenen UNIX-Versionen, wird jedoch eher eine Verschiebung zugunsten der Akzeptanz von UNIX im "32-Benutzer" Marktsegment bedeuten. Für das typische "1-Benutzer" Marktsegment sind diese UNIX-Versionen zu stark überdimensioniert, obgleich die hardware-mäßigen Voraussetzungen für den Einsatz von UNIX durchaus gegeben sind.

Ein anderer Aspekt, der diese Tendenz untermauert, liegt in der Streuung des Einsatzgebietes von Personal-Computern. Die Anzahl der Personal-Computer und der Grad ihrer dezentralen Organisation ist bedeutend höher als bei Systemen des "32-Benutzer" Marktsegmentes. Demzufolge ist der Austausch zentraler Software-Pakete (z.B. eines Betriebssystems) im Personal-Computer-Bereich mit erheblichem organisatorischem Aufwand verbunden. Allein die Tatsache, daß z.B. MS-DOS und PC-DOS für ca. 90 % des Personal-Computer-Marktes das Betriebssystem darstellen, wird den Einsatz von UNIX in diesem Markt erheblich erschweren, wenn nicht sogar unmöglich machen. Dies ist nun sicherlich nicht an UNIX gebunden, sondern gilt allgemein für jedes System, das in einen bestehenden Markt integriert werden soll.

1.1.2. Kommerzielle, industrielle und technische Randbedingungen

Die in der Praxis eingesetzten Applikationssysteme bestimmen letztendlich die Schnittstelle und die Funktionalitäten eines neu zu integrierenden Betriebssystems. Es wird nicht zu erwarten sein, daß Applikationssysteme nachträglich an eine neue Schnittstelle des Betriebssystems angepaßt werden. Für den Anbieter oder den Anwender dieser Systeme wäre solch eine Adaption aus ökonomischen, vieleicht sogar aus technischen Gründen nicht akzeptabel.

Bei der Integration von UNIX (und anderer Software-Systeme) in einen bestehenden Markt, darf dieser Aspekt nicht vernachlässigt werden. Es ist demzufolge besonders zu berücksichtigen, daß das jeweils betrachtete Produkt (in diesem Fall ein Betriebssystem) auch in einfacher Art und Weise in einen bestehenden Software-Komplex integriert werden kann.

[7] AOS ist ein eingetragenes Warenzeichen der *Data General Corporation*.

Übliche Verfahren sind in diesem Zusammenhang sogenannte *Compatibility-Libraries* bzw. die Emulation der "alten" Systemschnittstellen.

Entsprechend dieser Problematik sind gegenwärtig Bestrebungen im Gange, auf UNIX Schnittstellen für MS-DOS anzubieten [8] und damit Kompatibilität mit Produkten des MS-DOS Marktes zu erreichen. Für den kommerziellen Sektor des Personal-Computer-Marktes wird damit der Grundstein gelegt, UNIX- und MS-DOS Applikationen "gleichzeitig" auf einem System unterstützen zu können. Die für die Verknüpfung von UNIX und MS-DOS notwendigen Software-Komponenten erhöhen jedoch allgemein die Komplexität des in Anwendung gebrachten UNIX-/MS-DOS Systems. Sollten insbesondere nur Applikationen für MS-DOS auf dem betrachteten Rechnersystem ablaufen, wäre die Kombination UNIX (als Basis) und MS-DOS (als Emulation) in jedem Fall überdimensioniert. Der Benutzer bezahlt eine solche Organisation mit verminderter Leistungsfähigkeit seines Systems. Viele Resourcen liegen brach, da sie nur im Zusammenhang mit an UNIX orientierten Applikationen verwendet werden würden [9]. Zusätzlich wäre allgemein ein größerer Aufwand an Wartungsarbeiten zu erwarten.

Die Akzeptanz von UNIX im industriellen Bereich erfolgte erst, nachdem Versionen zur Verfügung standen, die bestimmten Randbedingungen aus diesem Bereich entsprechen konnten, bzw. annähernd entsprachen. Dazu mußte es möglich sein, Realzeit- und Multiprozessorapplikationen zu unterstützen. Dies bedeutete jedoch für UNIX eine z.T. vollständig neue Verwaltungsstrategie von Prozessen und, im Vergleich zu *pipes*, effizientere Mechanismen zur Interprozeßkommunikation. UNIX dahingehend funktional zu erweitern, daß Realzeit- und Multiprozessorapplikationen unterstützt werden können, kommt einem vollständigen Redesign dieses Betriebssystems gleich. Demzufolge ist es auch nicht verwunderlich, daß es gegenwärtig für den Bereich der Realzeitsysteme noch keinen ernsthaften UNIX-Konkurrenten zu RSX [DEC 1978] gibt. In [Bach, Buroff 1984] sind u.a. die Probleme und die Grenzen eines multiprozessorfähigen UNIX Systems skizziert.

Die Akzeptanz von UNIX im technischen Bereich schließlich, d.h. auf der Ebene des Betriebssystementwurfs selbst, erfolgte recht früh. Insbesondere viele Systeme für den Micro-Computer-Bereich profitierten von den in UNIX realisierten Ideen eines portablen Betriebssystems. In den jüngeren MS-DOS Versionen ist z.B. ein hierarchisches Filesystem integriert und die Entkopplung zwischen Kernel und Ein-/Ausgabesystem erfolgte nach ähnlichen Prinzipien wie in UNIX.

Der Einfluß von UNIX auf der technischen Ebene findet z.B. gegenwärtig seinen Abschluß in OS-9 [Microware 1984]. Dieses System kann als eine vollständige Reimplementierung von UNIX Version 7 [Kernighan, McIlroy 1979] betrachtet werden, mit der Aufgabe, einerseits Kompatibilität mit UNIX zu wahren, andererseits aber, und zwar bis zur letzten Konsequenz, eine enorm laufzeiteffiziente Version darzustellen. Die Implementierung des OS-9 Kernels erfolgte in Assembler und nicht in C, wie es mit UNIX der Fall war, und der Sicherheitsaspekt, hinsichtlich der Überprüfung von Zugriffsrechten auf bestimmte Systemresourcen, wurde

[8] *Mini Micro Systems/January 1986*, S. 37-38.

[9] In diesem, für die gegenwärtige Situation recht typischem Beispiel, werden insbesondere die Mechanismen zur Prozeßverwaltung und -kommunikation von UNIX nicht vollständig bzw. garnicht in Anwendung zu bringen sein. Dies bedeutet, daß ca. 50 % des UNIX-Kernels nicht genutzt werden.

weitestgehend in den Hintergrund gestellt. Das Ergebnis ist ein Kernel, der enorm laufzeiteffizient ist und weniger Speicherplatz benötigt (der Kernel belegt ca. 24K-Bytes) als z.B. das erste akzeptable UNIX-System, Version 6.

1.1.3. Die Notwendigkeit eines UNIX-Standards

Die relativ großzügige Handhabung der Herausgabe von Lizenzen für den UNIX-Quellcode förderte nicht nur die Verbreitung des Systems. Im gleichen Maße war es nun den Lizenzinhabern möglich, eigenständig Modifikationen an dem System vorzunehmen. Oftmals wurden dadurch erkannte Fehler des Kernels beseitigt. Das intensive Arbeiten mit UNIX und der Einsatz des Systems in einer konkreten Applikationsumgebung machten jedoch schnell Unzulänglichkeiten des UNIX-Kernels deutlich. Um UNIX zur Unterstützung bestimmter Applikationssysteme einsetzen zu können, galten die Modifikationen oftmals der funktionalen Anreicherung des Kernels. Die Konsequenz davon war, daß sich nach und nach verschiedene UNIX-Dialekte herausbildeten. In [Fiedler 1983] ist eine große Anzahl solcher UNIX-Dialekte im Überblick dargestellt und in sogenannte UNIX *look-alikes* und *work-alikes* klassifiziert worden.

Um die neuen (wie auch die alten) Funktionalitäten nutzen zu können, sind zusätzliche Dienstschnittstellen (*System-Calls*) mit aufgenommen worden. Für Applikationen, die diese neuen Funktionalitäten nutzten, ergaben sich somit Abhängigkeiten von bestimmten UNIX-Dialekten, da nicht alle Versionen über dieselben Dienstschnittstellen verfügten.

Der UNIX-Kernel – genauer sein Aufbau, die zentralen Datenstrukturen, sowie verschiedene zentrale Funktionen– hat sich im Laufe der Zeit für den Systemdesigner relativ wenig verändert. Dies spricht eindeutig für UNIX und ist sicherlich auch ein wesentlicher Grund, weshalb UNIX-Portierungen in recht kurzer Zeit vollzogen werden können. Es kann auf Wissen und Informationen zurückgegriffen werden, die durchweg für alle Versionen des UNIX-Kernels ihre Gültigkeit besitzen. Hervorzuheben sind etwa die in jedem Betriebssystem besonders kritischen Mechanismen der Synchronisationen von nebenläufigen Aktivitäten: in UNIX der *sleep*- und *wakeup*-Mechanismus [Lions 1977] und [Lions 1977b]. Aber ebenso die Art und Weise, wie die Argumente der Systemdienste zwischen Benutzer- und Kerneladreßraum ausgetauscht werden, das Modell und die Mechanismen zur Erzeugung von Prozessen, die Pufferverwaltung und die Schnittstelle sowie Interaktionen zwischen dem Kernel und dem Ein-/Ausgabesystem sind in ihrer Struktur, zum Teil auch in ihrer Implementierung, seit jeher unverändert geblieben.

Änderungen an den Dienstschnittstellen umfaßten einerseits *System-Calls*, die nicht mehr angeboten worden sind, da sie durch geeignete Parameterisierung anderer, neuer *System-Calls* wieder hergeleitet werden konnten. Genauso wurden bestimmte *System-Calls* in ihrer Funktionalität erweitert. Insbesondere zählen dazu Funktionalitäten, die im Zusammenhang mit der Deskriptorverwaltung von UNIX erbracht werden. Aus *pipes* wurden *named pipes* oder *multiplexed files*, bis schließlich *sockets* die zentrale Aufgabe im Zusammenhang mit der Interprozeßkommunikation übernahmen. Filesysteme wurden durch *lock*-Mechanismen erweitert, um exklusiven Zugriff entweder über das gesamte File oder nur über Teilbereiche davon zu erreichen.

Andererseits sind Unterschiede im Zusammenhang mit bestimmten Datenstrukturen, die dem Systembenutzer zur Verfügung gestellt werden, zu sehen. Ladeformate unterscheiden sich, je nachdem, ob eine 16-Bit oder eine 32-Bit Maschine zugrunde gelegt wird. Diese Formate können auch derartige Unterschiede aufweisen, daß für denselben Prozessor Inkompatibilitäten bei Binär-Programmen auftreten. Entsprechende Problematik gilt auch im Zusammenhang mit Datenstrukturen des Filesystems, zur Ansteuerung von zeichenorientierten Geräten und der Kodierung von *System-Calls*.

Es lassen sich noch viele Beispiele mit Unterschieden zwischen einzelnen UNIX-Versionen aufführen. Offensichtlich ist jedoch, auch wenn die Unterschiede zwischen den einzelnen Benutzerschnittstellen noch so klein sind, daß der Übergang von einer UNIX-Version zur anderen inzwischen mehr Probleme mit sich bringt, als eigentlich zu erwarten war. Diese Problematik wird gegenwärtig so ernst genommen, daß sie durch eine (für einen bestimmten Anwenderkreis von UNIX bestimmte) einheitliche Definition von Systemdiensten gelöst, ihr zumindest jedoch entgegnet werden soll.

Die Definition einer einheitlichen UNIX-Schnittstelle, bezeichnet mit X/OPEN [Bull et al. 1985], geht auf die Initiative führender europäischer Firmen aus dem EDV-Bereich zurück. Das Resultat einer solchen Schnittstellendefinition kann, aufgrund der Positionen der beteiligten Firmen im kommerziellen- und industriellen Bereich, als UNIX-Standard betrachtet werden.

Um die Portabilität von Applikationssystemen auch über mehrere UNIX-Installationen hinweg zu wahren, soll mit X/OPEN jede in Betracht gezogene UNIX-Version die Kompatibilität zu einer bestimmten Menge von Systemdiensten garantieren. Diese Menge von Systemdiensten soll so ausgelegt sein, daß ein möglichst großer Anteil von Applikationen unterstützt werden kann, indem zur Interaktion mit dem Betriebssystem nur Systemdienste aus dieser Menge angewendet werden.

Die Notwendigkeit eines UNIX-Standards macht deutlich, in welchem Dilemma sich UNIX für den kommerziellen- und industriellen Bereich befindet. Die oftmals unabhängig voneinander entstandenen und verschiedenen UNIX-Versionen lassen keinen anderen Weg offen, als die zur Realisierung bestimmter Applikationen in Anwendung zu bringenden Dienstschnittstellen auf eine gemeinsame Teilmenge zu beschränken. Die Gefahr ist jedoch relativ groß, daß mit X/OPEN nur ein weiterer UNIX-Dialekt in die Galerie aller bisher existierenden UNIX-Versionen mit aufgenommen wird. X/OPEN mag sicherlich ein Schritt in Richtung eines Industriestandards für UNIX sein. Es hat sich jedoch in der Vergangenheit gezeigt, daß die Freizügigkeit bei der Entwicklungsarbeit mit UNIX, allgemein einen wesentlichen Fortschritt für UNIX selbst bedeutete. Ein UNIX-Standard könnte die Kreativität im Zuge der Weiterentwicklungen von UNIX einschränken – diese Konsequenz ist bei einem Standard jedoch unvermeidbar und gerade beabsichtigt. Die Kunst bei der Standardisierung von UNIX wird deshalb darin bestehen, genügend Spielraum (im wahrsten Sinne des Wortes) für künftige Weiterentwicklungen offenzulassen.

1.2. Das Konzept einer Familie von Betriebssystemen

Im Zuge der Integration eines Betriebssystems in bestehende Applikationsumgebungen wird immer die Entscheidung gefällt werden müssen, entweder die Schnittstellen des auszuwechselnden Betriebssystems beizubehalten, oder diese Schnittstellen zu emulieren. In jedem Fall wird der Erfolg des neuen Betriebssystems wesentlich davon abhängen, ob alle bestehenden Applikationsprogramme weiterhin unterstützt werden.

Auch wenn es nur vorübergehend ist, so werden, im Falle der Emulation von Systemschnittstellen, oftmals die neuen und alten Systemdienste von dem Betriebssystemkern weiterhin zur Verfügung gestellt. Erst wenn der Übergang zu dem neuen System vollständig vollzogen worden ist, werden die noch verbliebenen alten Systemdienste, in einer abermals neuen Systemversion, nicht mehr weiter angeboten. Dieser letzte Schritt ist jedoch nicht immer durchführbar, es sei denn, es ist die Möglicheit der dynamischen Rekonfiguration von Software-Systemen gegeben [Schindler 1978]. Nur im Zusammenhang mit Applikationen, die neu generiert werden können (weil dafür Zugriff auf die entsprechenden Objektmoduln vorhanden sind), ist, mit Hilfe von *Compatibility-Libraries*, der Übergang zu einer neuen Betriebssystemschnittstelle direkt möglich. Für Applikationen, die nur in binärer, d.h. ausführbarer Form vorliegen, muß die alte Schnittstelle in ihrer bekannten Form weiterhin vorliegen.

Aufgrund dieser besonderen Problematik sollte bereits bei der Konzeption eines Betriebssystems darauf geachtet werden, daß eine Integration dieses Systems in bestehende Applikationsumgebungen leicht möglich ist. Das alte und das neue System sollten demzufolge nicht konkurrierend gegeneinander, sondern kooperierend miteinander betrachtet werden. Ebenso gilt es, einfache Mechanismen zu realisieren, mit deren Hilfe eine evtl. spätere Rekonfiguration des Systems (das Entfernen der Komponenten zur Schnittstellenemulation) in elementarer Weise vollzogen werden kann.

Die im Zusammenhang mit UNIX skizzierten Problempunkte ließen sich vermeiden, wenn bestimmte Systemkomponenten (z.B. die Prozeßverwaltung) nicht nur konzeptionell, sondern auch, im Zuge einer technischen Realisierung, konkret voneinander entkoppelt werden würden. Die einzelnen Komponenten eines so entworfenen und realisierten Systems werden als Bausteine einer Betriebssystemfamilie [Parnas 1975] betrachtet. Jede Komponente stellt hierarchisch höher angeordneten Komponenten bestimmte Dienste zur Verfügung. Die hierarchisch höher angeordneten Systemkomponenten benutzen [Parnas 1974] die hierarchisch tiefer liegenden Komponenten, um selbst komplexere und funktional angereicherte Dienste anzubieten.

In dieser Weise könnte ein System konstruiert werden, daß gleichzeitig z.B. MS-DOS- und UNIX-Funktionalitäten erbringt, indem auf gemeinsame Systemkomponenten zugegriffen wird. Beide Systeme, UNIX und MS-DOS, teilen sich in dieser Hinsicht gemeinsame Entwurfsentscheidungen, die zum Aufbau einer Betriebssystemfamilie getroffen worden sind. Mit dieser Voraussetzung brauchen nur die Systemkomponenten zur Unterstützung eines bestimmten Applikationssystems herangezogen werden, die auch tatsächlich benötigt werden.

1.2.1. Prozeβorientierter Systementwurf

Auf Grundlage eines prozeβorientierten Systementwurfs werden einzelne oder eine Gruppe zusammenhängender Systemkomponenten jeweils durch Prozesse repräsentiert. Die funktionale Anreicherung eines bestimmten Systemkomplexes, durch die Integration weiterer Systemkomponenten, wird durch die Hinzunahme entsprechender Prozesse in elementarer Weise ermöglicht. Entsprechendes gilt, wenn Systemkomponenten aus einem bestimmten Systemkomplex herausgelöst werden sollen. In diesem Fall brauchen nur die entsprechenden Prozesse aus dem System entfernt zu werden.

Die Anwendung des Prozeβkonzeptes zur Modellierung einzelner Mitglieder einer Betriebssystemfamilie eröffnet ein breites Einsatzspektrum dieser Betriebssystemfamilie. Prozesse, und damit auch ein auf Prozeβebene dynamisch rekonfigurierbares System, können prinzipiell auf allen gängigen Prozessoren realisiert werden. Mit RC4000 [Hansen 1970] sind Ansätze gezeigt worden, in welch eleganter Weise Prozesse zur Konstruktion einer Familie von Betriebssystemen eingesetzt werden können.

Die Realisierung eines auf prozeduraler Ebene dynamisch rekonfigurierbaren Systems ist dagegen nur auf Grundlage bestimmter Hardware-Architekturen möglich. MULTICS [Organick 1972], HYDRA [Wulf et al. 1973], FAMOS [Habermann et al. 1976] und DAS [Schindler 1978] zeigen in diesem Zusammenhang, welche komplexen Hardware- und Software-Umgebungen für solch ein System vorauszusetzen sind. Das Einsatzspektrum solch eines Systems bleibt demzufolge nur auf eine kleine Menge von Prozessoren beschränkt.

1.2.2. Interprozeβkommunikation durch Message-Passing

Mit der Zuordnung von Prozessen zu bestimmten Systemkomponenten muβ die Interaktion zwischen solchen Komponenten (wenn sie von unterschiedlichen Prozessen kontrolliert werden) über die Kommunikation zwischen den jeweiligen Prozessen realisiert werden. Die dazu in Anwendung zu bringenden Mechanismen zur Interprozeβkommunikation und zur Prozeβumschaltung sind in jedem Fall laufzeitintensiver als einfache prozedurale Beziehungen zwischen den betreffenden Komponenten. Diese Tatsache wurde deshalb auch seit jeher als Argument gegen einen prozeβorientierten Systementwurf hervorgebracht, u.a. in [Habermann et al. 1976] über das T.H.E. System [Dijkstra 1968]. Hätten jedoch im Zusammenhang mit der Realisierung von prozeβorientierten Betriebssystemen vergleichbare Anstrengungen auf dem Gebiet der Hardware-Entwicklung stattgefunden, wie es bei prozedurorientierten Systemen geschehen ist –MULTICS, HYDRA, FAMOS und DAS sind bereits als Beispiele dafür genannt worden–, so würde dieses Argument sicherlich auch gegenwärtig nicht mehr in dem Maβe seine Bedeutung besitzen (siehe u.a. [Gentleman 1981]).

Dennoch, unabhängig von unterstützender Hardware, sind an die Mechanismen zur Interprozeβkommunikation in einem prozeβorientierten Betriebssystem hohe Effizienzanforderungen zu stellen. Es hat sich gezeigt, insbesondere in [Cheriton 1979], daβ *Message-Passing* als Grundlage für die Interprozeβkommunikation eine sehr effiziente und leistungsfähige Realisierung ermöglicht. Dies wird dann erreicht, wenn die Kommunikationsprimitiven blockierend auf die anwendenden Prozesse wirken. In diesem Fall werden zusätzliche Kopiervorgänge der zu übermittelnden Nachrichten vermieden.

1.3. MOOSE

Das in der vorliegenden Arbeit beschriebene Konzept einer Betriebssystemfamilie basiert auf einem prozeβorientierten Systementwurf und auf Mechanismen zur Interprozeβkommunikation nach dem *Message-Passing* Modell. Diese beiden Prinzipien stellen den Rahmen zur Verfügung, applikationsorientierte Systeme modellieren und auf Prozeβebene dynamisch rekonfigurieren zu können, ohne spezielle Anforderungen an die zugrunde liegende Hardware stellen zu müssen. Das Konzept definiert damit eine einheitliche Umgebung für eine groβe Klasse von Betriebssystemen unterschiedlichster Ausprägungen.

Zur Realisierung einzelner Mitglieder der Betriebssystemfamilie wird ein "*Message Oriented Operating System Environment*", kurz MOOSE, zugrunde gelegt. Die in dieser Arbeit vorgestellten Konzepte von MOOSE gehen auf [Schroeder 1984] zurück und stellen eine konsequente Weiterentwicklung verschiedener dort beschriebener Entwurfsentscheidungen dar.

1.3.1. Innovative Konzepte und Techniken

MOOSE lebt von einer ausgeprägten Anwendung des Prozeβkonzeptes zur Modellierung einzelner Mitglieder einer Betriebssystemfamilie. Unterstützt wird dies erheblich dadurch, daβ hardware-nahe Operationen durch eigens dafür eingerichtete Prozesse realisiert werden können. Hierbei handelt es sich um die Behandlung von Traps und Interrupts.

Um die Laufzeiten innerhalb eines prozeβorientierten Systems zu minimieren, muβ die Identifikation der Prozesse eindeutig und effizient realisiert sein. Zusätzlich muβ der Identifikationsmechanismus aber auch ein hohes Maβ an Flexibilität offen lassen, um das System auf Prozeβebene dynamisch rekonfigurieren zu können.

Im Zuge der Behandlung von Traps und der Verwaltung von Prozessen muβ es Applikationen möglich sein, auf systemspezifische Ausnahmesituationen reagieren zu können. Dies gestattet es, eine applikationsspezifische Behandlung dieser Ausnahmesituationen duchzuführen.

Die Mechanismen zur Identifikation von Prozessen, zur systemspezifischen Behandlung von Traps und Interrupts und zur applikationsspezifischen Behandlung von Ausnahmesituationen sind eigens für MOOSE konzipiert worden. Sie stellen die elementaren Verfahren dar, mittels deren Anwendung Systeme unterschiedlichster Funktionalitäten aufgebaut und somit verschiedene Mitglieder einer Betriebssystemfamilie modelliert werden können.

1.3.1.1. Die systemspezifische Behandlung von Traps

Traps stehen im engen Zusammenhang mit der Ausführung von Prozessen. Hierbei ist es irrelevant, ob der zur Ausführung von Prozessen notwendige Prozessor abstrakter oder realer Natur ist [Schindler 1983]. Der jeweilige (abstrakte/reale) Prozessor stellt während der Ausführung eines Prozesses bestimmte Ausnahmesituationen fest und leitet die entsprechende Behandlung ein.

Es ist in diesem Zusammenhang die Aufgabe eines Betriebssystems, Traps (*Hardware-Traps*), die ein realer Prozessor erkannt hat, hinsichtlich des Kontextes des den Trap auslösenden Prozesses zu behandeln. Demzufolge sind bestimmte systemspezifische

Maßnahmen zu treffen, die es erlauben, auf solche Traps reagieren zu können.

Zur systemspezifischen Behandlung von Traps wird es unter MOOSE ermöglicht, jedem einzelnen Trap einen eigenen Prozeß (*Trap-Server*) zuzuordnen. Bei jedem auftretenden Trap identifiziert der Kernel zuerst den entsprechenden *Trap-Server* und leitet dann eine Kommunikation zwischen dem den Trap auslösenden Prozeß und dem *Trap-Server* ein. Die Behandlung des Traps erfolgt nicht im Kernel, sondern wird von dem jeweiligen *Trap-Server* übernommen.

Dieser Mechanismus ist ein zentraler Bestandteil von MOOSE und wird konsequent eingesetzt, um die Behandlung von systemspezifischen Ausnahmesituationen zu ermöglichen. Im Zusammenhang mit der Modellierung von Betriebssystemdomänen (z.B. UNIX und MS-DOS) kommt diesem Mechanismus jedoch noch eine größere Bedeutung zu. Bestimmten Betriebssystemdomänen wird jeweils ein Emulator zugeordnet, der die Aufgabe hat, z.B. UNIX-Systemdienste auf Systemdienste eines MOOSE Betriebssystems abzubilden. Solche Systemdienste werden üblicherweise über Traps (*System-Calls*) in Anspruch genommen. Der Emulator nimmt die Traps an, er fungiert somit als *Trap-Server*, und die Behandlung der Traps entspricht der Emulation der jeweiligen Systemdienste.

Die damit ermöglichte Flexibilität des Systems ist offensichtlich. Emulatoren, die Bindeglieder zwischen verschiedenen Betriebssystemdomänen darstellen, brauchen erst dann verfügbar zu sein, wenn die erste Applikation der durch den Emulator definierten Betriebssystemdomäne gestartet wird. Ebenso ist die Verfügbarkeit der Emulatoren dann nicht mehr notwendig, wenn keine Applikationen der betreffenden Betriebssystemdomäne mehr vorhanden sind. Systemprozesse unter MOOSE, die für die Erzeugung und Zerstörung von Applikationen verantwortlich sind, kontrollieren in diesem Zusammenhang die Zuordnung von Applikationen zu Emulatoren.

1.3.1.2. Die systemspezifische Behandlung von Interrupts

Entsprechend der systemspezifischen Behandlung von Traps, kann auch die Behandlung von Interrupts einem Prozeß, dem sogenannten *Gate-Server*, zugeordnet werden. *Gate-Server* nehmen in diesem Zusammenhang die von den Geräten erzeugten Interrupts (*Hardware-Interrupts*) an und leiten die entsprechende Behandlung ein. Dies bedeutet, daß mit jedem auftretenden Interrupt, der entsprechend zugeordnete und behandelnde *Gate-Server* direkt aktiviert wird. Der von dem Interrupt unterbrochene Prozeß wird erst dann wieder reaktiviert, wenn der *Gate-Server* sich auf das Auftreten eines neuen Interrupts synchronisiert.

Dieser Mechanismus entkoppelt die Verarbeitung von Interrupts im speziellen und die Geräteverwaltung im allgemeinen von den restlichen Aktivitäten des Betriebssystems. Der Kernel sorgt für die Identifikation und Aktivierung des *Gate-Servers* und synchronisiert in diesem Zusammenhang sämtliche Kommunikationsaktivitäten des Systems. Ferner richtet der Kernel vor Aktivierung des entsprechenden *Gate-Servers* den für diesen Prozeß zugehörigen Adreßraum ein. Der Kernel führt prinzipiell keine gerätespezifischen Operationen durch.

Zum Aufbau von Betriebssystemfamilien sieht es das MOOSE-Konzept u.a. vor, applikationsspezifische Ein-/Ausgabesysteme in ein laufendes System integrieren zu können. *Gate-Server* werden erst dann eingerichtet, wenn mit den durch diese Prozesse kontrollierten

Geräten gearbeitet werden soll (wenn die entsprechenden Geräte "eröffnet" werden). Ebenso werden diese Prozesse zerstört, wenn die Geräte nicht mehr benötigt (wenn sie "geschlossen") werden. Durch die strenge Entkopplung des Adreβraumes und der Aktivitäten des Kernels von dem jeweiligen Adreβraum bzw. den Aktivitäten der *Gate-Server* und von den restlichen Prozessen des Systems, ist eine wesentliche Grundlage für die auf der Prozeβebene ermöglichte dynamische Rekonfiguration des Betriebssystems gegeben.

Diese Dynamik des Famlienkonzeptes unter MOOSE basiert auf einem ähnlichen Ansatz, wie er im Zusammenhang mit *capability based* Systemen verfolgt worden ist. Das DAS-System [Schindler 1978] ermöglicht in diesem Zusammenhang die dynamische Rekonfiguration eines Software-Systems auf Modulebene. In MOOSE steht der Prozeβ im Vordergrund. Die Möglichkeit zur dynamischen Rekonfiguration ist in beiden Systemen gegeben, da jeweils das Konzept der abstrakten Datentypen bei dem Entwurf eines Software-Systems in den Vordergrund gestellt und konsequent zur Anwendung gebracht wird. Die Repräsentation des jeweiligen abstrakten Datentyps (bzw. Objekte davon) kann in DAS und MOOSE zur Laufzeit ausgetauscht werden.

1.3.1.3. Die Identifikation von Prozessen

Aufbauend auf den unter MOOSE zur Verfügung gestellten Mechanismen zur Behandlung von Traps und Interrupts, kann ein prozeβorientiertes Betriebssystem bis zur letzten Konsequenz (eben die Formulierung von Prozessen zur Behandlung von Interrupts) realisiert werden. Allgemein stellt sich jedoch in einem solchen System, mehr als in prozedurorientierten Systemen, das Problem einer eindeutigen und zugleich effizienten Prozeβidentifikation.

Zur Identifikation von Prozessen wird unter MOOSE ein zweistufiger Mechanismus zur Anwendung gebracht. Auf den höheren Ebenen werden Prozesse anhand der durch sie erbrachten Dienste identifiziert. Dieser Mechanismus erlaubt die Abbildung des Namens eines abstrakten Dienstes auf die Identifikation des jeweiligen diensterbringenden Prozesses. Auf der tieferen (Kernel-) Ebene dient diese Identifikation als Schlüssel zur Adressierung der miteinander kommunizierenden Prozesse. An dieser Stelle ist ein sehr effizientes Verfahren zur Identifikation der an der Kommunikation beteiligten Prozesse realisiert. Dieses Verfahren ermöglicht es zusätzlich, mit demselben Schlüssel, jedoch zu verschiedenen Zeitpunkten, unterschiedliche Prozesse adressieren zu können. Damit ist es möglich, z.B. Kontrollprozesse in das laufende System einzuschleusen, die, transparent für die an der Kommunikation beteiligten Prozesse, die Kommunikationsaktivitäten des Systems überwachen sollen.

Solche Kontrollprozesse sind ein wesentlicher Bestandteil des MOOSE-Konzeptes, da mit Hilfe dieser Prozesse die Kommunikationsprimitiven des Kernels sehr effizient und übersichtlich gestaltet werden können. Im Zuge der Übermittlung von Nachrichten, kann z.B. die Überprüfung besonderer Zustände des empfangenden Prozesses – der Prozeβ könnte bereits terminiert sein oder wurde ausgelagert und ist damit nicht empfangsbereit – vermieden werden, wenn immer ein Prozeβ empfangsbereit zur Verfügung steht. Mit der Termination von Prozessen verliert der Schlüssel für die Identifikation dieser Prozesse nicht sofort seine Gültigkeit. Vielmehr wird er zur Adressierung eines *Garbage-Collectors* (repräsentiert über einen entsprechenden Prozeβ) für terminierte Prozesse herangezogen. Dasselbe Verfahren wird

angewendet, um Prozesse auslagern (*swappen*) zu können. Ein Systemprozeß nimmt in diesem Zusammenhang alle Kommunikationsaktivitäten für den bereits terminierten bzw. für den ausgelagerten Prozeß an und leitet eine entsprechende Ausnahmebehandlung ein.

1.3.1.4. Die applikationsspezifische Behandlung von Ausnahmesituationen

Oftmals ist es notwendig, eine applikationsspezifische Behandlung von Traps durchführen zu können. Insbesondere im Zusammenhang mit der Emulation der Dienstschnittstellen anderer Betriebssysteme muß es möglich sein, Traps entsprechend der Vorgaben einer bestimmten Betriebssystemdomäne zu behandeln. Hierunter fällt z.B. die Erzeugung eines *Core-Dumps*, der für jede Betriebssystemdomäne ein unterschiedliches Format aufweisen kann.

Ein anderer Aspekt ergibt sich im Zusammenhang mit der Realisierung fehlertoleranter Applikationssysteme. Bei solchen Systemen ist es oftmals notwendig, die Termination von Prozessen explizit angezeigt zu bekommen. Diesen Applikationen wird damit die Möglichkeit gegeben, den jeweils terminierten Prozeß erneut einzurichten. Auf diese Weise kann die Funktionalität einer Applikationen weiterhin gewährleistet werden.

Die applikationsspezifische Behandlung von Ausnahmesituationen wird unter MOOSE auf die Interprozeßkommunikation zwischen dem behandelnden Applikationsprozeß und dem anzeigenden Systemprozeß zurückgeführt. Dies stellt somit einen grundsätzlich anderen Ansatz dar, als er in herkömmlichen Systemen, z.B. UNIX, zu finden ist. In UNIX findet in diesem Zusammenhang eine Manipulation des Adreßraumes des jeweils behandelnden Prozesses statt, um bei Reaktivierung dieses Prozesses direkt die Ausnahmebehandlung starten zu können. Die Behandlung der angezeigten Ausnahmesituationen auf Grundlage der Kommunikation zwischen Prozessen fügt sich dagegen konsequent in das prozeßorientierte Modell von MOOSE ein.

1.4. Überblick

In Kapitel 2 findet eine allgemeine Konzeptdiskussion statt, die die Entscheidung zu einem prozeßorientierten Betriebssystementwurf begründen soll. Der prozedurale Ansatz beim Betriebssystementwurf wird dem prozeßorientierten gegenübergestellt. Verschiedene Möglichkeiten der Zuordnung von Adreßräumen zu Prozessen werden ebenso diskutiert, wie auch der geeignete Mechanismus zur Interprozeßkommunikation innerhalb eines prozeßorientierten Betriebssystems. Exemplarisch werden zu dieser Diskussion Konzepte von bestehenden Betriebssystemen herangezogen.

In Kapitel 3 werden die Konzepte des *Message-Passing* Kernels von MOOSE vorgestellt. Das vom Kernel vorgegebene Prozeßmodell wird erläutert, genauso wie der im Kernel angewendete Identifikationsmechanismus von Prozessen. Desweiteren finden spezielle Mechanismen im Zusammenhang mit der Interprozeßkommunikation ihre Erwähnung. Hierbei handelt es sich insbesondere um das *Broadcast*-Konzept des Kernels, wie auch um Techniken, die das transparente Überwachen von Kommunikationsaktivitäten (*Monitoring*) ermöglichen. Besondere Aufmerksamkeit wird den Mechanismen zur Behandlung von Traps und Interrupts zukommen.

In Kapitel 4 wird die Modellierung von Prozessen und die Zuordnung von Adreβräumen für die einzelnen Prozesse erläutert. Desweiteren wird auf die systemglobale Identifikation der Prozesse eingegangen, die sich anhand der von den Prozessen exportierten Dienste orientiert. Den Abschluβ bildet die Darstellung der Mechanismen von MOOSE, ein auf Prozeβebene dynamisch rekonfigurierbares System aufzubauen.

In Kapitel 5 wird das Familienkonzept von MOOSE dargestellt. Es wird gezeigt, wie auf Grundlage der Mechanismen des Kernels und des Prozeβmodells von MOOSE, Prozesse als Bausteine zur Realisierung applikationsspezifischer Betriebssysteme eingesetzt werden.

In Kapitel 6 findet die Diskussion des ersten Mitglieds der MOOSE-Betriebssystemfamilie, AX, statt. Die Struktur von AX wird vorgestellt und es wird erläutert, welche Mechanismen des Kernels und anderer zentraler Systemkomponenten von MOOSE zur Realisierung von AX beigetragen haben. Diese Diskussion bezieht den Vergleich mit anderen modernen Betriebssystemen, insbesondere UNIX, V und SOS, mit ein und führt damit zu einer Bewertung der mit AX verwirklichten Konzepte von MOOSE.

In Kapitel 7 wird kurz die weitere Perspektive von MOOSE skizziert. Es werden Aspekte eines evtl. Redesigns einzelner Systemkomponenten und zur funktionalen Anreicherung von AX diskutiert.

Kapitel 2

Konzeptdiskussion

Der Betriebssystembau kann grundlegend auf zwei Entwurfskonzepte zurückgreifen. Diese Konzepte werden durch den prozeβorientierten und prozedurorientierten Systementwurf geprägt. Welches dieser beiden Entwurfsprinzipien allgemein zu bevorzugen ist, kann ohne eine Diskussion der Vor- und Nachteile der jeweiligen Konzepte nicht direkt entschieden werden.

Neben diesen beiden Entwurfsprinzipien gewinnt ein drittes, das objektorientierte Entwurfsprinzip, immer mehr an Bedeutung. Der objektorientierte Systementwurf wird jedoch in letzter Konsequenz auf die prozedur- oder prozeβorientierte Beziehung zwischen einzelnen Systemkomponenten (die als Objekte eines bestimmten Systems betrachtet werden) basieren müssen. Der Grund hierfür ist die konkrete technische Ausprägung eines Objektes, d.h. ob die Beziehung zu dem Objekt über prozedurale oder prozeβorientierte Mechanismen realisiert wird. Der prozedur- bzw. prozeβorientierte Systementwurf wird in diesem Sinne als grundlegendes Entwurfsprinzip eines Systems betrachtet. Deshalb findet in diesem Kapitel keine Diskussion des objektorientierten Systementwurfs statt.

Die nachfolgende Diskussion unterteilt sich in drei Themenbereiche. Zunächst wird das prozeβorientierte Modell dem prozedurorientierten gegenübergestellt. Daran anschließend werden verschiedene Möglichkeiten zur adreβraummäβigen Modellierung von Prozessen erläutert. Den Abschluβ bildet eine Diskussion darüber, auf welchen Mechanismen zur Interprozeβkommunikation ein prozeβorientiertes Betriebssystem aufgebaut werden sollte.

Zu jedem der angegebenen Themenbereiche werden beispielhaft entsprechende Funktionalitäten typischer Betriebssysteme genannt. Bestimmte Problempunkte einzelner Systemkonzepte sollen damit exemplarisch aufgezeigt werden, um somit die Motivation für die resultierenden Grundkonzepte von MOOSE zu untermauern.

2.1. Prozeβ- versus prozedurorientiertes Modell

In prozeβorientierten Systemen realisiert der Kernel oftmals nur die Mechanismen zur Interprozeβkommunikation. Auf der Grundlage dieses Prinzips können typische Komponenten eines Betriebssystems (z.B. die Prozeβverwaltung, das Filesystem usw.) in speziellen Systemprozessen angesiedelt und somit von anderen Komponenten des Systems entkoppelt werden. Die von solchen Komponenten zur Verfügung gestellten Dienste (z.B. die Erzeugung eines Prozesses) werden dann über die Kommunikation mit dem entsprechenden Systemprozeβ erbracht, indem die Mechanismen des Kernels zur Interprozeβkommunikation eingesetzt werden.

In prozedurorientierten Systemen sind üblicherweise die (bei prozeβorientierten Systemen) über Systemprozesse kontrollierten Systemkomponenten direkt im Kernel integriert. Die von diesen Komponenten zur Verfügung gestellen Dienste werden über prozedurale Mechanismen, üblicherweise durch einen *System-Call* eingeleitet, in Anspruch genommen. Auch dies findet unter Kontrolle eines Prozesses statt. Anders jedoch als bei prozeβorientierten Systemen, in denen immer derselbe Systemprozeβ für die Erbringung des angeforderten Dienstes verantwortlich ist, wird diese Verantwortlichkeit in prozedurorientierten Systemen auf mehrere Prozesse verteilt. Dies ist darin begründet, daβ, aufgrund des direkten prozeduralen Bezugs zu den entsprechenden Systemkomponenten, kein Prozeβwechsel (und zwar vom dienstanfordernden zum diensterbringenden Prozeβ) stattfindet, wenn der Dienst einer bestimmten Systemkomponente in Anspruch genommen wird. Der Dienst wird unter der Verantwortlichkeit des jeweils anfordernden Prozesses erbracht.

2.1.1. Verschiedene Phasen der Erbringung eines Dienstes

Um auch in prozedurorientierten Systemen eine Unterscheidung treffen zu können, ob bestimmte Aktivitäten innerhalb oder auβerhalb des Kernels stattfinden, wird eine Differenzierung in Benutzerprozeβ und Kernelprozeβ vorgenommen. Benutzerprozesse sind üblicherweise die dienstanfordernden Prozesse, wohingegen ein Kernelprozeβ einen Benutzerprozeβ repräsentiert, unter dessen Kontrolle der angeforderte Dienst erbracht wird bzw. die Kernelaktivität stattfindet. Anders als in prozeβorientierten Systemen bedeutet dies insbesondere, daβ, je nach der Funktionalität des zu erbringenden Dienstes, die Identifikation des dienstanfordernden mit der des diensterbringenden Prozesses übereinstimmt.

Die Erbringung eines Systemdienstes läβt sich grundsätzlich in drei Phasen aufteilen. Die initiale Phase sorgt für die Übernahme der Argumente von dem dienstanfordernden Prozeβ und leitet die Erbringung des Dienstes ein. Die terminale Phase sorgt für die Rückgabe der Resultate des erbrachten Dienstes an den anfordernden Prozeβ. Die eigentliche Erbringung des Dienstes findet während der intermediären Phase statt.

In der intermediären Phase erfolgen üblicherweise Interaktionen mit anderen Systemkomponenten, um den angeforderten Systemdienst erbringen zu können[10]. Diese Phase hat insbesondere im Zusammenhang mit der Behandlung von Interrupts ihre zentrale Bedeutung. Für Dienste, die nicht die Behandlung von Interrupts bedingen, werden sämtliche Phasen von demselben Kernelprozeβ kontrolliert. Ist jedoch die Behandlung von Interrupts notwendig, kann es vorkommen, daβ die Kontrolle der intermediären Phase einem anderen Kernelprozeβ obliegt, als die der initialen- und terminalen Phase des entsprechenden Dienstes. Diese Zuordnung ist insbesondere dynamisch, d.h., die intermediäre Phase eines Dienstes ist nicht immer demselben Kernelprozeβ zuzuordnen.

Die dynamische Zuordnung der intermediären Phase eines Systemdienstes an unterschiedliche Kernelprozesse ist typisch für die herkömmlichen Betriebssysteme. In [Lions

[10] Die Erzeugung eines Prozesses würde hierbei z.B. bedeuten, Speicherplatz anzufordern und zu initialisieren, Beziehungen zwischen Prozessen und damit Prozeβfamilien aufzubauen, Objekte des erzeugenden Prozesses dem erzeugten Prozeβ zu vererben, usw.

1977] ist beschrieben, wie sich z.B. in UNIX üblicherweise mehrere Kernelprozesse bei der Erbringung der intermediären Phase eines Systemdienstes (der Ein-/Ausgabeaufträge repräsentiert) abwechseln.

2.1.2. Beziehung zwischen Benutzer- und Kernelprozeß

Der Vorteil des prozedurorientierten Systems gegenüber dem prozeßorientierten System ist offensichtlich. Zur Interaktion zwischen dem dienstanfordernden (Benutzer-) und diensterbringenden (Kernel-) Prozeß findet effektiv kein Prozeßwechsel statt, sondern dieser Prozeßwechsel ist eher konzeptioneller Natur. Er wird nicht dadurch realisiert, daß ein vollständig neuer Prozeßkontext für den Kernelprozeß vom Kernel aufgesetzt wird. Vielmehr wird dieser "Prozeßwechsel" dadurch bewerkstelligt, daß die Hardware einen Adreßraumwechsel durchführt, der durch den im Zusammenhang mit der Anforderung eines bestimmten Systemdienstes abgesetzten *System-Call* hervorgerufenen worden ist. Der betreffende Prozeß wechselt hierbei lediglich vom *Usermode* in den *Kernelmode*.

Ein anderer wesentlicher Aspekt als Vorteil des prozedurorientierten Systems gegenüber dem prozeßorientierten System liegt in der Übergabe von Argumenten während der initialen Phase und der Rückgabe von Resultaten während der terminalen Phase eines Systemdienstes. Aufgrund der direkten prozeduralen Beziehung zwischen dem dienstanfordernden und diensterbringenden Prozeß kann der Datentransfer praktisch direkt erfolgen.

Viele Prozessoren sind in dieser Hinsicht so konzipiert, daß ein *System-Call* zwar einen Wechsel vom Benutzer- in den Kerneladreßraum bedingt, daß jedoch vom Kerneladreßraum weiterhin der Benutzeradreßraum des aktiven Prozesses direkt referenziert werden kann. Bei prozeßorientierten Systemen erfolgt der Datentransfer während der initialen- und terminalen Phase eines Systemdienstes durch Anwendung geeigneter Mechanismen zur Interprozeßkommunikation.

Aufgrund dieser beiden Aspekte wird ein prozeßorientiertes System allgemein immer laufzeitintensiver als ein prozedurorientiertes System einzustufen sein, wenn die Interaktionen zwischen Benutzerprozeß und Betriebssystem (Kernelprozeß bzw. Systemprozeß) verglichen werden. Für die intermediäre Phase eines Systemdienstes muß dies nicht gelten. Moderne Mikroprozessorarchitekturen (z.B. [Tyner 1981] und [Intel 1983]) ermöglichen jedoch sehr effiziente Realisierungen der Mechanismen zur Interprozeßkommunikation und zur Prozeßumschaltung. Mit Prozessoren, die prozeßorientierte Systeme hardware-mäßig unterstützen, könnte somit der mit solchen Systemen verbundene Mehraufwand an Laufzeit während der initialen- und terminalen Phase eines Systemdienstes weitestgehend kompensiert werden.

2.1.3. Abkapselung von Systemkomponenten

Betriebssysteme nehmen eine ganz bestimmte Position innerhalb der Hierarchie sehr komplexer Software-Systeme ein (siehe insbesondere [Tsichritzis, Bernstein 1974]). Die von einem Betriebssystem zur Verfügung gestellten Dienste sind allen hierarchisch höher angeordneten Systemebenen direkt oder indirekt zugänglich. Daraus ergeben sich besondere

Anforderungen an die Korrektheit und Sicherheit eines Betriebssystems.

Seit [Dijkstra 1968] ist es unumstritten, daß nur ein klarer Systementwurf die Grundlage für ein "fehlerfreies" Betriebssystem darstellen kann. Ein Betriebssystem – das gilt sicherlich auch für andere Software-Systeme – ist dazu in unterschiedliche funktionale Einheiten zu strukturieren, die jeweils bestimmten Systemkomponenten zugeordnet werden. Die Schnittstellen zwischen solchen Systemkomponenten sollten elementar ausgelegt sein, um somit eine sehr weitreichende Unabhängigkeit zwischen den einzelnen Komponenten zu erzielen.

Die Systemkomponenten eines Betriebssystems können jeweils derselben Abstraktionsebene zugeordnet sein, oder sie definieren jeweils unterschiedliche Abstraktionsebenen [Parnas 1976]. In diesem Zusammenhang hat die Benutztrelation [Parnas 1974] zwischen den einzelnen Systemkomponenten ihre zentrale Bedeutung. Gerade die Benutztrelation eines Systems gibt Aufschluß über Abhängigkeiten zwischen den einzelnen Systemkomponenten; sei es durch die Benutzung von Diensten hierarchisch tiefer liegender Ebenen oder dadurch, daß die betreffenden Komponenten demselben Adreßraum zugeordnet sind.

2.1.3.1. Physikalische Trennung von Systemkomponenten

Bei prozedurorientierten Systemen ist die monolithische Struktur des jeweiligen Systems das besondere Wesensmerkmal. Die Systemkomponenten stehen üblicherweise in einem sehr engen Bezug zueinander. Hierbei ist weniger die prozedurale Beziehung zwischen den einzelnen Systemkomponenten für die Sicherheit des Systems von Bedeutung. Auf Grundlage des Modells der abstrakten Datentypen läßt sich jedes System, so auch ein prozedurorientiertes Betriebssystem, beispielhaft konzipieren und realisieren. Vielmehr ergibt sich dieser enge Bezug aus der Tatsache, daß die Systemkomponenten üblicherweise demselben Adreßraum zugeordnet sind.

Die übliche Organisation eines prozedurorientierten Betriebssystems bedeutet, daß alle wesentlichen Systemkomponenten, die dann den Kernel konstituieren, in einem gemeinsamen Adreßraum zusammengefaßt sind. Dies bedeutet gleichzeitig, daß von jeder Systemkomponente aus andere Systemkomponenten direkt manipuliert werden können. Hierbei ist es unerheblich, ob diese Manipulation intendiert oder ob sie durch Fehler bestimmter Systemkomponenten bedingt ist.

Nur die eindeutige Zuordnung eines eigenen Adreßraumes für jede Systemkomponente würde in diesem Zusammenhang die Integrität der jeweiligen Systemkomponenten garantieren. Um dies für prozedurorientierte Systeme in effizienter Weise zu erreichen, müssen bestimmte Prozessorarchitekturen zugrunde gelegt werden. Mit dem DAS-System sind z.B. bestimmte Konzepte realisiert worden, die solch eine adreßraummäßige Abkapselung von Systemkomponenten gestatten [Mueller et al. 1980]. HYDRA dagegen basiert auf einem Kernel, der explizit spezielle Systemdienste zur adreßraummäßigen Abkapselung von Systemkomponenten zur Verfügung stellt [Wulf et al. 1973]. Diese Systemdienste sind in erster Linie durch Software realisiert und nutzen herkömmliche Hardware-Architekturen, um die Abkapselung der Systemkomponenten physikalisch berwerkstelligen zu können.

Bei prozeßorientierten Systemen ergibt sich bereits vom Modell her die Möglichkeit, die einzelnen Systemkomponenten konkret voneinander zu entkoppeln. Mit den verschiedenen

Systemprozessen eines solchen Systems können den Systemkomponenten jeweils eigene Adreβräume zugeordnet werden. Im Gegensatz zu prozedurorientierten Systemen sind hierzu insbesondere keine speziellen Anforderungen an die zugrunde liegende Prozessorarchitektur notwendig. Die Systemkomponenten können mit dieser Adreβraumzuordnung jeweils physikalisch voneinander abgekapselt werden. Eine direkte Manipulation der Systemkomponenten untereinander ist somit weitestgehend ausgeschlossen.

2.1.3.2. Logische Trennung von Systemkomponenten

In den gängigen Betriebssystemen, insbesondere in UNIX, besitzen alle Systemkomponenten dieselben Zugriffsrechte auf systeminterne Objekte. Dies ist durch den Einsatz klassischer Rechnerarchitekturen gegeben, die oftmals nur einen zweistufigen Schutzmechanismus anbieten. Die beiden Stufen ergeben sich hierbei aus dem nicht privilegierten *Usermode* und dem privilegierten *Kernelmode*. Der Wechsel zwischen den verschiedenen Privilegebenen erfolgt durch Absetzen eines *System-Calls* oder durch Auslösen eines Interrupts.

Dieser zweistufige Schutzmechanismus für ein komplexes Software-System ist nicht erst nach den heutigen Maβstäben vollkommen unzureichend. Diese Erkenntnis hatte bereits im Zusammenhang mit MULTICS [Organick 1972] dazu geführt, andere Schutzmechanismen für das System zugrundezulegen. Auch MULTICS ist ein prozedurorientiertes Betriebssystem, womit gesagt ist, daβ die gängigen, durch Hardware unterstützten Schutzmechanismen, keinesfalls durch den prozedurorientierten Betriebssystementwurf vorgegeben sind. Ein anderes beispielhaftes System in dieser Hinsicht stellt das DAS-System, auf Grundlage der in [Mueller et al. 1980] beschriebenen Techniken, dar.

Ebenso wie zur physikalischen Entkopplung der Systemkomponenten, kann über spezielle Vorkehrungen eine logische Entkopplung der Systemkomponenten voneinander erreicht werden. Diese logische Entkopplung spiegelt sich dadurch wider, daβ bestimmten Systemkomponenten eigene Zugriffsrechte zugeordnet werden. In MULTICS wäre es ein bestimmter *protection ring*, in DAS eine entsprechende *capability*. Offensichtlich ist hierbei jedoch, daβ wiederum konkrete Forderungen an entsprechend funktionale Hardware gestellt werden müssen.

Der Kernpunkt bei den beiden skizzierten Beispielen, MULTICS und DAS, ist jedoch, daβ die Schutzmechanismen nur dann ihre Wirkung erzielen, wenn Prozesse ausgeführt werden. Mit anderen Worten, jedem Prozeβ sind in diesen Systemen bestimmte Privilegien zugeordnet, wenn er zur Erbringung eines Systemdienstes verschiedene Systemkomponenten aktiviert.

Bei prozedurorientierten Betriebssystemen können die Zuordnung und die Kontrolle von Privilegien der jeweiligen Systemkomponenten nur durch entsprechenden Rechnerarchitekturen effizient ermöglicht werden. Bei prozeβorientierten Betriebssystemen dagegen stellt sich diese Situation in einer anderen Sichtweise dar. Den Systemkomponenten sind hierbei bestimmte Systemprozesse zugeordnet, die jeweils eine eigene eindeutige Identifikation aufweisen. Zusätzlich lassen sich mit jedem Systemprozeβ verschiedene Privilegien assoziieren. Anhand der Prozeβidentifikation können somit im Zuge der Interprozeβkommunikation in elementarer Weise die Zugriffsrechte eines Prozesses auf Dienste eines anderen Prozesses kontrolliert werden.

Eine besondere Problematik des Sicherheitsaspektes eines Betriebssystems ergibt sich im Zusammenhang mit der Behandlung von Interrupts. Interrupts sind in einem Betriebssystem oftmals direkt mit der Abarbeitung eines Ein-/Ausgabeauftrages verbunden. In prozedurorientierten Systemen führt dies dazu, daß die intermediäre Phase eines Systemdienstes nicht eindeutig einem Prozeß zugeordnet werden kann [11]. Demzufolge können die Zugriffsrechte von Systemkomponenten, die im Zuge der Behandlung eines Interrupts aktiviert werden, nicht anhand des jeweils aktiven (jedoch vom Interrupt unterbrochenen) Prozesses kontrolliert werden.

Eine grundlegend andere Situation ergibt sich, wenn dem *Interrupt-Handler* ein eigener kontrollierender Prozeß zugeordnet werden kann. Dazu ist jedoch das prozeßorientierte Betriebssystemmodell gefordert, zumindest im Zusammenhang mit der Interrupt-Behandlung. Jede Systemkomponente, die zur Behandlung von Interrupts aktiviert wird, kann mit diesem Ansatz logisch von den anderen Aktivitäten des Betriebssystems abgekapselt werden.

Die demzufolge notwendige Aktivierung eines Prozesses aufgrund eines Interrupts läßt sich auf Grundlage elementarer Hardware auch durch Software in verhältnismäßig effizienter Weise bewerkstelligen. Zumindest würde diese Variante der Interrupt-Behandlung auf den heutigen Prozessoren eine echte Alternative zu der üblichen prozeduralen Form darstellen. Insbesondere kann gegenwärtig auf Hardware zurückgegriffen werden, die die logische Abkapselung vom *Interrupt-Handler* auf technischer Ebene erheblich unterstützt [12]. Um diese Möglichkeiten ausnutzen zu können, ist jedoch ein prozeßorientiertes Betriebssystem die notwendige Voraussetzung.

2.1.4. Dynamische Rekonfiguration eines Systems

Prozesse ermöglichen die "natürliche" Modularisierung komplexer Software-Systeme und unterstützen damit einen flexiblen Systementwurf. Dadurch, daß Prozesse eindeutig bestimmten Systemkomponenten zugeordnet werden können, ist der Austausch solcher Komponenten in einfacher Weise möglich [Hansen 1970]. Soll eine Systemkomponente gegen eine andere ausgewechselt werden, so kann dies jederzeit durch die Erzeugung eines neuen Prozesses und die Zerstörung des alten Prozesses geschehen. Besondere hardware-technische Voraussetzungen sind hierbei nicht notwendig. Einzig konzeptionelle und technische Anforderungen existieren, die erfüllt sein müssen, um diesen neuen Prozeß in das laufende System so integrieren zu können, daß die von ihm zur Verfügung gestellten Dienste von anderen Prozessen auch in Anspruch genommen werden können.

Die dynamische Rekonfiguration eines prozedurorientierten Systems erfordert in jedem Fall eine dazu geeignete Prozessorarchitektur. Üblicherweise wird hierzu eine *capability based architecture* [Fabry 1974] von dem zugrunde liegenden Prozessor erwartet. Prinzipiell ist der Grad an Flexibilität eines dynamisch rekonfigurierbaren prozedurorientierten Systems als sehr

[11] Hierzu sei auf [Lions 1977] verwiesen, der dokumentiert, wie die Abarbeitung von Ein-/Ausgabeaufträgen unter UNIX durch unterschiedliche Kernelprozesse erreicht wird.

[12] Der i80286 [Intel 1983] erlaubt die Aktivierung eines Prozesses aufgrund eines Interrupts innerhalb von ca. 24 Mikrosekunden.

hoch anzusehen, jedenfalls höher, als der eines auf Prozeβebene dynamisch rekonfigurierbaren Systems. Dennoch ist in diesem Zusammenhang abzuwägen, ob der notwendige Aufwand um solch ein System zu realisieren – es sei hier nur auf MULTICS [Organick 1972], das DAS-System [Schindler 1978] und FAMOS [Habermann et al. 1976] verwiesen – gerechtfertigt ist. Techniken zur dynamischen Rekonfiguration eines Systems entsprechend DAS [Isle et al. 1977], d.h. auf Modulebene, sind überdimensioniert, wenn immer gesamte Systemkomplexe von der Rekonfiguration betroffen sind. In solch einem Fall stellt die Möglichkeit der Rekonfiguration eines Systems auf Prozeβebene den adäquaten Ansatz dar.

2.1.5. Portabilität und Adaptabilität

In jedem Betriebssystem sind hardware-abhängige Komponenten enthalten. Dies ist gerade darin begründet, daβ ein Betriebssystem allgemein die Abstraktion von bestimmten prozessor- oder gerätespezifischen Eigenschaften erbringen soll (siehe auch [Tsichritzis, Bernstein 1974] bzw. [Shaw 1974]). Demzufolge sind immer Änderungen in bestimmten Komponenten des Betriebssystems vorzunehmen, wenn eine Portierung auf einen anderen Prozessor oder eine Adaption an neue Gerätekonfigurationen erfolgen soll.

In prozedurorientierten Systemen sind die betreffenden Komponenten üblicherweise im Kernel zusammengefaβt. Damit ergeben sich im Zusammenhang mit der Portierung des Systems zwei wesentliche Probleme. Der erste Problempunkt ist die Systemgenerierung, die sich aufgrund der monolithischen Struktur eines prozedurorientierten Systems über das Gesamtsystem erstreckt, auch wenn effektiv nur vereinzelte Komponenten von Änderungsmaβnahmen betroffen sind. Abhilfe würde in diesem Zusammenhang nur die Möglichkeit der dynamischen Rekonfiguration auf prozeduraler- oder Modulebene schaffen [Isle et al. 1977].

Der zweite Problempunkt ist die Testphase für die modifizierten Systemkomponenten, die sich in prozedurorientierten Systemen schwieriger gestaltet. Gerade in dieser Phase ist mit fehlerhaft operierenden Systemkomponenten zu rechnen. Solche fehlerhaften Komponenten einzugrenzen, ist aufgrund des fast direkten Bezuges zwischen den jeweiligen Systemkomponenten in einem prozedurorientierten System sehr problematisch. Der Fehler in einer bestimmten Systemkomponente kann in diesem Zusammenhang als Nebeneffekt einer anderen fehlerhaft operierenden Systemkomponente aufgetreten sein und ist z.B. nur dadurch ermöglicht worden, daβ alle Systemkomponenten, obwohl logisch voneinander getrennt, physikalisch im selben Adreβraum (dem Kerneladreβraum) angesiedelt sind.

In prozeβorientierten Systemen braucht sich die Systemgenerierung nur auf den jeweils betroffenen Systemprozeβ zu beziehen. Die Auswirkungen fehlerhaft operierender Systemkomponenten können dabei weitestgehend lokal zu dem entsprechenden Systemprozeβ eingegrenzt werden.

Daβ z.B. mit UNIX, als dem wohl gegenwärtig erfolgreichsten Vertreter eines prozedurorientierten Betriebssystems, eine sehr hohe Portabilität erreicht worden ist, bedeutet nicht, daβ prozedurorientierte Systeme allgemein als sehr portabel einzustufen sind. Beide dargestellten Problempunkte behalten auch für UNIX ihre Gültigkeit. Bei UNIX spielt vielmehr die langjährige Erfahrung mit dem System eine wesentliche Rolle, die zu der hohen Portabilität

dieses Systems geführt hat. Da sich UNIX intern, d.h. von der Systemstruktur kaum verändert hat, kann auf viel *know how* zurückgegriffen werden, das sich seit [Thompson, Ritchie 1974] angesammelt hat.

2.1.6. Verschiedene Systemtopologien

Bei der Unterstützung bestimmter Systemarchitekturen (Systemtopologien) zeigen sich allgemein Vorteile im Zusammenhang mit prozeßorientierten Systemen. Oftmals ermöglicht die Unterteilung eines Systems in eine bestimmte Anzahl autonomer Prozesse, die durch geeignete Mechanismen zur Interprozeßkommunikation untereinander in Verbindung stehen, eine optimale Modellierung komplexer Software-Systeme. Dies ist insbesondere im Zusammenhang mit Systemen der Fall, die auf einer Vernetzung von Prozessoren basieren bzw. die mit ausgeprägt nebenläufigen Aktivitäten konfrontiert werden.

2.1.6.1. Monoprozessorsysteme

Ein Prozeß für sich alleine betrachtet ist streng sequentieller Natur, d.h. er kann zu einem bestimmten Zeitpunkt auch immer nur eine bestimmte Aktivität repräsentieren. Erst mehrere Prozesse ermöglichen parallel stattfindende Aktivitäten. Dies bedeutet jedoch nicht, daß damit in prozeßorientierten Systemen gleichzeitig ein hohes Maß an parallel stattfindenden Systemaktivitäten ermöglicht wird.

Da ein Prozeß prinzipiell sequentieller Natur ist, bedingt ein Systemprozeß somit, daß die an ihn gestellten Dienstanforderungen auch sequentiell von ihm abgearbeitet werden. Dies bedeutet insbesondere, daß die Abarbeitung von anstehenden Systemdiensten verzögert wird, wenn ein Systemprozeß auf andere, externe Ereignisse warten muß. Der Durchsatz des Systemprozesses wird dadurch erheblich gesenkt. Abhilfe schafft in diesem Fall ein bestimmtes systemprozeßlokales Protokoll zur Abarbeitung von Systemdiensten und die technische Möglichkeit, nebenläufige Aktivitäten eines Systemprozesses modellieren zu können [Gentleman 1981].

Bei prozedurorientierten Systemen ist diese Sequentialität üblicherweise nicht so ausgeprägt vorhanden. Da in solchen Systemen jedem Benutzerprozeß bei der Erbringung eines bestimmten Systemdienstes immer ein Kernelprozeß zugeordnet ist, lassen sich demzufolge auch mehrere Systemaktivitäten "gleichzeitig" realisieren. Der Grad an Parallelität von nebenläufig stattfindenden Systemaktivitäten desselben Systemkomplexes (z.B. eines Filesystems) ist somit in prozedurorientierten Systemen höher als in prozeßorientierten Systemen.

Für den Betriebssystementwurf stellen die nebenläufigen Systemaktivitäten eine besondere Problematik dar. Mit jeder Systemaktivität wird die (teilweise) Erbringung eines bestimmten Systemdienstes verbunden sein. In diesem Zusammenhang werden oftmals globale Datenobjekte referenziert. Wird mehreren Prozessen der Zugriff auf gemeinsame Datenobjekte (*shared objects*) ermöglicht, so müssen geeignete Synchronisationsmechanismen angewendet werden, die einen sequenziellen Zugriff für die einzelnen Prozesse erzwingen. Synchronisationsfehler innerhalb des Betriebssystems haben üblicherweise verheerende Auswirkungen auf das Gesamtsystem. In der einschlägigen Literatur [13] sind die

[13] In [Dijkstra 1968b], [Hoare 1974] und [Campbell, Habermann 1974] sind verschiedene grundlegende

verschiedensten Ansätze beschrieben, die Synchronisation in nebenläufig arbeitenden Systemen bewerkstelligen zu können.

Da in prozedurorientierten Betriebssystemen ein höheres Maß an Parallelität bei der Erbringung von Systemdiensten möglich ist als in prozeßorientierten Systemen, tritt die Synchronisationsproblematik dort in verschärfter Form auf. Jede Systemkomponente wird zu unterschiedlichen Zeitpunkten von verschiedenen Kernelprozessen kontrolliert. Je nach Funktionalität der einzelnen Systemkomponenten, wird somit der nebenläufige Zugriff auf die jeweiligen kritischen Datenbereiche zu synchronisieren sein. Diese kritischen Bereiche zu erkennen, ist von zentraler Bedeutung beim Betriebssystementwurf. Nur durch die explizite Einkapselung der kritischen Bereiche – z.B. durch einen Monitor im Sinne von [Hoare 1974] – können Synchronisationsfehlern in einem Betriebssystem von vornherein begegnet werden.

Bei prozeßorientierten Systemen ist bereits vom Konzept her die Möglichkeit gegeben, der Synchronisationsproblematik in elementarer Weise zu begegnen. Die Systemprozesse selbst erzwingen bereits ein bestimmtes Maß an Sequentialität, so daß zumindest die durch sie repräsentierten Systemaktivitäten nicht mehr synchronisiert zu werden brauchen.

Unabhängig von dem Modell des prozeß- oder prozedurorientierten Systems ist es primär eine Frage des Systementwurfs, ob sich der Zugriff auf systeminterne Datenobjekte als besonders problematisch darstellt. Werden die Datenobjekte als synchronisierte abstrakte Datentypen [Schindler 1979b] verstanden, ist ein wesentlicher Schritt getan, die besondere Problematik des nebenläufigen Zugriffs auf solche Datenobjekte zu umgehen. Im Sinne des prozeßorientierten Systems würde ein synchronisierter abstrakter Datentyp durch einen eigenen Systemprozeß in elementarer Weise repräsentiert werden können. Dieser sequentiell ablaufende Systemprozeß ist immer der einzige auf Objekten dieses Typs operierende Prozeß. Die Mechanismen zur Interprozeßkommunikation würden in diesem Zusammenhang die entsprechenden Synchronisationsoperationen darstellen.

In prozedurorientierten Systemen dagegen würde auf die Objekte des synchronisierten abstrakten Datentyps jeweils von verschiedenen Kernelprozessen zugegriffen werden. Die Synchronisation dieses Zugriffs erfolgt über die für diesen Bereich typischen Synchronisationsoperationen.

Ein prozeßorientierter Betriebssystementwurf bedeutet jedoch nicht, daß die mit prozedurorientierten Systemen verschärft auftretende Synchronisationsproblematik grundlegend vermieden wird. Das NUKE-System [Crowley 1981] stellt ein prozeßorientiertes System dar, in dem dieselbe Problematik wie im Zusammenhang mit prozedurorientierten Systemen vorhanden ist. NUKE ist ein Redesign von UNIX Version 6 [Thompson, Ritchie 1974], in dem zwar die zentralen Systemkomponenten von UNIX in einzelne Systemprozesse aufgegliedert worden sind, die Datenstrukturen, aus Gründen der Kompatibilität, jedoch weitestgehend identisch in ihrem Aufbau blieben.

Mechanismen zur Synchronisation in klassischen Betriebssystemen beschrieben und [Schindler 1979] gibt Techniken an, um die in Kommunikationssystemen noch weitergehende Synchronisationsproblematik in den Griff zu bekommen.

In NUKE werden die *user structure* und *proc structure* [Lions 1977] eines bestimmten Prozesses jedem Systemprozeß jeweils über *Shared-Memory* Mechanismen zugänglich gemacht. Die Systemprozesse können damit direkt und effizient die Verwaltungsinformationen für jeden Benutzerprozeß entsprechend auf ihren aktuellen Stand bringen. Diese Prozesse in NUKE sind somit direkt vergleichbar mit den Kernelprozessen in UNIX und damit auch mit der besonderen Synchronisationsproblematik behaftet, wenn auf gemeinsame systeminterne Datenobjekte "gleichzeitig" zugegriffen wird.

2.1.6.2. Multiprozessorsysteme

Die Nebenläufigkeit parallel stattfindender Systemaktivitäten ist in prozedurorientierten Systemen bereits sehr weit ausgeprägt. Dies wird dann noch einmal dadurch verstärkt, wenn Multiprozessorsysteme zugrunde gelegt werden. Die in Monoprozessorsystemen ermöglichte Pseudo-Parallelität von Systemaktivitäten, üblicherweise durch Vergabe von Zeitscheiben für die einzelnen Benutzer-/Kernelprozesse geregelt, kann vermieden werden. Da mehr als ein Prozessor zur Ausführung von Prozessen zur Verfügung steht, können mehrere Systemaktivitäten auch tatsächlich parallel stattfinden; eben pro Prozessor eine solche Aktivität.

Die üblichen Nebenläufigkeitsprobleme durch geeignete Synchronisationsmechanismen in den Griff zu bekommen, wird bei Multiprozessorsystemen erheblich erschwert. Multiprozessorsysteme jedoch einzusetzen, wenn der Kernel eines prozedurorientierten Betriebssystems z.B. vollständig als Monitor [Hoare 1974] realisiert worden ist – und damit sämtliche Systemkomponenten durch solch einen Monitor kontrolliert werden –, bedeutet dasselbe Maß an Parallelität innerhalb des Betriebssystems wie bei Monoprozessorsystemen. Die Nebenläufigkeitsprobleme werden dann innerhalb des Kernels sicherlich vermieden, jedoch können in diesem Fall auch keine parallelen Systemaktivitäten stattfinden und somit für das Betriebssystem selbst auch keine Vorteile aus der Multiprozessorarchitektur gezogen werden.

Eine minimale Parallelität von Systemaktivitäten ist dann erreicht, wenn die Aktivitäten einzelner Systemkomponenten sequentialisiert und diese Komponenten unterschiedlichen Prozessoren zugeordnet werden. In prozeßorientierten Systemen ist hierzu der erste Schritt prinzipiell immer bereits vollzogen worden, da einzelne Systemkomponenten jeweils von einem sequentiell ablaufenden Systemprozeß kontrolliert werden. Das Konzept eines prozeßorientierten Betriebssystems begünstigt hierbei direkt die Anwendung von Multiprozessorsystemen. Bei dem Entwurf von prozedurorientierten Systemen muß von vornherein die Entscheidung gefällt werden, ob auch Multiprozessorsysteme unterstützt werden sollen. Dieser Verlust an Transparenz [Parnas, Siewiorek 1972] ist von zentraler Bedeutung im Zusammenhang mit dem Entwurf einer Familie von Betriebssystemen. Eine Entscheidung zugunsten des den Kernel vollständig umfassenden Monitorkonzeptes würde zum Nachteil des späteren Einsatzes von Multiprozessorsystemen führen.

In diesem Zusammenhang ist auch gerade ein wesentlicher Problempunkt mit UNIX zu sehen, weshalb Multiprozessorsysteme, zumindest mit den gängigen Versionen, nicht unterstützt werden können. Von zentraler Bedeutung an dieser Stelle ist z.B. die *text structure* in UNIX [Lions 1977], die zur Verwaltung des Codesegmentes eines Prozesses dient. Objekte dieser Struktur sind nicht als synchronisierte abstrakte Datentypen für Multiprozessorsysteme ausgelegt, und der Zugriff erfolgt von verschiedenen Systemkomponenten des Kernels aus.

Solch eine Organisation des Systemkernels gilt allgemein für alle gängigen UNIX Varianten; von Version 6 [Thompson, Ritchie 1974] über LOCUS [Popek et al. 1981], bis hin zu System V [Bell 1983] und 4.2BSD [Joy et al. 1983]. In [Bach, Buroff 1984] ist hierzu konkret dargestellt, welche strukturellen und entwurfstechnischen Konsequenzen sich für ein multiprozessorfähiges UNIX ergeben.

Die Synchronisationsproblematik im Zusammenhang mit Multiprozessorsystemen kann erheblich vermindert werden, wenn einzelne logisch zusammenhängende Systemkomponenten immer insgesamt einen Prozessor zugeteilt bekommen. Dies erfordert jedoch eine entsprechend modulare Realisierung dieser Systemkomponenten. Bei prozeßorientierten Systemen ist ein modularer Systementwurf und eine entsprechende Realisierung von vornherein gegeben, zumindest wenn die jeweils durch eigene Systemprozesse kontrollierten Systemkomponenten betrachtet werden.

Wird jedes Datenobjekt (jeder abstrakte Datentyp) von genau einem und immer demselben Systemprozeß manipuliert und ist dieser Prozeß genau auch nur einem Prozessor zugeordnet, so lassen sich die Synchronisationsprobleme in einem Multiprozessorsystem in einfacher Weise umgehen. Diese Probleme können allein durch die Anwendung der Primitiven zur Interprozeßkommunikation gelöst werden, da jene Primitiven ohnehin so ausgelegt sein müssen, daß nebenläufige Kommunikationsaktivitäten, im Zusammenhang mit einem bestimmten Systemprozeß, zu synchronisieren sind.

2.1.6.3. Netzwerksysteme

Kommunikationssysteme haben den Betriebssystementwurf enorm beeinflußt. Mit der weiträumigen Vernetzung von Rechnersystemen entstand die Notwendigkeit, geeignete Kommunikationsmechanismen für die dadurch ermöglichten Netzwerksysteme zur Verfügung zu stellen. Ebenso ist eine einheitliche Kommunikationsarchitektur [ISO 1985] entstanden, die einen Bezugsrahmen für Kommunikationssysteme definiert.

Im Zuge der Realisierung von Kommunikationssystemen sind Prozesse zur Notwendigkeit geworden. In solchen Systemen übernehmen üblicherweise Systemprozesse zentrale Aufgaben zur Unterstützung der Kommunikation in einem vernetzten System. Prozeßorientierte Betriebssysteme erlauben hierbei die optimale Modellierung solcher Kommunikationssysteme und gestatten es zusätzlich, daß einzelne Kommunikationsprozesse in elementarer Weise in das Betriebssystem integriert werden können.

Auf der Grundlage von Netzwerkarchitekturen lassen sich Applikationen dezentralisieren bzw. auf verschiedene Rechnersysteme in dem Netzwerkkomplex verteilen. Im Gegensatz zu prozedurorientierten Systemen ist mit prozeßorientierten Betriebssystemen gleiches Vorgehen möglich. Besteht zwischen den einzelnen Systemprozessen keine direkte Beziehung über *Shared-Memory* (wie bei NUKE [Crowley 1981]), so ist zumindest die dezentrale Organisation eines prozeßorientierten Betriebssystems architekturell kein wesentliches Problem mehr. Das V-System [Cheriton 1984] zeigt in diesem Zusammenhang, wie ein leistungsfähiges prozeßorientiertes Betriebssystem zur Unterstützung von *Workstation*-Architekturen aufgebaut sein kann[14].

[14] Gerade im Zusammenhang mit dem V-System sind Mechanismen zur Realisierung von *Shared-Memory*

Sicherlich ließe sich ein prozedurorientiertes Betriebssystem ebenfalls dezentralisieren. Nur würde dies in jedem Fall bedeuten, daß bestimmte Eigenschaften eines prozeßorientierten Betriebssystems übernommen werden müssen. Die dezentrale Organisation eines Systems bedeutet gerade, daß bestimmte Dienste nicht lokal erbracht werden. Vielmehr wird die Diensterbringung nur lokal initiiert und dann durch Systemkomponenten erbracht, die einem entfernten System zugeordnet sind. Die Erbringung eines solchen Dienstes wird durch geeignete Mechanismen zur Realisierung von *remote procedure calls* [Nelson 1982] ermöglicht. Dazu wird jedoch mindestens auf der entfernt liegenden Seite ein Systemprozeß existieren müssen, der *remote procedure calls* annimmt und sie bei sich lokal ausführt. Dieser Systemprozeß ist dann auf derselben Abstraktionsebene anzusiedeln, wie der einen *remote procedure call* absetzende Kernelprozeß.

Solch eine hybride Organisation eines Betriebssystems, d.h. Kernelprozeß auf der lokalen und Systemprozeß auf der entfernten Seite, würde viele dezentral organisierte Systeme unterstützen. Dennoch stellt sie allgemein einen konzeptionellen Bruch im Betriebssystementwurf dar. Der Systemprozeß zur Annahme von *remote procedure calls* hat wesentliche Auswirkungen auf die Funktionalität und Korrektheit eines dezentral organisierten Betriebssystems. Diesem Prozeß (oder gar einem Komplex von Prozessen), seine Integration in das System und auch der Mechanismus zur Interprozeßkommunikation bei der Realisierung von *remote procedure calls*, sind somit bereits bei der Konzeption eines modernen Betriebssystems besondere Aufmerksamkeit zu widmen. Zu früh gefällte Entwurfsentscheidungen –z.B. für den prozedurorientierten Systementwurf– können dazu führen, daß die Realisierung von netzwerkorientierten Systemen nicht mehr in elementarer Weise möglich ist [Parnas 1975].

Gerade für das Gesamtverständnis über Netzwerksysteme ist es hilfreich, wenn Komponenten des Betriebssystems selbst als integre Bestandteile eines netzwerkglobalen Systemkomplexes verstanden werden. Das prozeßorientierte Modell eines Betriebssystems bringt in diesem Zusammenhang die geeigneten Voraussetzungen mit sich. Mit diesem Modell ist somit, zumindest von architektureller Seite her gesehen, ein fließender Übergang von zentralen zu dezentralen (mit letzter Konsequenz, auch verteilten) Systemen möglich.

2.1.7. Entscheidung zugunsten des prozeßorientierten Modells

Es ist letztendlich eine Frage der Bewertung der einzelnen dargestellten Aspekte, ob eine Entscheidung zugunsten des prozeß- oder prozedurorientierten Entwurfs gefällt wird. Ohne eine inhaltliche Bewertung vornehmen zu wollen, würde bei einfacher Aufzählung der Vor- und Nachteile der beiden diskutierten Modelle dem prozeßorientierten Modell der Vorrang vor dem prozedurorientierten Modell gewährt werden müssen.

---

in einem dezentral organisierten Betriebssystem vorgestellt worden [Cheriton 1985]. Diese Mechanismen sind jedoch sehr laufzeitintensiv und von daher nicht für jene Zwecke anzuwenden, wie NUKE sie bräuchte, um Statusinformationen für die einzelnen Prozesse verwalten zu können. In V wird *problem-oriented Shared-Memory* im Zusammenhang mit hochvolumigen Datentransfer betrachtet, d.h. vorwiegend bei Ein-/Ausgabevorgängen, um somit die Interaktionen über ein Netzwerk auf ein Minimum reduzieren zu können.

Festzuhalten ist, daß der einzig tatsächliche Nachteil des prozeßorientierten gegenüber dem prozedurorientierten System in der geringeren Laufzeiteffizienz liegt. Dieser Nachteil wird sich, wenn überhaupt, dann nur in Monoprozessorsystemen bemerkbar machen. Schon bei Multiprozessorsystemen und erst recht bei Netzwerksystemen ist zu erwarten, daß der Mehraufwand an Prozeßumschaltungen und Aktivitäten im Zusammenhang mit der Interprozeßkommunikation nicht mehr in dem Maße im Vordergrund steht. In diesen Systemen ist es von wesentlicher Bedeutung, wenn die hochgradig nebenläufigen Aktivitäten durch entsprechende Systemprozesse kontrolliert werden.

Aber auch für Monoprozessorsysteme ist ein prozeßorientiertes Betriebssystem allgemein vorzuziehen. Einerseits, damit mit denselben Werkzeugen, zur Verfügung gestellt über entsprechende Systemprozesse, der Übergang zu Multiprozessor- und Netzwerksystemen geradlinig möglich ist. Andererseits, damit applikationsspezifische Systemdienste, je nach Anforderung, dynamisch zur Verfügung gestellt werden können, ohne zugleich auf komplexe Hardware zurückgreifen zu müssen. Dies ist gerade im Zusammenhang mit Mikroprozessor- und Mikro-Computer-Systemen von wesentlicher Bedeutung. Die Integration des Betriebssystems in bestehende Applikationsumgebungen kann somit erheblich unterstützt werden, da durch die Dynamik des prozeßorientierten Modells eine Anpassung an bestimmte Applikationen elementar möglich ist.

Das prozeßorientierte Modell eines Betriebssystems entspricht dem heutigen Stand der Technik im Betriebssystementwurf und läßt viel Spielraum für zukünftige Perspektiven. Hierzu trägt insbesondere die Flexibilität des prozeßorientierten Entwurfs bei. Auch ohne besondere Anforderungen an bestimmte Hardware-Architekturen lassen sich dynamisch rekonfigurierbare Betriebs- und Applikationssysteme aufbauen. Die Portabilität ist beim prozeßorientierten System allgemein höher als bei prozedurorientierten Systemen. Die Ursache liegt hier insbesondere in dem "natürlichen" Zwang, ein Software-System zu modularisieren, d.h. in einzelne in sich abgeschlossene Prozesse zu unterteilen. Letztendlich findet der Sicherheitsaspekt bei prozeßorientierten Systemen mehr Berücksichtigung als bei prozedurorientierten Systemen. Dies gilt insbesondere im Zusammenhang mit der Behandlung von Interrupts.

Es ist jedoch zu beachten, daß sich der prozeßorientierte Betriebssystementwurf und besondere technische Hilfsmittel wie *Shared-Memory* und *capabilities* nicht gegenseitig ausschließen. Im Gegenteil, der Einsatz solcher Hilfsmittel –sicherlich funktional durch entsprechende Systemprozesse zur Verfügung gestellt– ermöglicht in vieler Hinsicht eine sinnvolle Ergänzung des prozeßorientierten Systems. Hierbei sind insbesondere *capability based architectures* [Fabry 1974] hervorzuheben, die es erlauben würden, ein auf Prozeßebene und auf Modulebene dynamisch rekonfigurierbares System aufzubauen.

Die Basis für das Konzept einer Betriebssystemfamilie mit Hilfe spezieller Hardware-Architekturen zu begründen, würde dem Familienbegriff selbst widersprechen. Es würden von vornherein bestimmte Rechnersysteme ausgeschlossen sein, die nicht die notwendigen Hardware-Voraussetzungen mitbringen, um auch nur den Einsatz des elementarsten Mitglieds dieser Familie zu ermöglichen. Das Grundkonzept einer solchen Betriebssystemfamilie wäre zu komplex. In [Lampson 1983] wird in diesem Zusammenhang anhand vieler Beispiele der einfache Systementwurf als eine wesentliche Grundlage für ein flexibles Software-System in

den Vordergrund gestellt. MOOSE soll diesen Maximen entsprechen. Demzufolge wird ein prozeßorientiertes Modell die konzeptionelle Grundlage zum Aufbau einer Betriebssystemfamilie darstellen.

2.2. Repräsentation von Prozessen

Prozesse stellen die Bausteine der MOOSE-Betriebssystemfamilie dar. Sie werden als das zentrale Werkzeug verstanden, um applikationsorientierte Systeme modellieren und realisieren zu können. Hierbei wird die Anwendung von Prozessen zum strukturierten Systementwurf sehr konsequent verfolgt. Die extreme Form des Einsatzes von Prozessen in MOOSE ergibt sich durch die Kontrolle von *Interrupt-Handlern* durch eigens dafür eingerichtete Prozesse. Diese Prozesse werden mit jedem Interrupt gestartet und definieren damit den immer gleichen Laufzeitkontext für den jeweiligen *Interrupt-Handler*.

Für ein prozeßorientiertes Betriebssystem der MOOSE-Betriebssystemfamilie sind somit allgemein hohe Anforderungen an eine effiziente Realisierung des zugrunde liegenden Prozeßmodells zu stellen. Dies gilt im besonderen Maße für das vom MOOSE-Kernel vorgegebene Prozeßmodell.

Ausschlaggebend für die effiziente Realisierung eines Prozeßmodells ist insbesondere die Repräsentation eines Prozesses hinsichtlich seiner Adreßraumzuordnung. Hierzu zeichnen sich drei Varainten ab, die nachfolgend gegenübergestellt werden.

2.2.1. Adreßraumzuordnungen für Prozesse

Prozesse können allgemein als logisch in sich abgeschlossene Einheiten betrachtet werden, die jeweils eine bestimmte Funktionalität eines Systemkomplexes repräsentieren. Diese Funktionalitäten können im einfachsten Fall darin bestehen, nur einen Laufzeitkontext für Applikationsprogramme zu definieren. Eine andere Sichtweise von Prozessen ergibt sich, wenn mehrere Prozesse gemeinsam zur Erbringung bestimmter Funktionalitäten herangezogen werden. In diesem Fall steht die Kommunikation zwischen den Prozessen im Vordergrund.

2.2.1.1. Ein Prozeß pro Adreßraum

Die übliche Sichtweise ist es, jedem Prozeß seinen eigenen Adreßraum zur Verfügung zu stellen, auf den nur er exklusiven Zugriff hat. Diese Organisation gestattet eine vollständige adreßraummäßige Entkopplung der Prozesse. Die direkte Manipulation von Adreßräumen anderer Prozesse ist damit ausgeschlossen. Dadurch ist jedoch in jedem Fall der Einsatz spezieller Hardware bedingt, die die Integrität der einzelnen Adreßräume sicherstellt[15].

[15] Diese Forderung gilt auch dann, wenn zur Realisierung der Prozesse, bzw. der von den Prozessen ausgeführten Programme, sogenannte sichere Programmiersprachen [Loehr 1980] als Entwicklungswerkzeug herangezogen werden. Die bewußte Manipulation von Adreßräumen kann damit zwar ausgeschlossen werden, nicht jedoch die unbewußte, d.h. die z.B. durch Hardware-Fehler verursachte Adreßraummanipulation. Zur Erhöhung der Fehlertoleranz eines Systems wäre jedoch die Erkennung solcher Fehler, oder gar ihre Vermeidung, unumgänglich.

Die autonome Sichtweise für einen Prozeß läßt sich durch *Shared-Memory* abschwächen. Mit *Shared-Memory* ist die Manipulation von Adreßräumen anderer Prozesse möglich. Demzufolge ist die Integrität des Adreßraumes eines Prozesses nicht mehr gesichert.

Daß mit *Shared-Memory* der Zugriff auf andere Prozeßadreßräume ermöglicht wird, bedeutet nicht, daß ein Prozeß mehrere Adreßräume besitzt. Vielmehr wird mit Hilfe von *Shared-Memory* der Adreßraum eines Prozesses "vergrößert", wobei in diesem Zusammenhang genaustens zwischen dem logischen und realen Adreßraum eines Prozesses zu unterscheiden ist. Der logische Adreßraum eines Prozesses wird durch *Shared-Memory* zwischen mehreren realen Adreßräumen gemultiplext. Die Prozesse wenden dazu explizit bestimmte Primitiven[16] an, um Objekte anderer Prozeßadreßräume in ihren logischen Adreßraum eingeblendet zu bekommen. Die Vergrößerung eines Adreßraumes bezieht sich somit auf den realen und nicht auf den logischen Adreßraum eines Prozesses. Der logische Adreßraum bleibt immer maximal gleich groß und ist von dem zugrunde liegenden Prozessor vorgegeben.

2.2.1.2. Ein Prozeß in mehreren Adreßräumen

Die exklusive Zuordnung eines Adreßraumes zu einem Prozeß stellt oftmals für bestimmte Applikationen ein besonderes Hindernis dar. Dies ist z.B. dann der Fall, wenn eine Applikation an die Grenzen des vom jeweiligen Prozessor vorgegebenen logischen Adreßraumes stößt, d.h daß die Applikation zu groß für den zugrunde liegenden Prozessor ist. Diese Problematik besitzt gegenwärtig zumindestens noch für den Personal-Computer-Bereich ihre Gültigkeit. Ein anderes Beispiel ist darin gegeben, daß verschiedene Applikationsprogramme für gewöhnlich auf dieselben Dienstprogramme (*Libraries*) zurückgreifen. Dies ist darauf zurückzuführen, daß die Applikationsprogramme mit diesen Dienstprogrammen zusammen generiert worden und demzufolge demselben Adreßraum zugeordnet sind. Neben der damit verbundenen Einschränkung des Adreßraumes für die jeweiligen Applikationsprogramme ergibt sich, insbesondere bei einer Änderung von Dienstprogrammen, das Problem der Rekonfiguration der Applikationsprogramme.

Einen Ausweg aus dieser Situation stellen Systeme dar, die es ermöglichen, Applikationen erst zur Laufzeit mit den entsprechenden Dienstprogrammen zusammenzubinden. Dies erfordert jedoch die Zuhilfenahme unterstützender Hardware. Mit MULTICS [Organick 1972] stand erstmalig ein System zur Verfügung, das, auf Grundlage entsprechender Hardware, die notwendige Dynamik für solch eine Organisation ermöglichte. Die *trap-on-use* Eigenschaft des zugrunde liegenden Prozessors, ermöglichte es, daß die erstmalige Referenz auf ein Objekt (z.B. eine *Library*-Funktion) dazu führte, es dynamisch dem betreffenden Prozeß zuzuordnen. Diese Objekte existierten nur einmal im System und es wurde zu gegebenen Zeitpunkten den Prozessen der Zugriff auf diese Objekte verschafft.

Die Weiterentwicklung dieses Prinzips führte zu den *capability based architectures* [Fabry 1974]. Die Konsequenz eines *capability based* Systementwurfs spiegelte sich z.B. in dem DAS-System [Isle et al. 1977] wider. Prozessen wurde es hierbei ermöglicht, während ihrer Ausführung verschiedene Adreßräume zu durchlaufen. Die Adreßräume wurden jeweils durch

[16] *Phys* in UNIX und *map* in NUKE sind in diesem Zusammenhang zwei elementare Beispiele.

capabilities beschrieben, die Objekte eines abstrakten Datentyps kontrollierten. Ebenso wurde es ermöglicht, die Ausprägung eines abstrakten Datentyps zur Laufzeit zu ändern und damit die Realisierung dynamisch rekonfigurierbarer Systeme erzielt.

2.2.1.3. Mehrere Prozesse pro Adreβraum

Bei den bisher betrachteten Mechanismen ist zu beobachten, daβ allgemein die Verwaltung von Adreβräumen im Vordergrund steht. Prozessen kommt in dem Sinne nur eine zweitrangige Bedeutung zu. Es ist mehr der Aspekt der Sicherheit und, im Falle von *capabilities*, der Flexibilität eines Programmsystems hinsichtlich des Zugriffs auf bestimmte Datenobjekte, der hierbei von Bedeutung ist. Für komplexe Applikationssysteme hat es sich jedoch als sehr günstig erwiesen, eine Applikation als untereinander kooperierende und miteinander kommunizierende Prozesse aufzufassen. Dies gilt insbesondere im Zusammenhang mit Kommunikationssystemen, in denen Prozesse die wesentlichen Hilfsmittel darstellen, hochgradig nebenläufige Systemaktivitäten kontrollieren zu helfen. Die Kontrolle von Adreβräumen tritt hierbei in den Hintergrund und ist insbesondere bei dezentralen bzw. verteilten Systemen mit Hilfe von *capabilities* im Sinne von [Fabry 1974] nicht mehr möglich. Vielmehr stellen miteinander kommunizierende Prozesse die "natürliche" Basis für solche Systeme dar.

Bei vielen Applikationen, insbesondere aus dem Bereich der Realzeit- und Systemprogrammierung, ist es sinnvoll, mehreren Prozessen statisch denselben Adreβraum zuzuordnen. Die im Zusammenhang mit *Shared-Memory* notwendigen Primitiven zum dynamischen Einblenden eines anderen Adreβraumes entfallen hierbei. Vielmehr wird der Zugriff auf den gemeinsamen Adreβraum bereits mit der Prozeβumschaltung sichergestellt.

In THOTH [Cheriton 1982] ist dieses Konzept erstmalig für ein prozeβorientiertes Betriebssystem realisiert worden. Mehrere Prozesse werden zu einem *Team* zusammengefaβt, das den Adreβraum für die einzelnen Prozesse statisch festlegt. In SOS [Shapiro et al. 1985] dagegen wird ein ähnliches Prinzip auf Grundlage eines prozedur- und objektorientierten Betriebssystems verfolgt. SOS gestattet, daβ mehrere Prozesse demselben *Context* zugeordnet werden können, wobei keine statische Beziehung zwischen Prozessen und *Context* bestehen muβ. In [Liskov 1979] nehmen die *Guardians* eine vergleichbare Position ein und in AMOEBA [Mullender, Tanenbaum 1986] entspricht der *Cluster* einem THOTH-*team*.

2.2.2. Entscheidung zugunsten des Team-Konzeptes

Wie mit THOTH gezeigt worden ist, sind *Teams*, im Vergleich zu *Shared-Memory* und *capabilities*, sehr elementar und effizient realisierbar. Insbesondere erfordert die Realisierung keine besondere Anforderungen an die zugrunde liegende Hardware, wie es gerade mit *Shared-Memory* und *capabilities* der Fall ist.

Die Entscheidung für eine Repräsentation von Prozessen, die auf *Shared-Memory* oder *capabilities* basiert, würde die Portabilität des MOOSE-Kernels von vornherein einschränken. *Shared-Memory* bzw. *capabilities* werden demzufolge nicht die konzeptionelle Grundlage für die Repräsentation von Prozessen von MOOSE darstellen. Diese Techniken werden vielmehr als Hilfsmittel betrachtet, um eine effiziente Realisierung bestimmter Konzepte von MOOSE zu

ermöglichen.

Den Ausgangspunkt für eine Prozeßrepräsentation in MOOSE soll deshalb das in THOTH vorgestellte *Team*-Konzept darstellen. *Teams* gestatten die Modellierung mehrerer nebenläufiger Aktivitäten innerhalb eines durch den "klassischen Prozeßbegriff" vorgegebenen logischen Adreßraumes. Damit lassen sich insbesondere viele Problematiken im Zusammenhang mit Systemprozessen lösen, wenn ein höherer Durchsatz im Zusammenhang mit der Erbringung von Systemdiensten erfordert ist.

Durch geeignete Kontrolle der Ausführung von Prozessen innerhalb eines *Teams* lassen sich die üblichem Probleme bei nebenläufigen Zugriffen auf globale Datenbestände weitestgehend umgehen.

2.3. Message-Passing versus Shared-Memory

Die Mechanismen zur Interprozeßkommunikation sind in einem prozeßorientierten System von zentraler Bedeutung. Mit ihrer Leistungsfähigkeit, insbesondere gemessen an dem Durchsatz, den die entsprechenden Primitiven bei ihrer Anwendung ermöglichen, steht und fällt die Akzeptanz eines prozeßorientierten Betriebssystems.

Es ergibt sich somit generell die Anforderung an den Kernel eines prozeßorientierten Betriebssystems, eine sehr laufzeiteffiziente Interprozeßkommunikation zu ermöglichen. Mit dieser harten Randbedingung scheiden bestimmte Kommunikationsmechanismen, zur Unterstützung eines prozeßorientierten Betriebssystems, von vornherein aus. Darunter sind auch alle Mechanismen zur Interprozeßkommunikation zu verstehen, die von den gängigen UNIX-Versionen zur Verfügung gestellt werden. Diese Mechanismen, in [Bormann 1985] im Überblick dargestellt, werden als höhere Mechanismen zur Interprozeßkommunikation betrachtet. Aufgrund des mit diesen Mechanismen verbundenen Verwaltungsaufwandes und der daraus resultierenden geringen Laufzeiteffizienz eignen sie sich nicht dafür, als effiziente und elementare Kommunikationsmechanismen die Basis eines prozeßorientierten Betriebssystems darzustellen. Vielmehr sind diese UNIX-*like* Interprozeßkommunikationsmechanismen dazu geeignet, in einem prozeßorientierten Betriebssystem durch entsprechende Systemprozesse zur Verfügung gestellt zu werden.

Die bekannten Mechanismen zur Interprozeßkommunikation unterscheiden sich allgemein darin, ob Informationen, die die jeweiligen Prozesse miteinander austauschen, kopiert werden oder ob auf diese Informationen von den Prozessen aus gemeinsam und direkt zugegriffen werden kann. Die zuletzt genannte Variante, *Shared-Memory*, stellt hinsichtlich der Informationsübertragung eine sehr effiziente Realisierung dar. Im Zusammenhang mit dem Kopieren von Informationen kann auf eine Anzahl von Mechanismen zur Interprozeßkommunikation zurückgegriffen werden – für einen Überblick sei hier auf [Herrtwich 1985] verwiesen. Von diesen Mechanismen sind jedoch nur wenige als ernsthafte Kandidaten für eine laufzeiteffiziente Realisierung der Interprozeßkommunikation zu betrachten. Dabei handelt es sich ausschließlich um elementare Kommunikationsmechanismen auf Grundlage des Nachrichtenübermittlungskonzeptes bzw. des *Message-Passing* Konzeptes. In [Liskov 1979] und [Shatz 1984] ist hierzu jeweils eine Klassifikation bzw. Diskussion der Semantiken verschiedener Mechanismen zur Interprozeßkommunikation, auf Grundlage von

Message-Passing, gegeben.

Die an dieser Stelle betrachteten Mechanismen zur Realisierung einer laufzeiteffizienten Interprozeβkommunikation, *Message-Passing* und *Shared-Memory*, lassen sich prinzipiell in zwei Kategorien unterteilen. Diese Kategorien unterscheiden sich im wesentlichen durch den Grad an nebenläufigen Aktivitäten, die bei Anwendung der jeweiligen Kommunikationsmechanismen noch möglich sind.

Interprozeβkommunikation basiert auf zwei wesentlichen elementaren Schritten. Zum einen der Austausch von Daten, die üblicherweise von einem bestimmten Prozeβ (dem Produzenten) erzeugt worden sind und von einem anderen Prozeβ (dem Konsumenten) verarbeitet werden sollen. Zum anderen das Interaktionsprotokoll zwischen Produzenten und Konsumenten, das den Kontrollfluβ im Zuge der Interprozeβkommunikation regelt.

2.3.1. Austausch von Daten

Der Austausch von Daten findet auf Grundlage von *Message-Passing* üblicherweise durch das Kopieren der entsprechenden Nachrichten vom Produzenten zum Konsumenten statt. Im Zusammenhang mit *Shared-Memory* gestattet der Produzent dem Konsumenten einen direkten Zugriff auf die jeweiligen Datenbestände.

Shared-Memory scheint somit, hinsichtlich des Zugriffs auf die produzierten Daten, eine laufzeiteffizientere Grundlage für die Interprozeβkommunikation zu sein als *Message-Passing*. Hierbei sind jedoch mehrere Fakten zu berücksichtigen, die diesen direkten Schluβ letztendlich nicht zulassen.

Um *Shared-Memory* realisieren zu können, muβ geeignete Hardware verfügbar sein. Die Übergabe von Zugriffsrechten auf einen bestimmten Datenbestand bedeutet, dem Konsumenten eine *capability* (nach [Dennis, van Horn 1966] bzw. [Fabry 1974]) auf die entsprechenden Datenobjekte zur Verfügung zu stellen. Bei Systemen, die auf einer *Memory Management Unit* (MMU) basieren, würde hierbei üblicherweise ein bestimmtes MMU-Segment des Produzenten und des Konsumenten denselben physikalischen Adreβbereich referenzieren, in dem der jeweilige Datenbestand angelegt ist. Bei *capability based* Systemen (z.B. [Tyner 1981] und [Intel 1983]) würden Objektdeskriptoren für den Konsumenten entsprechend aufgesetzt werden. Die dazu notwendigen Verwaltungsmaβnahmen können, je nach Komplexität der zugrunde liegenden Hardware-Architektur, einen erheblichen Aufwand bedeuten. Es wird somit im wesentlichen von der Größe der betrachteten Datenobjekte abhängen, ab wann *Shared-Memory* laufzeiteffizienter als *Message-Passing* ist. Eine Entwurfsentscheidung zugunsten von *Shared-Memory*, da z.B. die gegebene Hardware-Umgebung für die im Zuge der Interprozeβkommunikation zu übergebenen Datenobjekte eine optimale Realisierung ermöglicht, kann zu erheblichen Effizienzeinbuβen führen, wenn eine andere Hardware-Architektur zugrunde gelegt wird. Für die Applikationen, die Mechanismen zur Interprozeβkommunikation anwenden, die auf *Shared-Memory* basieren, kann dies zu einem deutlichen Verlust an Transparenz von dem entsprechenden Kommunikationssystem bedeuten.

Es wird somit in erster Linie eine Frage des Umfangs der Schnittstelle eines von einem Systemprozeß zur Verfügung gestellten Dienstes sein, ob das Kopieren der entsprechenden Argumente günstiger oder ob ein direkter Zugriff auf diese Argumente zu bevorzugen ist. Verschiedene prozeßorientierte Betriebssysteme, insbesondere THOTH [Cheriton 1982], V [Cheriton 1984] und bereits RC 4000 [Hansen 1970], haben gezeigt, daß *Message-Passing* den adäquaten Mechanismus zum Austausch von Daten zwischen Benutzer- und Systemprozeß darstellt [17]. *Message-Passing* wird somit in dieser Hinsicht *Shared-Memory* allgemein vorzuziehen sein.

Da *Shared-Memory* auf entsprechend unterstützender Hardware-Architektur basiert, schränkt sich zusätzlich der Einsatzbereich eines darüber realisierten Mechanismus zur Interprozeßkommunikation erheblich ein. Die sich mit *Shared-Memory* ergebende Abhängigkeit von der Hardware-Architektur des zugrunde liegenden Prozessors wird insbesondere bis zu den Systemprozessen durchgereicht. Damit ist nicht nur der Kernel von der Portabilitätsproblematik betroffen, sondern ebenso die *Shared-Memory* anwendenden Systemprozesse. Da zudem jeder Systemprozeß auf den Mechanismus zur Interprozeßkommunikation zurückgreifen muß, um Anforderungen zur Erbringung von Systemdiensten annehmen zu können, erstreckt sich diese Portabilitätsproblematik über das gesamte Betriebssystem. *Message-Passing* dagegen erlaubt an dieser Stelle für die Systemprozesse eine Abstraktion von der zugrunde liegenden Hardware-Architektur. Nur für den Kernel ergeben sich somit in diesem Zusammenhang noch Abhängigkeiten.

Obgleich der zusätzliche Kopiervorgang im Zusammenhang mit *Message-Passing* höhere Laufzeiten des entsprechenden Kommunikationsvorgangs mit sich bringen kann, ist dieser grundlegende Mechanismus, nicht nur wegen seiner höheren Portabilität, dem *Shared-Memory* vorzuziehen. Der Grund liegt gerade in dem Kopiervorgang selbst. Das Anlegen einer Kopie, d.h., das übergebene Datenobjekt, ermöglicht es, daß Produzent und Konsument mit disjunkten Datenbeständen arbeiten können. Damit braucht nach der Übergabe des Datenobjektes von den an der Kommunikation beteiligten Prozessen auch keine Synchronisationsmaßnahme getroffen zu werden, wenn mit dem jeweiligen Datenobjekt gearbeitet wird. Synchronisationsfehler beim Zugriff auf gemeinsame Daten sind häufig die Ursache sehr schwerwiegender Fehlfunktionen innerhalb eines Betriebssystems.

Das Kopieren der Datenobjekte vom Produzenten zum Konsumenten bedeutet, daß die Besitzerrechte auf diese Objekte wechseln. Immer nur ein Prozeß wird zu einem Zeitpunkt auf dem aktuellen Datenbestand dieser Objekte arbeiten können.

[17] Im Zusammenhang mit dem V-System sind insbesondere spezielle (programmiersprachlich unterstützte) Übergabemechanismen von Argumenten auf Grundlage des *Message-Passing* Konzeptes entwickelt worden [Cheriton 1984b]. Kurze Nachrichten werden hierbei in Registern übergeben, die ihren Inhalt beim Wechsel vom Produzenten zum Konsumenten nicht verändern.

2.3.2. Interaktionsprotokolle

Das jeweilige Interaktionsprotokoll, im Zusammenhang mit den Mechanismen zur Interprozeβkommunikation, bestimmt im wesentlichen den Grad an nebenläufigen Aktivitäten, der zwischen dem Produzenten und dem Konsumenten möglich ist. Unabhängig davon, nach welchen Mechanismen der Zugriff auf die zu übergebenen Datenobjekte stattfindet, müssen Produzent und Konsument ein gemeinsames Verständnis darüber besitzen, wann ein Zugriff auf diese Objekte möglich ist. Es müssen somit Synchronisationspunkte vorgegeben werden, die eine korrekte Übergabe der Datenobjekte ermöglichen.

Die durch die Interprozeβkommunikation in Anwendung gebrachten Interaktionsprotokolle können blockierend oder nichtblockierend auf den Produzenten und den Konsumenten wirken. Ebenso sind Mischformen möglich, bei denen der Produzent oder der Konsument jeweils [illegible] Übergabe mehrerer Datenobjekte blockiert wird (siehe hierzu u.a. [Liskov 1979] und [Shatz 1984]). All die verschiedenen Ausprägungen eines Interaktionsprotokolls müssen nicht unbedingt alternativ zueinander stehen. Durch entsprechende Auslegung eines grundlegenden Interaktionsprotokolls lassen sich andere Protokollvarianten herleiten. Als Ausgangspunkt kann in diesem Zusammenhang ein für den Produzenten blockierend oder nichtblockierend wirkendes Interaktionsprotokoll zugrunde gelegt werden, von dem sich dann die anderen Formen herleiten lassen[18]. Welches dieser beiden Varianten den Vorzug erhalten soll, bedarf einer genaueren Analyse ihrer Semantiken und insbesondere der sich daraus ergebenden Konsequenz für den Entwurf eines prozeβorientierten Betriebssystems.

2.3.2.1. Asynchrone Kommunikation

Die nichtblockierende Übergabe von Datenobjekten bedingt immer zusätzlichen Speicherplatz, um die jeweiligen Objekte zwischenspeichern zu können. Die Zwischenspeicherung wird immer dann notwendig sein, wenn es nicht möglich ist, die Datenobjekte direkt dem Konsumenten zuzuführen. Ob die Zwischenspeicherung tatsächlich den Kopiervorgang in einen systeminternen Puffer mit sich bringt, ist von der zugrunde liegenden Hardware abhängig. Bei *capability based* Systemen [Fabry 1974] läβt sich dieser zusätzliche Kopiervorgang vermeiden. In solchen Systemen braucht nur die *capability* des zu übergebenden Datenobjektes vermerkt und dem Produzenten eine neue *capability* zugeordnet zu werden. Diese neue *capability* verweist dann auf ein anderes zur Verfügung stehendes Datenobjekt [19]. Dennoch muβ diese Organisation nicht unbedingt eine laufzeiteffizientere Realisierung der Interprozeβkommunikation bedeuten. Es gelten hierbei dieselben Aspekte wie bereits im Zusammenhang mit der Übergabe der Datenobjekte auf Grundlage von *Shared-Memory* beschrieben worden ist.

---

[18] Die Herleitung eines nichtblockierenden Interaktionsprotokolls ist, auf Grundlage eines blockierenden Interaktionsprotokolls, durch Zuhilfenahme des *Team*-Konzeptes möglich. Damit ist die in [Liskov 1979] gefällte Entwurfsentscheidung, einem ausbaufähigen Kommunikationsprinzip ein nichtblockierendes Interaktionsprotokoll zugrunde zu legen, nicht zwingend notwendig.

[19] Auf Grundlage eines Systems wie in [Isle et al. 1977] beschrieben, würde der Deskriptor für das betreffende Datenobjekt nur umgesetzt werden und damit anschlieβend ein entsprechendes Datenobjekt aus dem systemglobalen Pufferbereich referenzieren.

Nur wenn der zusätzliche Pufferplatz innerhalb des Interprozeßkommunikationssystems unbegrenzt groß ist, würde jederzeit die nichtblockierende Übergabe von Datenobjekten möglich sein. Dadurch jedoch, daß der bereitstehende Pufferplatz begrenzt ist, entstehen *Starvation-* und *Deadlock*-Probleme für Prozesse [Shaw 1974], die auf Kommunikationsmechanismen basieren, die eine nichtblockierende Übergabe von Datenobjekten ermöglichen. Um *Deadlocks* innerhalb des Interprozeßkommunikationssystems zu vermeiden, die im Zusammenhang mit der Alloziierung von Pufferplatz auftreten können, müßten demzufolge Anforderungen zur Übergabe von Datenobjekten zurückgewiesen werden. Da diese Problematik aufgrund des Kommunikationsverhaltens der Prozesse untereinander auftritt, kann sie auch nur durch ein geeignetes applikationsspezifisches Kommunikationsprotokoll zwischen den Prozessen umgangen werden.

2.3.2.2. Synchrone Kommunikation

Die blockierende Übergabe von Datenobjekten benötigt prinzipiell keinen zusätzlichen Pufferplatz zur zeitweiligen Aufnahme der entsprechenden Datenobjekte. Dies ergibt sich daraus, daß der übermittelnde Prozeß solange blockiert (und demzufolge auch keine weiteren Daten mehr produzieren kann), bis der annehmende Prozeß bereit ist, mit ihm zu kommunizieren, d.h. das entsprechende Datenobjekt übernimmt. Der wesentliche Vorteil bei dieser Variante ist, daß keine zusätzlichen Kopiervorgänge notwendig sind, um die jeweiligen Daten zu übermitteln.

Dennoch existieren Systeme, THOTH [Cheriton 1982] und V [Cheriton 1984], in denen Datenobjekte zwischengespeichert werden. Hierbei handelt es sich dann um Objekte eines festen Formates, vorgegeben vom Interprozeßkommunikationssystem. Um die Zwischenspeicherung von Datenobjekten zu ermöglichen, wird jedem Prozeß zum Zeitpunkt seiner Erzeugung, ein systeminterner (im Kerneladreßraum gehaltener) *Message-Descriptor* zur Aufnahme jeweils eines Objektes zugeordnet. Die Notwendigkeit für solch ein Verfahren ist oftmals in der zugrunde liegenden Hardware-Architektur (insbesondere bei MMU-organisierten Systemen) zu sehen: Die Übergabe von Datenobjekten über den Umweg des Kerneladreßraumes kann durchaus laufzeiteffizienter sein, als würde sie direkt vom Adreßraum des Produzenten zum Adreßraum des Konsumenten erfolgen. Es ist hierbei einfach eine Frage, wie lange das Programmieren von MMU-Deskriptoren und das einmalige Kopieren dauert und in welchem Verhältnis diese Zeit zu dem doppelten Kopiervorgang über den jeweiligen *Message-Descriptor* steht. In diesem Zusammenhang spielt die Tatsache eine wesentliche Rolle, daß viele Prozessoren so konzipiert sind, daß vom Kerneladreßraum immer eine direkte Zugriffsmöglichkeit auf den aktuellen Benutzeradreßraum besteht. Für diese Zugriffe ist somit kein zusätzlicher Aufwand notwendig, um den entsprechenden Benutzeradreßraum für die Übergabe des Datenobjektes erst verfügbar zu machen.

Der Nachteil bei der blockierenden Übergabe von Datenobjekten ist offensichtlich. Die Anzahl der möglichen nebenläufigen Aktivitäten innerhalb des Systems wird reduziert und die Gefahr von *Deadlocks* ist jederzeit gegeben. Ob sich dieser Nachteil tatsächlich auswirkt, ist jedoch von der jeweiligen Kommunikationsbeziehung zwischen den Prozessen abhängig.

Innerhalb eines prozeßorientierten Betriebssystems wäre die blockierende Kommunikation zwischen Benutzer- und Systemprozessen kein generelles Problem, sondern vielmehr die

erwartete Kommunikationsbeziehung zwischen diesen Prozessen. Die Benutzerprozesse fordern die Erbringung von Systemdiensten an und würden erst dann ihre Ausführung wieder aufnehmen, wenn der Dienst erbracht (oder abgebrochen) worden ist. Mit THOTH [Cheriton 1982] ist gezeigt worden, daß diese Sichtweise zu einem sehr leistungsfähigen prozeßorientierten Betriebssystem führen kann.

2.3.3. Entscheidung zugunsten des Rendezvous-Konzeptes

Es sprechen viele Gründe für die Mechanismen zur Interprozeßkommunikation auf Grundlage des *Message-Passing* Konzeptes und diese Mechanismen jeweils blockierend für den übermittelnden und den annehmenden Prozeß wirken zu lassen. Es ist einmal der Aspekt der Effizienz dieses Verfahrens, der blockierendes *Message-Passing* motiviert. Die Daten brauchen nicht zwischenzeitlich gepuffert zu werden, wenn gegenwärtig keine Kommunikation möglich ist. Dies gestattet es, die Laufzeiten innerhalb eines prozeßorientierten Betriebssystems zu minimieren und damit allgemein bessere Antwortzeiten für die Benutzerprozesse zu erzielen. Andererseits ist es der Aspekt der Portabilität, der *Message-Passing* als geeigneten Mechanismus zur Interprozeßkommunikation hervorhebt. Letztendlich gewährt blockierendes *Message-Passing* genug Flexibilität, um auch komplexere (insbesondere nichtblockierende) Kommunikationsmechanismen realisieren zu können.

Im Zusammenhang mit Mechanismen zur nichtblockierenden Interprozeßkommunikation müßten sich die Benutzerprozesse eigenständig auf die Fertigstellung des Systemdienstes synchronisieren. Der damit verbundene Mehraufwand – die einzelnen Prozesse müßten neben den jeweiligen Kommunikationsprimitiven noch Synchronisationsprimitiven explizit anwenden –, würde vollständig auf Kosten der Leistungsfähigkeit des prozeßorientierten Betriebssystems gehen. Insbesondere wären die Systemprozesse mit der Problematik konfrontiert, daß die Benutzerprozesse sich nicht eigenständig synchronisiert haben, ihre Ausführung damit weiter fortsetzen konnten, und ihnen damit die Fertigstellung des Systemdienstes nicht in direkter Weise mitgeteilt werden kann (indem der jeweilige Benutzerprozeß deblockiert werden würde).

Die Interprozeßkommunikation vom Systemprozeß zum Benutzerprozeß, um die Fertigstellung des Systemdienstes anzuzeigen, scheidet schon allein aufgrund der damit offensichtlichen *Deadlock*-Problematik aus. Vielmehr wäre es die Aufgabe des Benutzerprozesses, eine Anfrage über die Fertigstellung des Systemdienstes an den entsprechenden Systemprozeß zu stellen, ihn zu *pollen*, mit der Konsequenz, abermals eine Interprozeßkommunikation mit dem Systemprozeß einzugehen. Für solch einen Fall müßte der Systemprozeß alle erbrachten Dienste aufzeichnen und die damit verbundenen Informationen solange aufbewahren, bis der Benutzerprozeß die Erbringung des Dienstes nachfragt. Damit gilt es für den Systemprozeß sicherzustellen, daß im Zusammenhang mit dieser zusätzlichen Verwaltungsmaßnahme keine *Deadlocks* für Benutzerprozesse entstehen können.

Die Interaktion zwischen Benutzerprozeß und Systemprozeß über nichtblockierende Mechanismen zur Interprozeßkommunikation zu bewerkstelligen, bedingt somit in jedem Fall einen höheren Aufwand an Laufzeiten bei der Erbringung des jeweiligen Systemdienstes. Da unabhängig davon, ob Benutzerprozesse untereinander kommunizieren wollen,

Interprozeβkommunikation zwischen Benutzer- und Systemprozessen in einem prozeβorientierten System notwendig ist, wäre der mit der nichtblockierenden Kommunikation vorgegebene Mehraufwand nicht zu vertreten.

Blockierende Mechanismen zur Interprozeβkommunikation stellen aus diesem Grunde die adäquaten Hilfsmittel dar, Systemdienste in Anspruch nehmen zu können. Diese Mechanismen sollten insbesondere so ausgelegt sein, daβ alle Phasen eines Dienstes für den dienstanfordernden Prozeβ blockierend wirken. Nur so kann gerade die Fertigmeldung des Dienstes (die terminale Phase) in elementarer Weise dem anfordernden Prozeβ übermittelt werden. Die Mechanismen zur Interprozeβkommunikation würden somit vorsehen, daβ der Konsument, nach Aufnahme des entsprechenden Datenobjektes, den Produzenten explizit reaktivieren muβ. Zwischen beiden Prozessen findet somit ein *Rendezvous* statt, während dem sie vollständig aufeinander synchronisiert sind. In [Gentleman 1981] sind die Vorteile dieses Verfahrens zur Konstruktion eines prozeβorientierten und leistungsfähigen Betriebssystems skizziert worden.

Das *Rendezvous*-Konzept eröffnet die Möglichkeit, verschiedene Strategien zum Austauch groβer Datenbestände anzuwenden. Insbesondere würden sich hierzu *Shared-Memory* Mechanismen oder *capabilities* anbieten, die einen effizienten Austausch der Daten während des *Rendezvous* bewerkstelligen könnten.

Das geringe Maβ an nebenläufigen Aktivitäten aufgrund blockierender Kommunikationsmechanismen stellt kein wesentliches Hindernis dar. Durch Systemprozesse, lassen sich jederzeit höhere (insbesondere nichtblockierende) Kommunikationsdienste realisieren. Dies ist auch bereits mit Hilfe des *Team*-Konzeptes von MOOSE möglich und benötigt somit keine zusätzliche Unterstützung durch Systemprozesse, mit denen ein eigener Adreβraum verknüpft ist. Die in [Liskov 1979] klassifizierten Kommunikationsmechanismen – das *no-wait send*, *synchronization send* und *remote invocation send* – lassen sich auf Grundlage des *Rendezvous*-Konzeptes und des *Team*-Konzeptes von MOOSE in elementarer Weise realisieren.

Kapitel 3

Der Kernel

Anders als in prozedurorientierten Systemen, wie z.B. UNIX, stellt der Kernel eines prozeβorientierten Systems eine Systemkomponente von relativ geringer Komplexität dar. Dies ist, zumindest für MOOSE, darin begründet, daβ die jeweiligen Aufgabengebiete der Betriebssystemkernel in den genannten Systemen sehr unterschiedlich sind.

Der MOOSE-Kernel realisiert zwei wesentliche Funktionalitäten. Zum einen handelt es sich allgemein um Mechanismen, die den Austausch von Nachrichten bzw. die Übermittlung von Signalen ermöglichen und zum anderen gestattet der Kernel eine sehr flexible Handhabung der Behandlung von Traps und Interrupts durch spezielle Systemprozesse. Gerade dieser zweite Aspekt stellt die wesentlichen innovativen Konzepte des MOOSE-Kernels dar.

Der MOOSE-Kernel soll in seiner Realisierung sehr laufzeiteffizient sein. Diese Anforderung kann durch eine geeignete Realisierung der Mechanismen zur Interprozeβkommunikation und zur Behandlung von Traps und Interrupts jedoch nur teilweise erfüllt werden. Von zentraler Bedeutung sind hierbei die effiziente Realisierung des Prozeβmodells und effiziente Verfahren zur Identifikation von Prozessen. Die dafür getroffenen Entwurfsentscheidungen für den MOOSE-Kernel werden deshalb in den nachfolgenden Abschnitten ebenso erläutert, wie die Mechanismen zur Interprozeβkommunikation und zur systemspezifischen Behandlung von Traps und Interrupts.

Zur Erläuterung der Konzepte des MOOSE-Kernels werden vereinzelt bestimmte Primitiven des Kernels als Beispiele zu Hilfe genommen. Diese Primitiven werden jedoch nur in dem Maβe betrachtet, wie sie zur Unterstützung der Darstellung zusammenhängender Funktionalitäten des Kernels beitragen. Die genaue Beschreibung der Primitiven des MOOSE-Kernels ist [Schroeder et al. 1986] zu entnehmen.

3.1. Teams zur Modellierung von Adreβdomänen

Ähnlich wie in THOTH [Cheriton 1982] sieht es das Prozeβmodell von MOOSE vor, mehrere Prozesse einem Adreβraum zuzuordnen und damit ein *Team* zu bilden. Die Prozesse eines *Teams* erhalten dadurch den gemeinsamen und direkten Zugriff auf die Objekte desselben *Teams*. Objekte innerhalb eines solchen *Teams* enthalten hierbei sowohl Daten als auch die Operationen auf diesen Daten. Jeder Prozeβ besitzt jedoch weiterhin seinen eigenen lokalen Laufzeitkontext, d.h. den Prozessen ist jeweils ein eigener Stack zugeordnet.

3.1.1. Datenstrukturelle Abstraktion

Die Verwaltung eines Prozesses innerhalb des Kernels läßt sich in zwei Komplexe unterteilen. Der erste Komplex befaßt sich mit der logischen Verwaltung von Prozessen und *Teams*. Beziehungen zwischen Prozessen und *Teams* bzw. Prozessen untereinander, sowie die Zuordnung von Objekten des Kernels zu Prozessen, werden hierbei festgehalten. Der andere Komplex befaßt sich mit der physikalischen Verwaltung von Prozessen und *Teams*. Die Aufgabe dieses Komplexes ist es, die adreßraummäßige Abbildung von Prozessen und *Teams* festzuhalten und zu bewerkstelligen.

Entsprechend der grob skizzierten Aufgaben dieser einzelnen Komplexe sind zur Verwaltung von *Teams* und Prozessen jeweils zwei Datenstrukturen vorgegeben. Jedes Paar dieser Datenstrukturen ermöglicht hierbei die logische und physikalische Modellierung von Prozessen bzw. *Teams*.

3.1.1.1. Physikalische Sichtweise eines Prozesses

Das *Team*-Konzept von MOOSE sieht vor, jedem Prozeß einen eigenen Laufzeitkontext zuzuordnen, jedoch mehreren Prozessen denselben globalen Adreßraum zur Verfügung zu stellen. Dazu zeichnen sich zwei voneinander unabhängige Strukturen ab, die zur Verwaltung der einzelnen Adreßräume verwendet werden.

Der *Image-Descriptor* (ID) eines *Teams* definiert den für alle Prozesse des *Teams* gültigen globalen Adreßraum. Der *Context-Descriptor* (CD) dagegen definiert den lokalen Adreßraum eines jeden Prozesses, d.h., er repräsentiert den aktuellen Laufzeitkontext eines Prozesses.

Beide Datenstrukturen enthalten die Software-Prototypen[20)] der jeweils zugrunde liegenden Hardware zur Adreßraumverwaltung. Die Einblendung des globalen Adreßraumes eines *Teams* erfolgt dadurch, daß die entsprechende Hardware anhand der Software-Prototypen neu programmiert wird. Entsprechendes gilt im Zusammenhang mit der Einblendung des lokalen Adreßraumes eines Prozesses.

Damit das Programmieren der Hardware effizient vollzogen werden kann, entsprechen die Software-Prototypen direkt den jeweiligen Adreßraumdeskriptoren (Hardware-Prototypen) der zugrunde liegenden Hardware. Der *Image-Descriptor* wie auch der *Context-Descriptor* können beide somit optimal entsprechend der zur Verfügung stehenden Hardware-Architektur ausgelegt werden und sind damit jedoch hardware-abhängig.

Über den *Image-Descriptor* und *Context-Descriptor* werden jeweils die physikalischen Zugriffsrechte eines Prozesses beschrieben. Diese Zugriffsrechte richten sich nach der Anzahl und der Belegung der jeweiligen Software-Prototypen der Hardware zur Adreßraumverwaltung.

[20)] Als Software-Prototyp wird in diesem Zusammenhang ein Objekt bezeichnet, daß das direkte Abbild z.B. eines MMU-Deskriptors darstellt. Der reale Deskriptor ist der Hardware-Prototyp des betreffenden Objektes.

3.1.1.2. Logische Sichtweise eines Prozesses

Die logische Sichtweise eines Prozesses wird durch seine Zuordnung zu einem bestimmten *Team*, durch seine eigenen Attribute sowie durch seine Beziehungen zu anderen Prozessen geprägt. Ein *Team* definiert eine Laufzeitdomäne für eine bestimmte Anzahl von Prozessen, die demselben *Team* zugeordnet sind. Damit erhalten alle Prozesse desselben *Teams* auch dieselben Laufzeiteigenschaften, d.h., denselben Adreβraum und insbesondere dieselben Zugriffsattribute des *Teams*. Um Prozesse und *Teams* logisch und unabhängig voneinander modellieren zu können, werden für Prozesse und *Teams* jeweils eigene Datenstrukturen zur Verwaltung verwendet.

3.1.1.2.1. Der Team Control Block

Der *Team Control Block* (TCB) dient zur logischen Beschreibung eines *Teams*. Hierzu gehören, neben der Festlegung der Zugriffsattribute eines *Teams*, noch weitere Verwaltungsinformationen. So ist insbesondere die Identifikation des dem *Team* zugeordneten *Image-Descriptors* in jedem TCB enthalten. Diese Identifikation wird als Zeiger auf den entsprechenden *Image-Descriptor* vermerkt. Damit ergibt sich, wie Bild 3.1 zeigt, eine konsequente datenstrukturelle Trennung zwischen hardware-abhängigen und hardware-unabhängigen Strukturen zur Modellierung eines *Teams*.

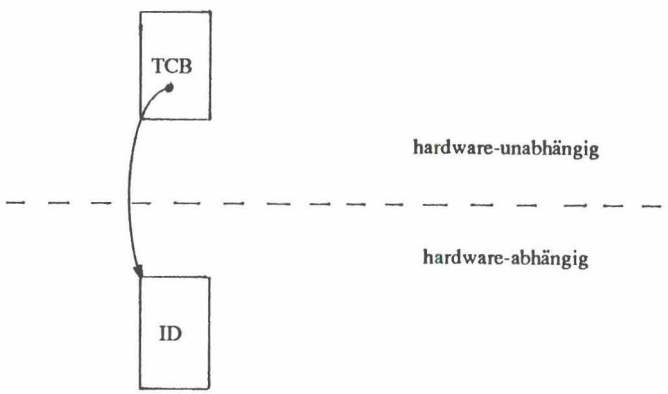

Bild 3.1: Beziehung zwischen TCB und *Image-Descriptor*

Neben der Referenz auf den jeweiligen *Image-Descriptor* wird mit jedem TCB eine lokale *Ready-List* assoziiert. In dieser Liste sind Prozesse vermerkt, die ausführbereit sind und somit dem Prozessor zugeteilt werden können. Desweiteren ist jedem *Team* eine bestimmte Zeitscheibe zugeordnet, die angibt, wie lange die Ausführung von Prozessen des *Teams* ohne Eingriff des Kernels möglich ist. Diese Zeitscheibe – deren Wert im TCB vermerkt ist – kann für jedes *Team* unterschiedlich ausgelegt sein. Und letztendlich werden die TCBs, die *Teams* mit ausführbereiten Prozessen zugeordnet sind, untereinander verkettet. Diese Liste wird vom Kernel abgearbeitet, um ein (evtl.) neues *Team* zur Ausführung zu bringen.

Die vom Kernel verwalteten TCBs werden in einer *Team-Table* zusammengefaßt. Der Umfang dieser Tabelle ist statisch festgelegt. Nur speziellen Systemprozessen, die für die Rekonfiguration des Systems verantwortlich sind, ist es möglich, die *Team-Table* selbst zu rekonfigurieren. Hiervon würden *Teams* betroffen sein, die jeweils bestimmten Applikationen fest zugeordnet sind. In dieser Hinsicht ergibt sich faktisch "keine" Begrenzung an zur Verfügung stehenden TCBs.

3.1.1.2.2. Der Process Control Block

Der *Process Control Block* (PCB) dient zur logischen Beschreibung eines Prozesses. Hierzu zählen insbesondere prozeßlokale Attribute, sowie die Identifikation des Laufzeitkontextes eines Prozesses.

Der Laufzeitkontext eines Prozesses wird im wesentlichen durch zwei Parameter bestimmt. Einerseits wird jedem Prozeß statisch ein bestimmter globaler Adreßraum zugeordnet. Dies erfolgt in elementarer Weise durch die Identifikation des *Teams*, in dem der betreffende Prozeß angesiedelt ist. Andererseits ist mit jedem Prozeß ein eigener lokaler Adreßraum verbunden. Dieser Adreßraum ist der jeweilige Laufzeitstack eines Prozesses, definiert durch den entsprechenden *Context-Descriptor*.

Die Identifikation des *Teams* und des jeweiligen *Context-Descriptors* eines Prozesses wird durch Zeiger auf die entsprechenden Datenstrukturen repräsentiert. Bild 3.2 zeigt, wie aufgrund dieser Organisation ein *Team* bestehend aus drei Prozessen vollständig modelliert wird.

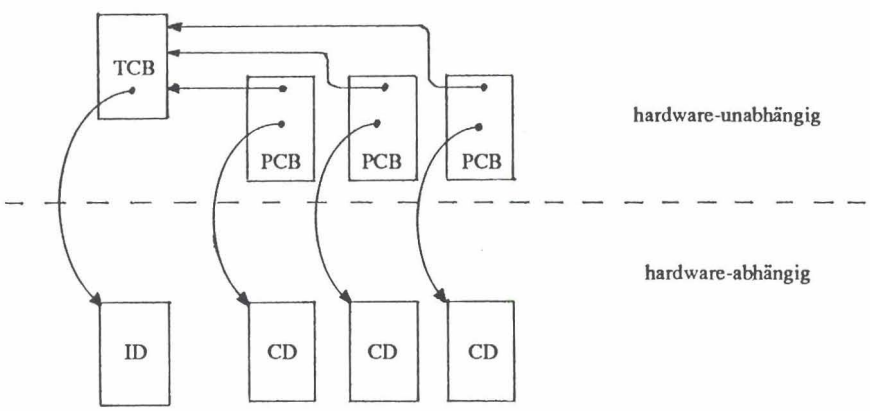

Bild 3.2: Datenstrukturelle Modellierung eines *Teams*

Die im PCB verzeichneten Attribute bestimmen das Laufzeitverhalten eines Prozesses. Desweiteren sind im PCB Informationen enthalten, die Aufschluß darüber geben, mit welchem Prozeß eine Kommunikation stattfindet bzw. welche Prozesse eine Kommunikation mit dem, durch den PCB beschriebenen, Prozeß eingehen wollen.

Die PCBs selbst können, je nach Zustand der entsprechenden Prozesse, in der *team*-lokalen *Ready-List* enthalten sein. In diesem Fall sind die den PCBs zugehörigen Prozesse ausführbereit. Ferner ist in den PCBs der Anfang und das Ende einer *Message-List* enthalten. In dieser Liste sind alle Nachrichten verzeichnet, die zum Empfang durch den jeweiligen Prozeβ bereitstehen.

Die vom Kernel verwalteten PCBs werden über eine *Process-Table* zusammengefaβt. Der Umfang dieser Tabelle ist, wie im Zusammenhang mit der *Team-Table* bereits erläutert, statisch festgelegt. Doch ist es auch in diesem Fall bestimmten Systemprozessen möglich, die *Process-Table* zu rekonfigurieren. Hiervon würden all die PCBs betroffen sein, die auf TCBs verweisen, die wiederum den zu rekonfigurierenden Applikationen zugeordnet sind.

3.1.2. Zugriffsdomänen

Ein *Team* unter MOOSE definiert eine Domäne, mit der bestimmte Attribute assoziiert werden, die für alle Prozesse des *Teams* ihre Gültigkeit besitzen. *Rescheduling* und *Swapping* betrifft immer das gesamte *Team* und demzufolge alle Prozesse des *Teams*. Der Zugriff auf bestimmte Objekte des Kernels, d.h., die Zuordnung dieser Objekte an einen bestimmten Prozeβ, wird nur möglich sein, wenn das jeweilige *Team* mit entsprechenden Zugriffsattributen versehen ist.

Die für die Modellierung eines prozeβorientierten Systems bedeutsamen Zugriffsattribute eines *Teams* ergeben sich im Zusammenhang mit der Interprozeβkommunikation unter MOOSE. Sobald eine Kommunikation zwischen Prozessen, die unterschiedlichen *Teams* zugeordnet sind, eingeleitet worden ist, kann diese von beliebigen Prozessen des *Teams*, mit dem die Kommunikation durchgeführt wird, weitergeführt und beendet werden. Das bedeutet also, daβ alle Prozesse eines *Teams*, mit dem die Kommunikation durchgeführt wird, Zugriff auf den Prozeβ des *Teams* besitzen, von dem die Kommunikation ausgegangen ist. Im Zusammenhang mit der Darstellung der Mechanismen zur Interprozeβkommunikation des MOOSE-Kernels wird auf diese wesentliche Semantik von *Teams* noch genauer eingegangen.

3.1.3. Ausführungsdomänen

Zur Ausführung von Prozessen verfolgt der Kernel eine zweistufige Strategie. Einerseits versucht der Kernel nach dem Lokalitätsprinzip zu verfahren. Dies bedeutet, im Zusammenhang mit der Ausführung von Prozessen wird versucht, solange als möglich innerhalb eines *Teams* zu bleiben.

Team-Wechsel im Zusammenhang mit der Ausführung von Prozessen – und dies stellt die zweite Stufe der Ausführungsstrategie des Kernels dar– erfolgen nur aufgrund zweier Gegebenheiten. Die erste Situation ist die, daβ innerhalb eines *Teams* kein Prozeβ mehr ausführbereit ist. Dieser Fall wird auf die Anwendung der blockierend wirkenden Mechanismen zur Interprozeβkommunikation zurückzuführen sein. Die zweite Möglichkeit ist, daβ die Zeitscheibe für das gegenwärtig aktive *Team* abgelaufen ist. Dieser Fall tritt auch dann ein, wenn nur noch ein Prozeβ innerhalb des *Teams* aktiv ist und dieser Prozeβ eigenständig (und zwar ohne zu blockieren) die Zeitscheibe abgibt.

Alle ausführbereiten *Teams* werden vom Kernel in einem *Dispatch-Set* zusammengefaßt. Dieser *Dispatch-Set* wird durch eine zyklische und doppelt verkettete Liste von TCBs repräsentiert. Mit dieser Organisation werden dem Kernel mehrere Prozesse zur Ausführung zur Verfügung gestellt. Der gegenwärtig aktive (vom Kernel zur Ausführung gebrachte) Prozeß wird durch einen Zeiger auf seinen PCB, *curr_proc*, vermerkt. Ausgehend von diesem Zeiger, kann der nächste ausführbereite Prozeß aufgrund zwei verschiedener Strategien selektiert werden:

a) der Kandidat zur Zuteilung des Prozessors ist der direkte Nachfolger von *curr_proc* in der *team*-lokalen *Ready-List*. Der betreffende Prozeß ist damit demselben *Team* zugeordnet wie der durch *curr_proc* identifizierte Prozeß;

b) der Kandidat zur Zuteilung des Prozessors ist am Kopf der *Ready-List* des *Teams*, welches als direkter Nachfolger, des mit *curr_proc* assoziierten *Teams*, im *Dispatch-Set* erreichbar ist. Der betreffende Prozeß ist damit einem anderen *Team* zugeordnet –jedoch nur, wenn der Sonderfall nicht betrachtet wird, daß nur ein *Team* im *Dispatch-Set* enthalten ist.

In Abbildung 3.3 ist beispielhaft ein *Dispatch-Set*, bestehend aus zwei *Teams* mit insgesamt fünf Prozessen, skizziert.

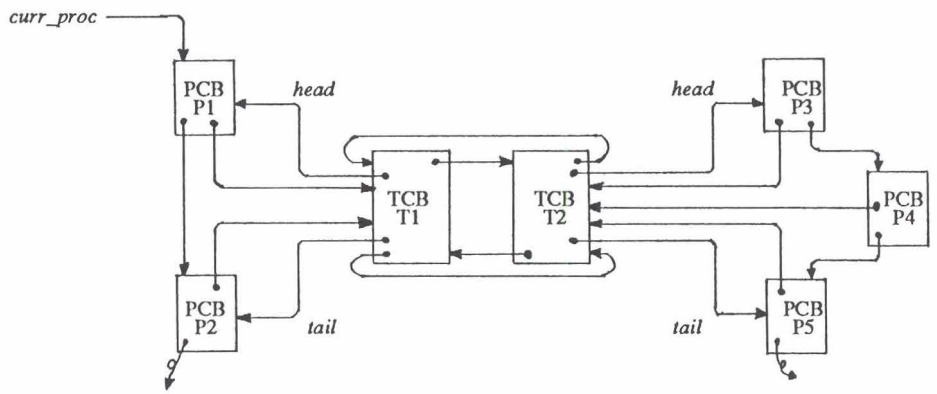

Bild 3.3: Aufbau eines *Dispatch-Set*

In diesem Beispiel ist P1 der gegenwärtig aktive Prozeß, der zu einem früheren Zeitpunkt vom Kernel den Prozessor zugeteilt bekommen hat. Die Selektion eines neuen ausführbaren Prozesses entsprechend Fall a), führt direkt zu P2. Das *Team* T1 wird hierbei nicht gewechselt. Die Prozeßumschaltung wird in diesem Fall nur den *Context-Descriptor* des Prozesses P2 betrachten, um den Laufzeitkontext für P2 zu definieren. Die Selektion eines ausführbereiten Prozesses entsprechend Fall b), führt indirekt zu P3. Über *curr_proc* wird das gegenwärtig aktive *Team* T1 bestimmt. Die Nachfolgerelation im *Dispatch-Set* führt von T1 zu T2. Die Position am Kopf der *team*-lokalen *Ready-List* von T2 nimmt P3 ein. Die Prozeßumschaltung von P1 zu P3 bedingt in diesem Fall den Zugriff auf Informationen des

Image-Descriptors von T2 und des *Context-Descriptors* von P3, da ein *Team*-Wechsel vollzogen wird.

Unter welchen Bedingungen ein Prozeβwechsel und damit die Manipulation des *Dispatch-Set* bzw. der *team*-lokalen *Ready-List* stattfindet, ist im einzelnen den nachfolgenden Abschnitten zu entnehmen.

3.1.3.1. Blocking

Die Anwendung blockierend wirkender Interprozeβkommunikationsprimitiven führt dazu, daβ der betreffende Prozeβ den Prozessor zugunsten der Ausführung eines anderen Prozesses abgibt. Der gegenwärtig aktive Prozeβ blockiert.

Die Blockierung eines Prozesses bewirkt, daβ er aus der *Ready-List* seines *Teams* entfernt wird. Ein anderer Prozeβ aus dieser Liste, soweit vorhanden, bekommt den Prozessor zugeteilt. Hierzu sieht es die Strategie des Kernels vor, daβ immer der Prozeβ am Kopf einer *Ready-List* den aktiven Prozeβ des jeweiligen *Teams* darstellt. Die Blockierung des aktiven Prozesses bedeutet dann, daβ der nachfolgende Prozeβ in der betreffenden *Ready-List* als neuer aktiver Prozeβ des *Teams* ausgewählt wird. Bild 3.4 skizziert diesen Sachverhalt.

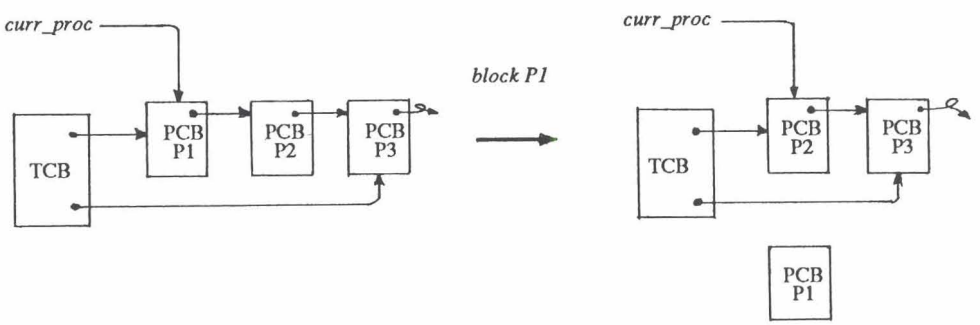

Bild 3.4: Blockierung eines Prozesses

Sollte nach der Blockierung des aktiven Prozesses in seinem *Team* kein weiterer Prozeβ ausführbereit sein, so blockiert das *Team*. Das bedeutet, das *Team* wird aus dem *Dispatch-Set* des Kernels entfernt und es wird versucht ein neues *Team* aus diesem *Dispatch-Set* zur Ausführung zu bringen. Hierzu wird die direkte Nachfolgerrelation in dem *Dispatch-Set* betrachtet.

Ein Sonderfall tritt ein, wenn der *Dispatch-Set* nach der Blockierung eines *Teams* leer ist. In dieser Situation ist kein Prozeβ im System ausführbereit, obgleich der *Dispatch-Set* noch genau einen TCB enthält – die *Ready-List* des zugehörigen *Teams* ist jedoch leer. Wenn die Ausnahmesituation eines systemglobalen *Deadlocks* in dieser Situation vernachlässigt wird, kann die Reaktivierung von Prozessen in diesem Zusammenhang nur noch aufgrund externer

Ereignisse, die über Interrupts angezeigt werden, erfolgen. Ist die Blockierung eines Prozesses aufgrund eines Systemdienstes, der Ein-/Ausgabevorgänge erfordert, erfolgt, bedeutet dies, daß die Prozesse direkt oder indirekt auf die Fertigstellung der Ein-/Ausgabeanforderungen warten.

Der Kernel beginnt in einer solchen Konstellation seine *Idle*-Phase. In dieser Phase wartet der Kernel auf das Eintreten von Interrupts. Die Interrupts würden zur Fertigstellung von Ein-/Ausgabeoperationen führen und können demzufolge auch die Reaktivierung der darauf wartenden Prozesse bewirken.

3.1.3.2. Dispatching

Die Abgabe des Prozessors zugunsten der Ausführung eines anderen Prozesses kann auch ohne die Blockierung des aktiven Prozesses eines *Teams* erfolgen. Hierfür sind, wie bereits erwähnt, zwei Ereignisse verantwortlich; einerseits die abgelaufene Zeitscheibe des *Teams* und andererseits die explizite Abgabe des Prozessors durch den betreffenden Prozeß selbst. Für den ersten Fall ist die Uhr des Kernels und für den zweiten Fall die *dispatch*-Primitive verantwortlich.

Obgleich in beiden Fällen der aktive Prozeß nicht blockiert, sind beide Situationen nur in einer bestimmten Konstellation identisch. Diese Konstellation ergibt sich dann, wenn nur ein Prozeß in einem *Team* ausführbereit ist. In diesem Fall nämlich wird ein *Team*-Wechsel auch bei der Anwendung von *dispatch* stattfinden.

Sollte mehr als ein Prozeß in dem *Team* ausführbereit sein, so wird bei Anwendung von *dispatch* der aktive Prozeß des *Teams* ausgewechselt. Der die Kontrolle abgebende Prozeß bleibt jedoch in der *Ready-List* seines *Teams* enthalten und steht somit weiterhin zur Ausführung bereit. Bild 3.5 zeigt diese Situation.

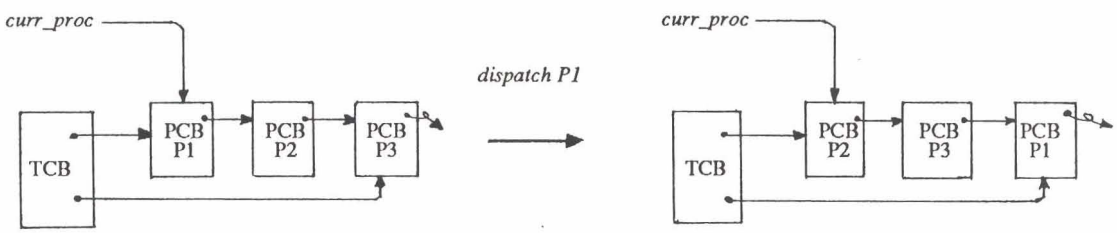

Bild 3.5: Austausch des aktiven Prozesses eines *Teams*

Wie aus diesem Beispiel zu ersehen ist, wechselt der betreffende Prozeß lediglich vom Kopf an das Ende der jeweiligen *Ready-List*. Diese Vorgehensweise entspricht dem grundlegenden Prinzip der Aufnahme von ausführbereiten Prozessen in ein *Team*: Prozesse, die

blockiert gewesen sind, werden bei ihrer Deblockierung immer an das Ende der *Ready-List* ihres *Teams* plaziert[21].

Die Deblockierung eines Prozesses kann ebenfalls zur Deblockierung eines *Teams* führen. Dies ist dann der Fall, wenn der zu deblockierende Prozeβ zu diesem Zeitpunkt der einzige ausführbereite Prozeβ seines *Teams* ist. Das entsprechende *Team* wird dann wiederum in den *Dispatch-Set* des Kernels mit aufgenommen. An welcher Stelle das *Team* in den *Dispatch-Set* eingetragen wird, richtet sich nach der Dringlichkeit der Reaktivierung dieses *Teams*.

Die Deblockierung eines *Teams* führt normalerweise dazu, daβ das *Team* als Vorgänger des gegenwärtig aktiven *Teams* in den *Dispatch-Set* mit aufgenommen wird. Die andere Alternative sieht vor, das *Team* als Nachfolger des aktiven *Teams* zu betrachten und entsprechend in dem *Dispatch-Set* zu plazieren. Die Abgabe der Kontrolle des aktiven *Teams* würde in diesem Fall das zuletzt deblockierte *Team* vor allen anderen *Teams* in dem *Dispatch-Set* den Vorzug geben.

3.1.3.3. Preemptive Dispatching

Die Strategie im Zusammenhang mit der Deblockierung eines Prozesses gestattet es, die üblichen Synchronisationsprobleme, im Zusammenhang mit dem Zugriff auf gemeinsame Datenbestände, zu minimieren. Vollständig vernachlässigt werden können diese Probleme jedoch nicht.

Die bisher dargestellte Strategie der Vergabe des Prozessors an Prozesse desselben *Teams* gestattet einen geregelten Ablauf beim Zugriff auf die gemeinsamen Datenbestände des *Teams*. Die Prozesse selbst können sich in elementarer Weise synchronisieren und bestimmen letztendlich, wann sie ihre Ausführung zugunsten eines anderen Prozesses unterbrechen. Dies entspricht vollständig dem *Coroutinen*-Prinzip[22].

Bei bestimmten Applikationen ist es jedoch sinnvoll mit dieser Strategie zu brechen. Dies wird insbesondere dann notwendig sein, wenn sehr viel Prozesse innerhalb eines *Teams* angesiedelt sind, bestimmte Prozesse aus diesem *Team* jedoch mit höherer Priorität bei der Reaktivierung betrachtet werden sollen. Unter besonderen Umständen ist es somit sehr wohl möglich, daβ ein Prozeβ bei seiner Deblockierung an den Kopf der *Ready-List* seines *Teams* gelangt. Dies stellt nicht den Normalfall dar, sondern findet nur bei der Deblockierung von speziell attributierten Prozessen, den *Interrupt-Tasks*, statt. Bild 3.6 zeigt, wie die Deblockierung einer *Interrupt-Task* sich auf die *Ready-List* ihres *Teams* auswirkt.

Interrupt-Tasks erzielen ihre Wirkung jedoch nur dann, wenn sie von Prozessen anderer *Teams* deblockiert werden. Das *Team* der *Interrupt-Task* ist demzufolge zum Zeitpunkt der Deblockierung dieses Prozesses nicht aktiv gewesen.

Unabhängig davon, weshalb ein *Team* die Kontrolle über den Prozessor abgegeben hat, gilt normalerweise, daβ zwischen zwei aktiven Phasen eines *Teams* der jeweils aktive Prozeβ nicht wechselt. Nur mit diesem Prinzip kann der Zugriff auf die gemeinsamen Datenbestände eines

[21] Eine Ausnahme bilden die sogenannten *Interrupt-Tasks*.

[22] Die Prozesse innerhalb eines *Teams* können als *Coroutinen* aufgefaβt werden, mit denen eine eindeutige, vom System vorgegebene, Prozeβidentifikation verbunden ist.

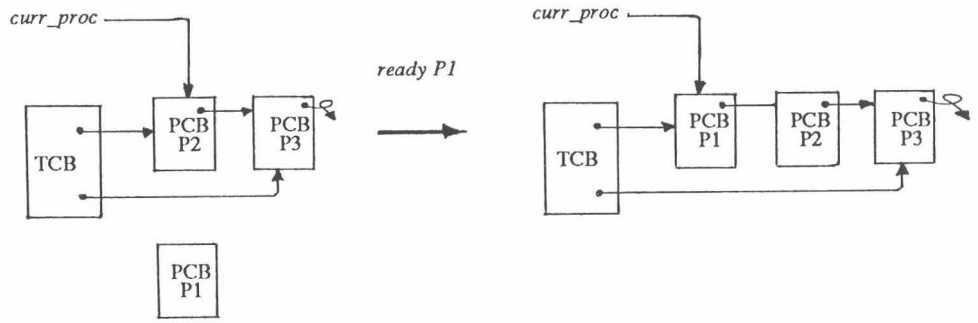

Bild 3.6: Deblockierung einer *Interrupt-Task*

Teams von den Prozessen des *Teams* eigenständig geregelt werden. Die *Interrupt-Tasks* durchbrechen dieses Prinzip, da sie nur dann als aktiver Prozeβ in die *Ready-List* eingetragen werden, wenn ihr *Team* nicht aktiv ist. Demzufolge wechselt der aktive Prozeβ eines *Teams*, ohne daβ das betreffende *Team* selbst darüber die Kontrolle besitzt. Die Konsequenz davon ist, daβ die Synchronisationsproblematik beim Zugriff auf die globalen Datenbestände von *Teams*, in denen wenigstens eine *Interrupt-Task* angesiedelt ist, vollständig zum Vorschein tritt.

Wenn in einem *Team* wenigstens eine *Interrupt-Task* angesiedelt ist, obliegt die Kontrolle, welcher Prozeβ des betreffenden *Teams* ausführbereit ist, nicht dem aktiven Prozeβ dieses *Teams* selbst. Die Aktivierung von Prozessen innerhalb desselben *Teams* muβ somit nicht nur von internen Ereignissen des entsprechenden *Teams* abhängen, sondern sie kann auch durch externe Ereignisse für dieses *Team* herbeigeführt und somit nicht *team*-lokal kontrolliert werden.

3.1.3.4. Slicing

Die Aktivierung eines *Teams* erfolgt implizit durch Ablauf der Zeitscheibe oder explizit durch Blockierung bzw. eigenständige Abgabe der Kontrolle eines anderen *Teams*.

Zur Aktivierung eines anderen *Teams* wird einfach das Nachfolger-*Team* des abgebenden *Teams* ausgewählt. Dies kann insbesondere das abgebende *Team* selbst sein, wenn der *Dispatch-Set* nur aus einem *Team* besteht. Mit jedem neu ausgewählten *Team* wird gleichzeitig eine neue Zeitscheibe aufgesetzt. Dies erfolgt unabhängig davon, weshalb die Kontrolle des vorigen *Teams* abgegeben worden ist.

Eine Zeitscheibe wird für jedes *Team* durch zwei Komponenten bestimmt. Erstens, die maximale Größe der Zeitscheibe, *slice*, und zweitens, der aktuell genutzte Anteil der Zeitscheibe, *ticks*. Mit jedem *Clock-Tick* wird *ticks* um 1 inkrementiert. Erreicht *ticks* die Größe der Zeitscheibe *slice*, so kann ein *Team*-Wechsel erfolgen.

Ein solcher *Team*-Wechsel wird hier nur dann nicht erfolgen, wenn der gegenwärtig aktive Prozeβ als unteilbar attributiert ist. Solche Prozesse werden benötigt, um kritische Bereiche im Zusammenhang mit der Manipulation von PCBs und TCBs in MOOSE implementieren zu können.

Erfolgt ein *Team*-Wechsel aufgrund der abgelaufenen Zeitscheibe des betreffenden *Teams*, so wird *ticks* für die nächste Aktivierung des *Teams* auf 0 gesetzt. Das *Team* erhält somit eine neue Zeitscheibe zugeordnet, die durch *slice* festgelegt ist. Erfolgt jedoch ein *Team*-Wechsel aufgrund der Blockierung des aktiven *Teams*, so behält *ticks* seinen aktuellen Wert bei. Mit seiner Reaktivierung wird dieses *Team* mit dem bisher genutzten Anteil seiner Zeitscheibe wieder als ausführbereit markiert.

3.1.3.5. Scheduling

Der MOOSE-Kernel realisiert nur sehr elementare, dafür jedoch sehr laufzeiteffiziente Mechanismen zur Aktivierung und Deaktivierung von Prozessen bzw. *Teams*. Diese Mechanismen basieren im wesentlichen auf sehr einfachen Operationen (*round-robin*) im Zusammenhang mit der jeweiligen *Ready-List* eines *Teams* und dem *Dispatch-Set*. Die Kommunikationsaktivitäten der Prozesse bestimmen im wesentlichen die Menge der ausführbereiten Prozesse des Systems.

Diese Menge kann explizit manipuliert werden, indem *Teams* aus dem *Dispatch-Set* entfernt oder in den *Dispatch-Set* wieder aufgenommen werden. Hierzu stellt der MOOSE-Kernel entsprechende Mechanismen zur Verfügung. Die von der Ausführung explizit ausgeschlossenen *Teams* beinhalten immer ausführbereite Prozesse. Diese Prozesse werden jedoch erst dann wieder zur Ausführung kommen, wenn das entsprechende *Team* wieder in den *Dispatch-Set* mit aufgenommen wird.

Die explizite Manipulation des *Dispatch-Sets* ermöglicht die Integration komplexer *Schedule*-Strategien in das System. Der *Scheduler* ist unter MOOSE ein Systemprozeβ, der den *Dispatch-Set* des Kernels manipuliert, um eine Gruppe von *Teams* (ein neues *Schedule*) zur Ausführung zu bringen.

Die Aktivierung eines *Schedulers* erfolgt aufgrund einer sehr einfachen und laufzeiteffizienten Strategie. Der *Scheduler* ist immer im jeweiligen Zustand des *Dispatch-Sets* des Kernels enthalten. Damit wird der *Scheduler* automatisch vom Kernel reaktiviert, wenn der entsprechende TCB in dem *Dispatch-Set* zur Ausführung selektiert wird. Dies erfolgt durch die üblichen Strategien des Kernels zur Aktivierung von Prozessen. In diesem Fall würde der Kernel nie seine *Idle*-Phase betreten, da mindestens immer der *Scheduler* im *Dispatch-Set* enthalten sein wird.

Mit diesem Ansatz können komplexe *Schedule*-Strategien aus dem Kernel heraus- und in bestimmte Systemprozesse hineinverlagert werden. Der Kernel hat keine Information über das Vorhandensein eines *Schedulers* im System. Er würde den *Scheduler*, wie jedes andere *Team* auch, über die üblichen Mechanismen aktivieren. Die Konsequenz davon ist, daβ *Schedule*-Strategien zur Laufzeit des Systems dynamisch ausgetauscht werden können.

Zur Manipulation des *Dispatch-Sets* gibt es zwei Ansätze. Einerseits durch Anwendung entsprechender Primitiven und andererseits, indem es dem *Scheduler* ermöglicht wird, direkt

auf den *Dispatch-Set* zuzugreifen. Der letztere Fall würde über *Shared-Memory* zur Verfügung gestellt werden.

Beide Varianten müssen jedoch davon ausgehen, daß die Manipulation des *Dispatch-Set* eine unteilbare Operationsfolge darstellt. Für solche Zwecke ist es daher möglich (und auch notwendig) Prozesse als "unteilbar" zu markieren. Die Unteilbarkeit unter MOOSE ist ein bestimmtes Prozeßattribut und unterbindet einen *Team*-Wechsel, wenn bei der Ausführung des betreffenden Prozesses die Zeitscheibe seines *Teams* ablaufen sollte.

Nach der Manipulation des *Dispatch-Sets,* um einen neuen *Schedule* zur Ausführung zu bringen, gibt der *Scheduler* explizit die Kontrolle mittels *dispatch* ab. Damit sind zwei wesentliche Aspekte verbunden. Erstens startet der *Scheduler* damit die Abarbeitung des *Dispatch-Sets* durch den Kernel, da ein *Team*-Wechsel vollzogen werden wird, und zweitens bleibt der *Scheduler* im *Dispatch-Set* enthalten. Der *Scheduler* kann somit wiederholt aktiviert werden, was bedeuten würde, daß der zuletzt berechnete *Schedule* vom Kernel abgearbeitet worden ist. Nach der Reaktivierung des *Schedulers* sind im *Dispatch-Set Teams* enthalten, die entweder noch vom alten *Schedule* her stammen oder neu, aufgrund ihrer Deblockierung, hinzugekommen sind. Ebenso sind *Teams* nicht mehr enthalten, die bei Abarbeitung des *Scheduls* durch den Kernel blockierten.

3.2. Identifikation von Prozessen

Die effiziente Identifikation von Prozessen ist ein wesentlicher Schlüssel dafür, einen sehr leistungsfähigen Kernel zu gestalten. Hierbei ist die elementare Abbildung von Prozeßidentifikationen auf die entsprechenden Datenstrukturen zur Verwaltung von Prozessen von zentraler Bedeutung. Dies gilt insbesondere im Zusammenhang mit den Mechanismen zur Interprozeßkommunikation.

Die Identifikation der Prozesse richtet sich danach, auf welcher Ebene des Systems – auf der Prozeßebene oder auf der Kernelebene – sie angewendet wird. Die explizite Identifikation von Prozessen ist nahezu mit jeder Primitive des Kernels verbunden. Sie wird auf Prozeßebene benötigt, um für bestimmte vorgegebene Prozesse Verwaltungsmaßnahmen tätigen zu können. Die implizite Identifikation von Prozessen gilt immer, um den gegenwärtig aktiven Prozeß (*curr_proc*) zu adressieren, der vom Kernel den Prozessor zugeteilt bekommen hat. Diese Identifikation wird auf der Kernelebene bei jeder Kernelprimitive in Anwendung gebracht, um Zugriffsrechte des aktiven Prozesses überprüfen und für diesen Prozeß selbst Verwaltungsmaßnahmen durchführen zu können.

3.2.1. Process IDs und Team IDs

Mit jedem Prozeß ist eine eindeutige Identifikation assoziiert, die sogenannte *process ID*. Diese *process ID* wird mit jeder Erzeugung eines Prozesses neu generiert. Sie wird von den Prozessen innerhalb des Systems verwendet, um andere Prozesse (bzw. sich selbst) zu adressieren, bzw. bestimmte (Verwaltungs-) Informationen mit diesen Prozessen assoziieren zu können.

Zusätzlich zu der *process ID* ist jedem Prozeß eine *Team ID* zugeordnet. Die *Team ID* gibt an, in welchem *Team* der Prozeß angesiedelt ist. Operationen im Zusammenhang mit der Angabe von *Team IDs* adressieren somit alle in dem jeweiligen *Team* angesiedelten Prozesse.

Mit der impliziten wie auch expliziten Identifikation ist immer eine Referenz auf den PCB des entsprechenden Prozesses verbunden. Damit der Kernel den hohen Laufzeitanforderungen genügen kann, gilt es somit, einen effizienten Mechanismus zugrunde zu legen, mit dem eine möglichst direkte Abbildung von der expliziten Identifikation (*process ID*) zum PCB möglich ist.

Die implizite Identifikation ist durch einen Zeiger (*curr_proc*) auf den entsprechenden PCB des aktiven Prozesses gegeben und ermöglicht demzufolge innerhalb des Kernels den schnellen Zugriff auf Informationen des aktiven Prozesses. Um mit der expliziten Identifikation eine direkte Abbildung auf den entsprechenden PCB zu erreichen, werden an die Struktur einer *process ID* bestimmte Anforderungen gestellt.

3.2.2. Struktur einer Process ID

Die explizite Identifikation eines Prozesses basiert auf Angabe einer *process ID*. Um in diesem Fall direkt den entsprechenden PCB referenzieren zu können, ist eine *process ID* unter MOOSE aus zwei Komponenten aufgebaut (diese Organisation ist ebenfalls unter THOTH [Cheriton 1982] wiederzufinden):

index identifiziert direkt den Eintrag der *Process-Table*, der den PCB des mittels *process ID* angegebenen Prozesses enthält;

version definiert, wie oft der PCB des mittels *process ID* angegebenen Prozesses wiederverwendet worden ist.

Beide Komponenten werden über einen entsprechenden Datentyp erfaßt, der so ausgelegt ist, daß ein Objekt (die *process ID*) dieses Typs von dem anwendenden Prozeß elementar zu verwalten ist [23]. Um die Identifikation des PCBs für eine gegebene *process ID* effizient gestalten zu können, sind beide Komponenten einer *process ID* so angeordnet, daß *index* die niederwertigen und *version* die höherwertigen Bitpositionen eines *process ID*-Objektes einnehmen. Der *index* kann damit durch eine einfache logische Maskenoperation von einer gegebenen *process ID* hergeleitet und der PCB des explizit identifizierten Prozesses kann vom Kernel direkt referenziert werden.

Die Struktur einer *process ID* bleibt dem anwendenden Prozeß verborgen. Dies gilt ebenso für den Kernel. Die *process ID* wird im MOOSE-System als abstrakter Datentyp aufgefaßt. Demzufolge existieren bestimmte Zugriffsoperationen, die die Implementierungsdetails von einem Objekt dieses Typs verbergen.

Die Zugriffsoperationen im Zusammenhang mit der Verwaltung von *process IDs* sind durch Makros realisiert. Damit wird ein laufzeiteffizienter Zugriff auf eine *process ID* ermöglicht. Diese Realisierung entspricht dennoch den Maximen abstrakter Datentypen, da der

[23] Die *process ID* wird, je nach zugrunde liegendem Prozessor, durch einen 16-Bit oder 32-Bit Wert repräsentiert.

Entwurf bzw. die Konzeption und nicht eine Realisierung die entsprechende Abstraktion aufzuweisen hat [Habermann et al. 1976].

3.2.3. Die Eindeutigkeit der Process ID

Mit der Identifikation von Prozessen besteht das typische Problem, die entsprechende *process ID* eindeutig vergeben zu können. Mehrdeutig ist eine *process ID* dann, wenn zwei voneinander unabhängige Prozesse mit einer *process ID* zwar denselben Prozeβ identifizieren können, dieser Prozeβ jedoch jeweils in einen unterschiedlichen Applikationskontext von den adressierenden Prozessen eingeordnet wird. Diese Situation tritt dann auf, wenn eine *process ID* wiederholt vergeben worden ist – die Lebensdauer einer *process ID* (genauer, eines *process ID*-Objektes) kann größer sein als die Lebensdauer des durch die *process ID* referenzierten Objektes, dem Prozeβ. In diesem Fall bestände keine eindeutige Zuordnung mehr von einer *process ID* zu einem bestimmten diensterbringenden Prozeβ.

Die Wiedervergabe einer *process ID* durch das Betriebssystem wird nach einer bestimmten Folge der Termination und Erzeugung von Prozessen stattfinden. Wann dies der Fall sein wird, ist von der Größe und der Struktur eines *process ID*-Objektes abhängig. Wird eine *process ID* als die *capability* auf einen bestimmten Prozeβ betrachtet, so ließe sich das Mehrdeutigkeitsproblem einer *process ID*, mit den bekannten Verfahren zur Verwaltung von *capabilities*, auf der Systemebene lösen. Diese Verfahren müßten

a) alle lokal von den Prozessen gehaltenen *process IDs* auffinden, dem "*review of capabilities*", und

b) die jeweils an die Prozesse vergebene *process ID* zurückfordern, dem "*revocation of capabilities*".

In [Behr 1983] wird u.a. auf diese Verfahren eingegangen. Ohne die Unterstützung geeigneter Hardware-Architekturen, ist jedoch eine effiziente Lösung des Wiedervergabeproblems auf Grundlage dieser Verfahren nicht möglich. Auch wenn solch eine Unterstützung gegeben ist, bedeutet dies immer noch einen relativ großen Aufwand, *process IDs* für ungültig zu erklären.

Die systemglobale Eindeutigkeit einer *process ID* kann in einfacher Weise dann erreicht werden, wenn die Termination eines Prozesses als eine Ausnahmesituation betrachtet wird. Werden Mechanismen vom Betriebssystem zur Verfügung gestellt, die eine applikationsspezifische Behandlung solcher Ausnahmesituationen ermöglichen, kann die Eindeutigkeit von *process IDs* auf der Applikationsebene gewährleistet werden. Die applikationsspezifische Behandlung der Termination eines Prozesses würde bedeuten, die mit diesem Prozeβ assoziierte *process ID* im Kontext der Applikation für ungültig zu erklären.

In diesem Zusammenhang entscheidet sich dann auch, ob eine Applikation fehlertolerant ist, indem nämlich (für den hier skizzierten Fall) der terminierte Prozeβ als Maβnahme der entsprechenden Applikation wieder erzeugt wird. Ein Betriebssystem kann hierbei nur Hilfsmittel zur Verfügung stellen, die es gestatten, solche Ausnahmesituationen applikationsspezifisch behandeln zu lassen[24]. Die eigentliche Fehlertoleranz muβ jedoch von

[24] MOOSE ermöglicht es, daβ Applikationsprozesse die abnormale Termination von Prozessen erfahren können, die demselben Applikationskontext zugeordnet sind. Dazu werden Mechanismen zur Verfügung

den Applikationssystemen selbst erbracht werden.

Unter MOOSE wird dem Wiedervergabeproblem von *process IDs* durch zwei Maßnahmen begegnet. Erstens wird jedem *index* ein Zähler (*version*) vorangestellt, der angibt, wie oft der *index* wiederholt vergeben, d.h., wie oft der PCB von einem Prozeß in Benutzung gewesen ist, und zweitens wird jedem PCB selbst eine *process ID* zugeordnet.

Damit ergibt sich eine Kreuzreferenz von *process ID* zum PCB und umgekehrt. Die Überprüfung, ob die Zuordnung der mit der expliziten Identifikation vorgegebenen *process ID* eines Prozesses noch gültig ist, bedeutet einen elementaren Vergleich der angegebenen *process ID* mit der im PCB enthaltenen.

3.2.4. Die Generierung einer neuen Process ID

Ob eine *process ID* wiederholt generiert wird – und damit potentiell ihre Eindeutigkeit verliert – ist von zwei Faktoren abhängig. Diese Faktoren sind jeweils durch die Wertebereiche von *index* und *version* vorgegeben.

Mit jeder Zuordnung eines PCBs zu einem Prozeß wird *version* um 1 inkrementiert, wohingegen *index* immer konstant bleibt. Da *version* und *index* jeweils Bitfelder darstellen, werden die Wertebereiche beider Komponenten durch die Größe dieser Bitfelder definiert.

Sei i die Größe des Bitfeldes für *index* und v die Größe des Bitfeldes für *version*. Dann wird die maximale Anzahl von adressierbaren PCBs durch 2^i bestimmt und der Wertebereich für *index* liegt zwischen 0 und 2^i -1. Die Anzahl 2^v -1 gibt an, wie oft ein PCB vergeben werden kann, ohne daß *index* die Eindeutigkeit verliert.

Da die Handhabung einer *process ID* allgemein elementarer Natur sein sollte, gilt für $i+v$, daß es nicht größer als die Wortbreite (üblicherweise 16-Bit oder 32-Bit) des zugrunde liegenden Prozessors sein darf. Damit ergibt sich für i und v, zumindest bei einer Wortbreite von 16-Bit, eine Einschränkung des zur Verfügung stehenden Spielraumes für die jeweiligen Wertebereiche. Je größer v gewählt werden kann, umso geringer ist die Wahrscheinlichkeit, daß eine *process ID* wiederholt vergeben wird. Je größer i gewählt werden kann, umso mehr Prozesse können über die *Process-Table* adressiert werden. Beide Wertebereiche wirken somit gegeneinander und es gilt genau abzuwägen, wie i und v zu bestimmen sind, um die günstigste Konstellation zu erreichen.

In THOTH [Cheriton 1982] wird die *process ID* nach dem gleichen Verfahren bestimmt. Es hat sich dabei gezeigt, daß sich dieses Verfahren hinsichtlich der Eindeutigkeit von *process IDs* und der Effizienz bei der Generierung von *process IDs*, wie auch bei der Identifikation von Prozessen (PCBs), auszeichnet.

In UNIX ist es z.B. üblich, eine *process ID* dadurch zu vergeben, indem die *Process-Table* solange nach *process IDs* durchsucht wird, bis die Eindeutigkeit eines mit jedem Durchlauf inkrementierten Zählers sichergestellt ist. Das bedeutet, daß in der *Process-Table* keine *process ID* existiert, die mit dem neu bestimmten Zählerwert übereinstimmt. Dieses Verfahren

---

gestellt, die es gestatten, Prozeßgruppen zu formulieren und die Termination von Prozessen aus solchen Gruppen anzuzeigen.

ermöglicht es, an eine *process ID* keine Formatanforderungen stellen zu müssen. Es führt jedoch auch dazu, daß die Generierung einer *process ID* und die Identifikation von prozeßzugeordneten Informationen (z.B. den PCB, aber auch von Systemprozessen verwaltete Datenstrukturen für jeden Prozeß) sehr laufzeitintensiv ist.

In MOOSE wird das in THOTH realisierte Verfahren dadurch erweitert, daß mit jeder neuen *process ID* auch versucht wird, einen neuen PCB zu alloziieren. Dazu wird ein *round robin* Algorithmus bei der Vergabe eines PCBs angewendet. Mit dieser Vergabestrategie ist gleichzeitig die Generierung einer neuen *process ID* verbunden, da *index* seinen Wert verändert haben muß, zumindest wenn noch mehr als ein PCB frei zur Disposition steht. Demzufolge richtet sich unter MOOSE die Wiedervergabe einer *process ID* nicht allein nach n. Dies wird nur unter äußerster Belastung des Systems der Fall sein, dann nämlich, wenn immer nur noch ein Prozeß erzeugt werden kann, da nur noch ein PCB zur Verfügung steht. Unter normaler Systembelastung führt somit die *round robin* Vergabe von PCBs dazu, die Wiedervergabe einer *process ID* herauszuzögern.

3.2.5. Die Abbildung der Process ID auf einen PCB

Die Struktur einer *process ID* ermöglicht es bereits, ein sehr effizientes Verfahren in Anwendung zu bringen, um den der *process ID* zugeordneten PCB herleiten zu können. Eine einfache logische Maskenoperation auf ein *process ID*-objekt reicht aus, um *index* zu extrahieren, damit die *Process-Table* zu indizieren und direkt den entsprechenden PCB zu referenzieren.

Diese Abbildung läßt sich noch erheblich effizienter gestalten, wenn beachtet wird, daß mit jeder Indizierung üblicherweise eine Multiplikation von *index* mit der Größe eines PCBs verbunden ist (das sogenannte *scalling* eines Indexes). Der damit verbundene Mehraufwand läßt sich minimieren, wenn das Objekt einer Tabelle (hier der PCB) möglichst klein gehalten wird.

Der Kernel führt solch eine Abbildung durch, um mit der Adresse des PCBs arbeiten zu können, d.h., die Indizierung wird üblicherweise nur einmal stattfinden, nämlich um die Adresse des PCBs in der *Process-Table* zu berechnen. Es bietet sich daher an, eine Abbildungstabelle einzuführen, die es ermöglicht, anhand einer vorgegebenen *process ID* direkt die Adresse des PCBs bestimmen zu können. Diese Tabelle, die sogenannte *Process-Map*, enthält mindestens so viele Einträge wie die *Process-Table* und kann demzufolge mit *index* indiziert werden. Jeder Eintrag in der *Process-Map* enthält bereits die Adresse auf den entsprechenden PCB der *Process-Table*. Da eine Adresse auf einen PCB weniger an Platz benötigt als der PCB selbst, ist demzufolge die Indizierung in die *Process-Map* laufzeiteffizienter als die Indizierung in die *Process-Table*. Die Initialisierung der *Process-Map* findet einmal statt und zwar während der Initialisierungsphase des Kernels.

Die *Process-Map* hilft zusätzlich, einen bestimmten Nebeneffekt zu vermeiden, der aufgrund der Organisation einer *process ID* auftritt. Aus Gründen der Laufzeiteffizienz findet bei der Applikation von *index* keine Überprüfung statt, ob der momentane Wert dieser Komponente im Wertebereich von 0 bis 2^i -1 liegt. Eine korrekte Applikation von *index* wäre nur dann jederzeit gegeben, wenn 2^i Objekte (PCBs) zur Verfügung ständen. Dies

sicherzustellen kann jedoch einen groβen Bedarf an Speicherplatz für die *Process-Table* bedeuten. Der benötigte Speicherplatz wächst insbesondere exponentiell, wenn die *Process-Table* erweitert werden soll.

Diese Tatsache gilt auch für die *Process-Map*. Jedoch benötigen in dieser Tabelle die einzelnen Objekte erheblich weniger Speicherplatz als die Objekte der *Process-Table*. Durch eine geeignete Repräsentation der Beziehung zwischen *Process-Map* und *Process-Table* ist es möglich, den benötigten Speicherplatz für die *Process-Table* zu minimieren. Diese Repräsentation würde vorsehen, die *Process-Map* immer mit 2^i Objekten auszustatten und die *Process-Table* nur mit soviel (*nproc*) Objekten zu konfigurieren, wie für ein bestimmtes System vorhanden sein sollen. Bild 3.7 gibt ein Beispiel für solch eine Repräsentation.

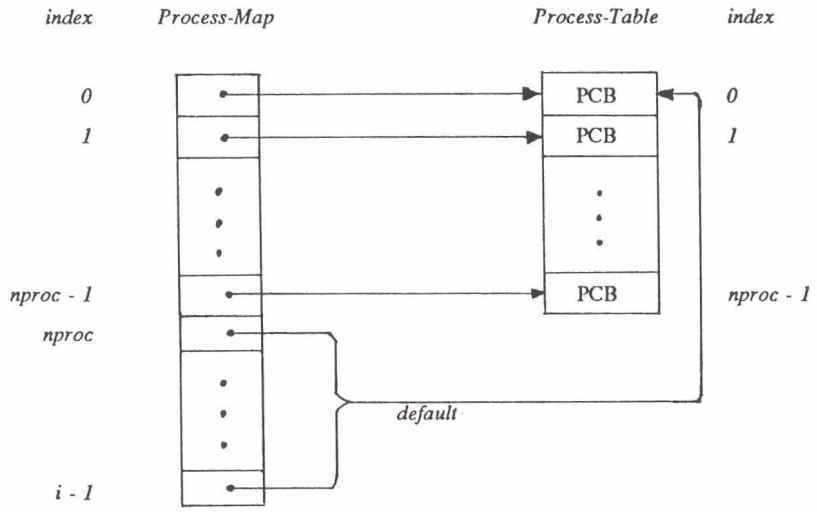

Bild 3.7: Beziehung zwischen *Process-Map* und *Process-Table*

Jeder *index*, für den kein gültiger PCB vorhanden ist, wird auf einen *default*-PCB abgebildet. Dieser PCB identifiziert immer Prozeβ 0, der repräsentativ für den Kernel vorhanden ist. Mit dieser Technik kann für jeden *index* jederzeit eine Abbildung auf einen PCB erzielt werden.

3.3. Mechanismen zur Interprozeβkommunikation

Die Mechanismen zur Interprozeβkommunikation unterteilen sich in zwei Kategorien. Die eine Kategorie erlaubt den Austausch von Nachrichten auf Grundlage des *Rendezvous*-Konzeptes. Die dazu in Anwendung zu bringenden Mechanismen wirken blockierend für den sendenden und den empfangenden Prozeβ. Die andere Kategorie erlaubt die Übermittlung von Signalen.

Signale selbst sind eine besondere Form von Nachrichten. Sie repräsentieren Nachrichten ohne Inhalt, d.h., es wird bei der Übermittlung eines Signals kein Datentransfer erfolgen. Die Konsequenz daraus ist, daβ die Übermittlung eines Signals nichtblockierend auf den jeweils sendenden Prozeβ wirkt, da keine Zwischenpufferung von Nachrichten zu erfolgen braucht.

Unter MOOSE erfolgt die Identifikation der jeweiligen Kommunikationspartner bei dem Austausch von Nachrichten nach anderen Prinzipien als bei der Übermittlung von Signalen. Im Zusammenhang mit Nachrichten wird der Kommunikationspartner durch Angabe seiner *process ID* direkt adressiert. Die Signale dagegen werden indirekt über *Ports* weitergeleitet. Die direkte Adressierung des Kommunikationspartners über seine *process ID* ermöglicht eine sehr laufzeiteffiziente Realisierung der entsprechenden Kommunikationsprimitiven. Die über *Ports* bedingte Indirektion bei der Adressierung des Kommunikationspartners dagegen kann eine erhebliche Laufzeiteinbuße für die Interprozeßkommunikation bedeuten[25].

Die Begründung der zugrunde gelegten Funktionalität für die nachfolgend skizzierten Mechanismen des MOOSE-Kernels zur Interprozeßkommunikation orientiert sich in erster Linie an [Cheriton 1979] und [Gentleman 1981]. In [Gentleman 1981] ist dargestellt, wie konkrete Prozeßsysteme auf Grundlage solcher Kommunikationsmechanismen aufgebaut werden können.

3.3.1. Rendezvous und Message-Passing

Die Anforderung zur Erbringung eines Systemdienstes und die Fertigmeldung des jeweils erbrachten Dienstes wird in einem prozeßorientierten System nach gleichartigen Prinzipien erfolgen. In beiden Fällen müssen Informationen (die Argumente bzw. Resultate) übertragen und der Kontrollfluß zwischen den beiden Prozessen muß geregelt werden. Hierzu bietet sich ein Mechanismus zur Interprozeßkommunikation an, der auf drei elementare Operationen zurückgeführt wird:

send	Übermitteln einer Nachricht und blockieren;
receive	Empfangen einer Nachricht und evtl. blockieren;
reply	Übermitteln von Resultaten und deblockieren des Senders.

Die Konstruktion des Mechanismus' zur Interprozeßkommunikation mit Hilfe dieser drei Primitiven schafft eine Basis, auf deren Grundlage sichere Interaktionsprotokolle zwischen den miteinander kommunizierenden Prozessen formuliert werden können.

Mittels *send* gibt ein Prozeß seine Bereitschaft bekannt, eine Nachricht zu übermitteln und eine andere Nachricht zu empfangen. Das *receive* dient dazu, die zu übermittelnde Nachricht anzunehmen und *reply* wird angewendet, um dem sendenden Prozeß eine andere Nachricht zurückzugeben.

Neben dieser Aufgabe des reinen Datentransfers, implizieren die auf *Message-Passing* basierenden Mechanismen zur Interprozeßkommunikation zusätzlich einen bestimmten Kontrollfluß zwischen den entsprechenden Prozessen. Mit *send* blockiert der sendende Prozeß solange, bis der empfangende Prozeß, im Anschluß an *receive*, ein *reply* initiiert. Ein empfangender Prozeß blockiert mittels *receive* dagegen nur dann, wenn keine Nachricht zum

[25] Die erste Version des MOOSE-Kernels basierte auf einem verbindungsorientierten Modell der Interprozeßkommunikation. *Ports* wurden hierbei jeweils als Endpunkte der jeweiligen Verbindung betrachtet. Mit diesem Modell konnten zwar sehr flexible Kommunikationsarchitekturen aufgebaut werden, der dazu notwendige Verwaltungsaufwand war jedoch erheblich. Die Laufzeiten bei der Interprozeßkommunikation lagen ca. um den Faktor 2 bis 3 höher als bei den Laufzeiten des in dieser Arbeit vorgestellten Kernels.

Empfang bereitsteht. Das erste *send* zu einem blockierten Empfangsprozeβ kann zu dessen Deblockierung führen. Die Rückgabe einer Nachricht durch *reply* führt nicht zur Blockierung des aktiven Prozesses. Tabelle 3.1 zeigt mögliche Zustände der Prozesse bei Anwendung dieser drei grundlegenden Operationen zur Interprozeβkommunikation unter MOOSE.

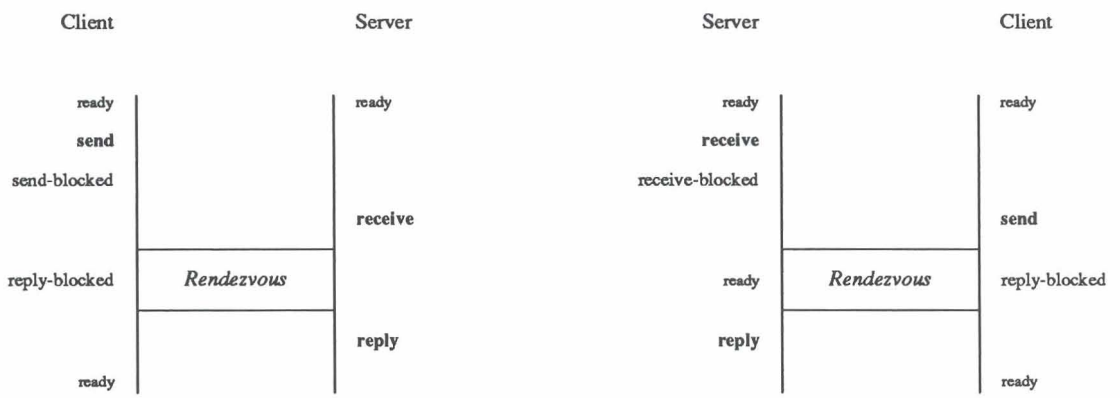

Tabelle 3.1: Prozeβzustände während eines *Rendezvous*

In der Phase zwischen *receive* bzw. *send* und *reply* sind beide an der Kommunikation beteiligten Prozesse aufeinander synchronisiert, d.h., sie befinden sich in einem *Rendezvous*. Während des *Rendezvous* ist von den beiden involvierten Prozessen nur der Empfänger aktiv, sofern dieser sich nicht in ein anderes *Rendezvous* begibt und selbst blockiert. Es liegt somit in der Verantwortung des empfangenden Prozesses, das *Rendezvous* zu beenden und damit die Deblockierung des sendenden Prozesses herbeizuführen.

Der jeweils sendende Prozeβ eines *Rendezvous* wird als *Rendezvous-Client* bezeichnet. Der empfangende Prozeβ stellt den *Rendezvous-Server* dar.

3.3.1.1. Beginn der Rendezvousphase

Damit ein *Rendezvous* zustande kommt, muβ einer der an der Kommunikation beteiligten Prozesse blockiert sein und damit die Bereitschaft für ein *Rendezvous* anzeigen. Dazu muβ das Senden oder Empfangen von Nachrichten von dem betreffenden Prozeβ initiiert worden sein.

3.3.1.1.1. Senderinitiiertes Rendezvous

Der *Rendezvous-Client* erreicht die Übermittlung einer Nachricht mittels *send*. Der *Rendezvous-Server* wird hierbei explizit durch die Angabe der entsprechenden *process ID* identifiziert. Mit der Übermittlung der Nachricht blockiert der *Rendezvous-Client*. Er wird erst dann seine Ausführung wieder aufnehmen können, nachdem er explizit reaktiviert worden ist. Dies wird unter normalen Umständen durch das *reply* des *Rendezvous-Servers* erreicht[26].

[26] In Ausnahmefällen kann die Reaktivierung eines Prozesses auch durch andere Mechanismen erfolgen, als die im Zusammenhang mit dem *Rendezvous* angewendeten.

Das *Rendezvous*-Konzept von MOOSE sieht vor, daß der *Rendezvous-Client*, nach seiner Reaktivierung, auf eine *Reply-Message* Zugriff hat, die ihm mittels *reply* von dem *Rendezvous-Server* zugestellt worden ist. Dies bedeutet, daß dem *Rendezvous-Client* mit der Übermittlung einer Nachricht implizit eine Rücknachricht vom *Rendezvous-Server* zugestellt wird. Das jeweilige Kommunikationsprotokoll zwischen *Rendezvous-Client* und *Rendezvous-Server* definiert hierbei, ob eine *Reply-Message* vom *Rendezvous-Server* zum *Rendezvous-Client* übermittelt werden soll.

Damit der *Rendezvous-Client* darüber informiert ist, mit welchem Prozeß er ein *Rendezvous* durchgeführt hat, liefert *send* jeweils die *process ID* des entsprechenden *Rendezvous-Servers* zurück, der das *Rendezvous* beendet hat[27].

3.3.1.1.2. Empfängerinitiiertes Rendezvous

Der Empfang von Nachrichten erfolgt durch *receive*. Der *Rendezvous-Server* blockiert, wenn keine Nachricht zum Empfang bereitsteht, d.h., wenn noch kein *Rendezvous-Client* eine Nachricht zu dem betrachteten *Rendezvous-Server* übermittelt hat.

Der *Rendezvous-Server* kann den *Rendezvous-Client* explizit identifizieren, indem er bei *receive* die *process ID* des entsprechenden Prozesses angibt. Damit blockiert der *Rendezvous-Server* bis der angegebene *Rendezvous-Client* eine Nachricht zur Übermittlung (mittels *send*) bereitgestellt hat. Diese Variante wird als *specific receive* bezeichnet.

Neben dem *specific receive* bietet der Kernel noch eine andere Variante, das *nonspecific receive*, an. Diese Variante wird von Prozessen angewendet, bei denen von vornherein nicht bekannt ist, wie sich das typische Sende- und Empfangsprofil eines Systems zusammensetzt. Die Systemprozesse stellen hierbei solche Empfangsprozesses dar. Das *nonspecific receive* findet bei diesen Prozessen seine Anwendung, um die Anforderungen zur Erbringung von Systemdiensten annehmen zu können. Mit dem *nonspecific receive* werden Nachrichten in *first-come-first-serve* Strategie vom empfangenden Prozeß angenommen.

Damit, insbesondere im Falle des *nonspecific receive*, der *Rendezvous-Server* darüber informiert ist, mit welchem Prozeß er ein *Rendezvous* durchführt, liefert *receive* jeweils die *process ID* des *Rendezvous-Clients* zurück.

3.3.1.2. Beenden der Rendezvousphase

Ein *Rendezvous* kann auf zwei verschiedene Arten abgeschlossen werden. Zum einen ist der Abschluß eines *Rendezvous* dadurch möglich, daß eine *Reply-Message* an den *Rendezvous-Client* zurückgegeben wird. Zum anderen kann der *Rendezvous-Server* für den *Rendezvous-Client* ein neues *Rendezvous* ermöglichen. Dies erfolgt dadurch, indem die von dem *Rendezvous-Client* übermittelte Nachricht wiederholt übermittelt wird und damit erneut empfangen werden kann. Hierbei kann der Empfängerprozeß selbst wieder den die Nachricht weiterleitenden Prozeß (der *Rendezvous-Server*) darstellen, d.h., der betreffende Prozeß

[27] Die explizite Adressierung des *Rendezvous-Servers* muß nicht bedeuten, daß dieser Prozeß auch das *Rendezvous* beendet. In den nachfolgenden Abschnitten werden dafür die entsprechenden Beispiele genannt.

übermittelt eine Nachricht zu sich selbst. Im Normalfall wird jedoch ein anderer Prozeß bei der wiederholten Übermittlung einer Nachricht als Empfängerprozeß adressiert.

Beim Weiterleiten von Nachrichten werden zwei Strategien unterschieden. Einerseits gibt der weiterleitende Prozeß vor, welcher Prozeß die Nachricht erneut empfangen soll. In diesem Fall wird von einem *specific relay* gesprochen. Andererseits überläßt der weiterleitende Prozeß dem Kernel die Entscheidung, welcher Prozeß die Nachricht erneut empfangen soll. In diesem Fall handelt es sich um ein *nonspecific relay*. Das *nonspecific relay* besitzt im Zusammenhang mit dem *Broadcast*-Modell des MOOSE-Kernels seine zentrale Bedeutung.

Der *Rendezvous-Server* kann mehrere *Rendezvous* zu *Rendezvous-Clients* offenstehen haben. Dies wird dadurch ermöglicht, indem das *Rendezvous*-Konzept in MOOSE keine strenge *send-receive-reply* Folge vorschreibt. Es können mehrere Nachrichten empfangen werden, ohne zwischenzeitlich ein *Rendezvous* zu beenden.

Ist mehr als ein *Rendezvous* offen, so muß demzufolge die Möglichkeit bestehen, selektiv ein bestimmtes *Rendezvous* zu beenden. Dazu wird beim *reply* explizit die *process ID* des Prozesses angegeben, zu dem ein *Rendezvous* besteht und das quittiert werden soll.

3.3.1.3. Implikationen des Rendezvous-Konzeptes

Das *Rendezvous*-Konzept von MOOSE birgt verschiedene Konsequenzen für die anwendenden Prozesse in sich. Von Bedeutung sind hierbei die blockierend wirkenden Mechanismen zur Interprozeßkommunikation und die Beziehung zwischen dem *Rendezvous-Client* und dem *Rendezvous-Server*.

3.3.1.3.1. Deadlock-Erkennung und -Auflösung

Die Erkennung und insbesondere die Auflösung von *Deadlocks* ist ein sehr komplizierter Themenkomplex, der an dieser Stelle nicht vollständig behandelt werden soll. Für den MOOSE-Kernel stellt sich die *Deadlock*-Problematik auch nur im Zusammenhang mit der Interprozeßkommunikation, weshalb genau dieser Bereich hier im Vordergrund steht. Eine allgemeine Darstellung der *Deadlock*-Problematik ist z.B. [Shaw 1974] zu entnehmen.

Die Möglichkeit von *Deadlocks* im Zusammenhang mit der Anwendung der blockierenden Kommunikationsprimitiven, *send* und *receive*, ist jederzeit gegeben. Die Entstehung eines *Deadlocks* kann hierbei auf sehr unterschiedliche Konstellationen bei der Interprozeßkommunikation zurückgeführt werden.

Einfache *Deadlocks* sind möglich, wenn ein Prozeß sich selbst als Kommunikationspartner identifiziert und demzufolge im Zusammenhang mit der Übermittlung und dem Empfang von Nachrichten auf sich selbst blockiert. Eine andere einfache Form existiert, wenn zwei Prozesse sich gegenseitig eine Nachricht zusenden und damit aufeinander blockieren.

Komplizierte Formen von *Deadlocks* im Zusammenhang mit der Interprozeßkommunikation sind dann möglich, wenn eine ausgeprägte Kommunikation zwischen mehreren Prozessen vorausgesetzt wird. Prozesse können bei sehr komplexen Kommunikationsbeziehungen indirekt in einen *Deadlock* geraten, wenn durch die Blockierung dieser Prozesse ein Zyklus aufeinander blockierter Prozesse geschlossen wird.

Die Erkennung von *Deadlocks* aufgrund der Anwendung von Kommunikationsmechanismen ist dann verhältnismäßig einfach, wenn die Mechanismen blockierend auf die anwendenden Prozesse wirken. In dieser Situation existiert ein stabiler Zustand für die sich im *Deadlock* befindlichen Prozesse. Dieser Zustand ist – zumindest für das *Rendezvous*-Konzept von MOOSE – "direkt" ersichtlich. Er kann festgestellt werden, indem die Kommunikationsbeziehung zwischen den einzelnen Prozessen nachvollzogen wird.

Im Zusammenhang mit nichtblockierenden Kommunikationsprimitiven ist dagegen die Erkennung eines *Deadlocks* nicht direkt möglich. Dies hat seine Ursache gerade darin, daß ein stabiler Zustand, wie er im Zusammenhang mit den blockierenden Kommunikationsmechanismen bei einem *Deadlock* vorhanden ist, nicht notwendigerweise eintreten muß. Die Folge davon ist, daß eine Kommunikationsbeziehung zwischen den Prozessen, die aufgrund des entsprechenden *Deadlocks* nachvollziehbar wäre, nicht hergeleitet werden kann.

Im Zusammenhang mit blockierenden Kommunikationsprimitiven ist es sogar möglich, einen *Deadlock* prophylaktisch erkennen zu können (*Deadlock Prevention*). Bevor die angewandte Kommunikationsprimitive (*send* oder *receive*) wirksam wird, kann untersucht werden, ob der aktive Prozeß in einen *Deadlock* hinein gerät. Sollte ein *Deadlock* ersichtlich sein, kann die weitere Ausführung der jeweiligen Kommunikationsprimitive abgebrochen werden.

Dennoch sind die dazu notwendigen Verfahren zu laufzeitintensiv, als daß sie bei jeder in Betracht kommenden Kommunikationsprimitive eingesetzt werden können, um *Deadlocks* zu vermeiden. Der Versuch der Erkennung eines *Deadlocks* bevor eine Kommunikationsprimitive wirksam wird, d.h., bevor der anwendende Prozeß blockiert, würde sich nur im Ausnahmefall auszahlen, nämlich dann, wenn ein *Deadlock* tatsächlich auftreten wird. Im Normalfall würden die entsprechenden Verfahren nur zusätzliche Laufzeiten bei der Interprozeßkommunikation mit sich bringen.

Der MOOSE-Kernel führt demzufolge im Zuge der Interprozeßkommunikation keine Erkennung von *Deadlocks* durch. Dies gilt auch dann, wenn nur ein Prozeß (der einfachste und offensichtlichste Fall) von einem *Deadlock* betroffen sein sollte. So auch im Zusammenhang mit dem Sonderfall, bei dem versucht wird, eine Kommunikation mit bereits terminierten Prozessen einzugehen. Das Konzept von MOOSE sieht es jedoch vor, spezielle Systemprozesse einzusetzen, die das System überwachen und versuchen, *Deadlocks* aufzuspüren. Die Erkennung und demzufolge auch die Behandlung von *Deadlocks* ist vollständig aus dem MOOSE-Kernel herausgezogen worden, zugunsten einer laufzeiteffizienten Realisierung der Kommunikationsprimitiven.

3.3.1.3.2. Benutztrelation zwischen Prozessen

Im Zusammenhang mit einem *Rendezvous* läßt sich eine klare Abhängigkeit des *Rendezvous-Clients* von dem jeweiligen *Rendezvous-Server* erkennen. Der *Rendezvous-Server* besitzt immer die Kontrolle über den *Rendezvous-Client*. Demzufolge ist der *Rendezvous-Client* abhängig von der Korrektheit des *Rendezvous-Servers*, womit eine eindeutige Benutztrelation, [Parnas 1974] und [Parnas 1976], zwischen den beiden Prozessen definiert wird.

Durch die mit einem *Rendezvous* vorgegebene Kommunikationsbeziehung lassen sich Prozesse in eine eindeutige hierarchische Beziehung zueinander bringen. Die Prozesse stellen allgemein bestimmte Dienste zur Verfügung, die von Prozessen hierarchisch höher liegender Ebenen genutzt werden können. Somit lassen sich in elementarer Weise immer komplexere Dienste zusammenstellen, wodurch eine geeignete Beziehung zwischen miteinander kommunizierenden Prozessen aufgebaut wird.

Jeder Prozeβ entspricht mit dieser Sichtweise einem abstrakten Prozessor, der einen bestimmten Satz von Elementaroperationen (die Dienste) implementiert [Schindler 1983]. Demzufolge wirkt sich die Inkorrektheit eines solchen Prozesses direkt oder indirekt auf die mit diesem Prozeβ kommunizierenden Prozesse aus.

Insbesondere bei konsequenter Anwendung des *Team*-Modells zum Aufbau von Applikationssystemen bietet das *Rendezvous*-Konzept des MOOSE-Kernels einen interessanten Ansatzpunkt. Die Beendigung eines *Rendezvous* kann von beliebigen Prozessen innerhalb des *Teams*, in dem der eigentliche *Rendezvous-Server* angesiedelt ist, erreicht werden. Das bedeutet somit, daβ jeder Prozeβ innerhalb eines *Teams* dieselben Zugriffsrechte, im Zusammenhang mit der Interprozeβkommunikation durch die *Message-Passing* Primitiven des MOOSE-Kernels, auf einen *Rendezvous-Client* besitzt. Ein *reply*- oder *relay*-Zugriffsrecht auf einen *Rendezvous-Client* wird somit durch das *Team*-Konzept implizit auf allen Prozessen innerhalb desselben *Teams* übertragen, ohne dazu explizit das entsprechende Zugriffsrecht an einen anderen Prozeβ abgeben ("*relayen*") zu müssen.

3.3.1.3.3. Austausch groβer Datenmengen

Message-Passing unter MOOSE muβ nicht bedeuten, daβ jeglicher Datentransfer nur mit Hilfe der Primitiven zur Interprozeβkommunikation stattfindet. Auf Grundlage des *Rendezvous*-Konzeptes ergeben sich insbesondere für den Transfer groβer und variabler Datenbestände andere interessante Perspektiven. Hierbei stellt der Synchronismus zwischen *Rendezvous-Client* und *Rendezvous-Server* die wesentliche Voraussetzung dar.

Shared-Memory und *capabilities* [Fabry 1974] stellen sinnvolle technische Ergänzungen dar, um während eines *Rendezvous* den Transfer von groβen und insbesondere variabler Datenmengen zu ermöglichen. Diese Mechanismen würden explizit von dem *Rendezvous-Server* in Anwendung gebracht werden und sind nicht implizit mit der Anwendung der Kommunikationsprimitiven vorgegeben.

Ein anderes (hardware-unabhängiges) Verfahren für den Austausch beliebiger Datenmengen wird durch den "DMA-Mechanismus" des MOOSE-Kernels ermöglicht. Dieser Mechanismus ermöglicht den *transfer* von Daten zwischen beliebigen Adreβräumen. Die einzelnen Adreβräume werden hierbei explizit anhand von *process IDs* identifiziert.

Das *Rendezvous*-Konzept ermöglicht für diese drei Varianten einen synchronisierten Zugriff auf den Adreβraum des *Rendezvous-Clients*. Ein Kommunikationsprotokoll zwischen *Rendezvous-Client* und *Rendezvous-Server* kann daraus Nutzen ziehen, wenn beliebige Datenmengen zwischen den beiden Prozessen ausgetauscht werden sollen. Dieses Protokoll könnte es vorsehen, über *Message-Passing* jeweils nur die Zugriffsrechte und die Adressen von bestimmten Datenobjekten im Adreβraum des *Rendezvous-Clients* an den *Rendezvous-*

Server zu übermitteln. Die Anzahl der Objekte könnte, je nach Protokoll, variabel sein. Mit Hilfe von *Shared-Memory*, *capabilities* oder *transfer* würde dann durch den *Rendezvous-Server* der eigentliche Datentransfer stattfinden.

3.3.1.3.4. Semantiken von Kommunikationsdiensten

In [Liskov 1979] sind verschiedene Semantiken der Interprozeβkommunikation dargestellt. Von den drei dort skizzierten Varianten, sind mit dem *Rendezvous*-Konzept des MOOSE-Kernels prinzipiell zwei direkt realisierbar.

Das *synchronization send* wird durch ein sehr einfaches Protokoll zwischen *Rendezvous-Client* und *Rendezvous-Server* realisiert, indem der *Rendezvous-Server* sofort nach Empfang einer Nachricht dem jeweiligen *Rendezvous-Client* eine *Reply-Message* zurückgibt. Das *Rendezvous* wird damit vom *Rendezvous-Server* sofort mittels *reply* beendet und dient lediglich dazu, Daten vom *Rendezvous-Client* übernehmen zu können. Die Verarbeitung der angenommenen Daten durch den *Rendezvous-Server* erfolgt damit nebenläufig zur Erzeugung weiterer Daten durch den *Rendezvous-Client*.

Das *remote invocation send* blockiert den *Rendezvous-Client* solange, bis die vom *Rendezvous-Server* empfangenden Daten verarbeitet worden sind. Die Rückgabe der *Reply-Message* ist damit gleichbedeutend mit der Rückgabe von Resultaten eines *remote procedure calls* [Nelson 1982].

Das *no-wait send* kann durch das *Rendezvous*-Konzept allein nicht realisiert werden. Hierzu müβte der *Rendezvous-Client* bereits vor Empfang einer entsprechenden Nachricht (durch den *Rendezvous-Server*) weiterarbeiten können. Das *Team*-Konzept bietet jedoch die Möglichkeit an, Kommunikationsmechanismen mit der Semantik eines *no-wait send* auch auf Grundlage des *Rendezvous*-Konzeptes des MOOSE-Kernels realisieren zu können. In einem der nachfolgenden Kapitel dieser Arbeit werden solche Kommunikationsmechanismen exemplarisch modelliert.

3.3.2. Signale und Ports

Neben dem Austausch von Nachrichten stellt der MOOSE-Kernel noch andere Möglichkeiten zur Interprozeβkommunikation zur Verfügung. Hierbei handelt es sich um die Übermittlung von Signalen.

Zwischen Signalen und Nachrichten bestehen auf der applizierenden Ebene zwei wesentliche Unterschiede. Erstens ist mit Signalen keine weitere Information verbunden, auβer dem Ereignis der Übermittlung selbst. Es findet kein Datentransfer statt. Zweitens führt die Übermittlung von Signalen nicht zur Blockierung des sendenden Prozesses. Dies ist in MOOSE gerade dadurch begründet, daβ kein Datentransfer stattfindet.

3.3.2.1. Message-Descriptor

Auf der Kernelebene werden Signale und Nachrichten über dieselben Strukturen verwaltet. Signale und Nachrichten werden jeweils über einen *Message-Descriptor* beschrieben. Dieses Verfahren erlaubt es Nachrichten und Signale innerhalb des Kernels gleich behandeln zu können.

Ein *Message-Desriptor* wird den Prozessen über einen *Port* zur Verfügung gestellt. Jeder Prozeβ besitzt implizit bereits einen *Port*, der ihm zum Zeitpunkt seiner Erzeugung zugeordnet wird. Dieser *Port* wird von dem Prozeβ benötigt, damit er Nachrichten übermitteln kann. Dies wird zumindest im Zusammenhang mit den *System-Calls* geschehen, die ja auf Nachrichten abgebildet und den entsprechenden Systemprozessen ihrer Betriebssystemdomäne zugesandt werden. Der *Port* wird daher als *Domain-Port* des Prozesses bezeichnet.

Im Gegensatz zu dem *Domain-Port* kann ein Prozeβ mehrere *Event-Ports* besitzen. Diese *Ports* werden dynamisch von den Prozessen angefordert und wieder freigegeben. Zur Anforderung eines *Event-Ports* dienen zwei Varianten. Einerseits kann die Identifikation des *Event-Ports*, an den sich der Prozeβ anbinden will, vorgegeben werden. In diesem Fall wird von einem *specific attach* gesprochen. Andererseits kann sich der Prozeβ einen beliebigen *Port* vom Kernel zuordnen lassen. In diesem Fall, dem *nonspecific attach*, wird die Identifikation des jeweils belegten *Event-Ports* an den Prozeβ zurückgeliefert.

Mit der Termination eines Prozesses werden alle von ihm belegten (*Domain-* und *Event-*) *Ports* wieder freigegeben und stehen damit erneut frei zur Disposition.

3.3.2.2. Applikation von Ports

Ports ermöglichen einem Prozeβ die Abstraktion von der Ursache eines Signals. Sie werden an Stellen eingesetzt, wo verschiedenartige Komponenten miteinander in Verbindung gebracht werden müssen [Balzer 1971]. In MOOSE sind dies auf der einen Seite Geräte und auf der anderen Seite Prozesse. Prozesse synchronisieren sich hierbei über *Ports* auf das Eintreten eines Signals. Die Signale können explizit durch andere Prozesse erzeugt, aber auch implizit durch bestimmte Aktivitäten eines *Interrupt-Handlers* verursacht werden.

Die Synchronisation auf ein Signal wird durch zwei Varianten ermöglicht. Zum einen das *specific await* im Zusammenhang mit dem *Port*, an dem das Signal erwartet wird, und zum anderen das *nonspecific receive*. Insbesondere im letzteren Fall ist ein Prozeβ somit in der Lage, zugleich auf Signale und Nachrichten warten zu können. Im Falle eines eintreffenden Signals gibt des Resultat von *receive* die Identifikation des *Ports* an, an dem das Signal abgesetzt worden ist. Die Identifikation eines *Ports* unterscheidet sich eindeutig von jeder möglichen *processID*, woraufhin, im Falle des *nonspecific receive*, eindeutig bestimmt werden kann, ob ein Signal oder eine Nachricht empfangen worden ist.

Die Anbindung eines *Ports* an einen Prozeβ definiert gleichzeitig die Besitzerrechte des Prozesses zu dem *Port*. Dieser *Port* kann somit keinem anderen Prozeβ zur Verfügung gestellt werden, es sei denn, der besitzende Prozeβ gibt den *Port* wieder frei. Es kann jedoch die Identifikation des Besitzers eines *Ports* in Erfahrung gebracht werden. Das *specific attach* und das *detach* liefern jeweils die *processID* des Besitzers eines *Ports* zurück, wenn die beiden Primitiven auf einen *Port* angewendet werden, der bereits von einem anderen Prozeβ

belegt ist.

Dieses Verfahren unterstützt bereits die Rekonfiguration eines prozeßorientierten Systems auf sehr einfache Weise. Werden mit einem *Port* bestimmte Dienste assoziiert, so können die dienstanfordernden Prozesse von dem jeweiligen diensterbringenden Prozeß abstrahieren. Damit kann der diensterbringende Prozeß ausgewechselt werden, d.h., der entsprechende *Port* "wechselt" seinen Besitzer[28].

3.3.2.3. Abarbeitungsreihenfolge von Signalen

Die Verwaltung von Signalen über *Message-Descriptors* ermöglicht es einem Prozeß, auf Grundlage des *nonspecific receive*, gleichzeitig auf Nachrichten und Signale warten zu können. In diesem Zusammenhang bedeutet die Übermittlung von Signalen jedoch nicht nur die Möglichkeit zur nichtblockierenden Kommunikation zwischen sendenden und empfangenden Prozeß. Der Empfang von Signalen und Nachrichten wird beim *nonspecific receive* unterschiedlich gewichtet.

Signale unter MOOSE werden beim Empfang durch das *nonspecific receive* den Nachrichten gegenüber bevorzugt behandelt. Nachrichten untereinander werden immer nach der *first-come-first-serve* Strategie zum Empfang durch einen Prozeß verwaltet. Dies gilt unabhängig davon, ob das *specific-* oder *nonspecific receive* zum Empfang von Nachrichten verwendet wird.

Der Empfang von Signalen dagegen durchbricht diese Strategie. Signale werden dem empfangenden Prozeß immer vor den anstehenden Nachrichten zugestellt und demzufolge auch vor den Nachrichten empfangen. Hierzu muß jedoch der empfangende Prozeß mittels dem *nonspecific receive* seine Bereitschaft zu dieser Empfangsstrategie dem Kernel bekannt geben und dieser Prozeß muß mindestens einen *Event-Port*, zur Aufnahme von Signalen, besitzen.

3.4. Broadcasting

In einem prozeßorientierten Betriebssystem sind üblicherweise die typischen Funktionalitäten eines Betriebssystems in verschiedenen Systemprozessen angesiedelt. Diese Systemprozesse verwalten oftmals Informationen, die sie lokal bei sich für jedes *Team* oder für jeden Prozeß halten.

Diese Organisation ist typisch für MOOSE, sie muß jedoch nicht die allgemein gültige Vorgehensweise in einem prozeßorientierten System darstellen. In NUKE [Crowley 1981] z.B., werden die prozeßgebundenen Informationen in systemglobalen Strukturen verwaltet, die allen Systemprozessen auf Anforderung über *Shared-Memory* zugänglich gemacht werden. Dieses Verfahren erlaubt es, daß die Systemprozesse globale Zustandsänderungen der Prozesse (z.B. ihre Termination) anhand der zentral verwalteten Datenstrukturen direkt in Erfahrung bringen können. Der Nachteil ist jedoch, daß auf solch einer Basis die Realisierung eines dezentralen prozeßorientierten Betriebssystems nicht möglich ist. Letztendlich führt dieses Verfahren zu

[28] Dieses Verfahren wird in MOOSE jedoch nicht zur Anwendung gebracht, um die auf Prozeßebene dynamische Rekonfiguration eines Systems zu erzielen. Es wird später in diesem Zusammenhang ein allgemeineres Verfahren beschrieben.

einer globalen Abhängigkeit der Systemprozesse von den zentralen Datenstrukturen und widerspricht damit der strengen Forderung in MOOSE nach einem modularen und flexiblen Systementwurf.

Es ist eine Frage des Entwurfs eines prozeßorientierten Systems, wie umfangreich das gemeinsame "Wissen" der Systemprozesse über Zustände oder Aktivitäten aller Prozesse des Systems sein muß. Unabhängig von diesem Wissenspotential ist es wesentlich für die Systemprozesse, darüber informiert zu werden, wann ein Prozeß systemglobale Zustandsänderungen durchführt. Diese Information wird innerhalb des MOOSE-Systems durch einen *Broadcast* verbreitet. Systemprozesse, die einen *Broadcast* annehmen, können sich daraufhin die für sie notwendigen Prozeßinformationen verschaffen, indem sie dazu evtl. Dienste anderer Systemprozesse in Anspruch nehmen.

3.4.1. Verschiedene Klassen von Broadcasts

In einem Betriebssystem lassen sich verschiedene markante Zustandsänderungen für Prozesse identifizieren, die jeweils systemglobale Auswirkungen besitzen. Jede dieser Zustandsänderungen definiert eine eigene *Broadcast*-Klasse. Diese Klassen und ihre Bedeutung für das System sind im einzelnen:

termination class In diese Klasse fällt die Mitteilung über die Termination von Prozessen. Es ist oftmals wesentlich, einen Prozeß erst dann vollständig terminieren zu lassen (d.h., ihn aus dem Hauptspeicher zu entfernen), nachdem alle Systemprozesse die Möglichkeit besessen haben, bei sich *Cleanup*-Maßnahmen für den Prozeß durchführen zu können.

creation class In diese Klasse fällt die Mitteilung über die Erzeugung von Prozessen. Hiermit ist die Möglichkeit gegeben, bestimmte Familienbeziehungen zwischen Prozessen aufbauen zu können. Andererseits sind oftmals Strategien sinnvoll, bei denen bestimmte Objekte des erzeugenden Prozesses an den erzeugten Prozeß vererbt werden sollen.

attribute class In diese Klasse fällt die Mitteilung über Attributänderungen von Prozessen. Hierunter sind bestimmte Zugriffsattribute zu verstehen, die von den Systemprozessen in Anwendung gebracht werden, um Zugriffsrechte von Prozessen auf bestimmte Objekte überprüfen zu können.

execution class In diese Klasse fällt die Mitteilung über die Ausführung neuer Programme durch Prozesse. Die Erzeugung von Applikationen bedeutet, daß mindestens ein Programm durch einen bestimmten Prozeß zur Ausführung gebracht werden muß. Ebenso kann es notwendig sein, daß derselbe Prozeß zu verschiedenen Zeitpunkten unterschiedliche Programme ausführt, mit denen jeweils bestimmte Systemattribute verbunden sein können.

Die verschiedenen *Broadcast*-Klassen haben für den MOOSE-Kernel die Bedeutung, nur die Prozesse anhand des jeweiligen *Broadcasts* zu selektieren, die auch nur auf eine bestimmte Klasse von *Broadcasts* reagieren wollen. Entsprechend dazu werden die im PCB enthaltenen Attribute eines Prozesses so aufgesetzt, daß für den Kernel ersichtlich ist, für welchen *Broadcast* ein Prozeß empfänglich ist. Jedem Prozeß können somit bestimmte *Broadcast*-Attribute zugeordnet werden.

3.4.2. Selektierung von Broadcast-Servern

Ein *Broadcast* ist immer mit einem Prozeß, dem *Broadcast-Client*, verbunden. Der *Broadcast-Client* ist dafür verantwortlich, daß ein *Broadcast* gestartet, d.h., daß ein bestimmter Dienst systemglobal erbracht werden soll. Der *Broadcast-Client* ist solange blockiert, wie der *Broadcast* noch nicht vollständig bearbeitet worden ist. Die *Broadcasts* annehmenden Prozesse werden als *Broadcast-Server* bezeichnet.

Die Selektierung eines *Broadcast-Servers* richtet sich danach, welche Klasse von *Broadcasts* der *Broadcast-Client* behandelt haben will. Dazu ist jeder *Broadcast-Client* mit der Klasse des jeweiligen *Broadcasts* attributiert. Diese Attribute werden zurückgenommen, wenn der *Broadcast* abgeschlossen ist und bevor der *Broadcast-Client* reaktiviert wird. Die *Broadcast-Server* sind ebenfalls mit *Broadcast*-Klassen attributiert, und zwar mit den Klassen, die sie behandeln wollen. Die Entscheidung, welchem Prozeß ein *Broadcast* zugestellt werden soll, richtet sich somit einfach nach den Attributen des jeweiligen *Broadcast-Clients* und der potentiellen *Broadcast-Server*. Die Attribute müssen übereinstimmen. Um *Broadcast-Server* zu identifizieren, durchsucht der Kernel die *Process-Table* nach entsprechend attributierten Prozessen.

Ist ein *Broadcast-Server* identifiziert worden, so wird er jedoch erst dann tatsächlich selektiert, wenn sein *Team* allgemein über *Broadcasts* informiert werden soll. Ob dieses *Team* bei der Selektierung von *Broadcast-Servern* betrachtet werden soll, entscheidet sich anhand der Attribute des *Teams*.

Damit ergibt sich allgemein eine zweistufige Vorgehensweise bei der Selektierung von *Broadcast-Servern*. *Broadcast-Server* bekommen einen *Broadcast* nur dann zugestellt, wenn ihr jeweiliges *Team* die Annahme von *Broadcasts* global zuläßt. Innerhalb des *Teams* erhalten nur die Prozesse den *Broadcast*, die als *Server* ausgezeichnet und die der entsprechenden *Broadcast*-Klasse zugeordnet sind.

3.4.3. Protokoll zur Verarbeitung eines Broadcasts

Die Verarbeitung eines *Broadcasts* unterteilt sich in drei Protokollphasen. Diese Protokollphasen entsprechen den verschiedenen Phasen der Erbringung von Systemdiensten. Ein *Broadcast* repräsentiert letztendlich auch einen Systemdienst, der an mehrere Systemprozesse gerichtet ist.

3.4.3.1. Initiale Phase

Ein *Broadcast* wird durch die Übermittlung einer *Broadcast*-Nachricht gestartet. Dies kann explizit mit Hilfe von *send* geschehen, oder es erfolgt implizit durch Anwendung spezieller Mechanismen des Kernels. In jedem Fall ist der *Broadcast-Client* der Prozeβ, der die initiale Phase eines *Broadcasts* startet.

3.4.3.2. Intermediäre Phase

Die *Broadcast-Server* nehmen einen *Broadcast* mit Hilfe der üblichen Empfangsprimitive zur Interprozeβkommunikation, *receive*, an. Aufgrund des *Rendezvous*-Konzeptes erhalten sie damit gleichzeitig die Identifikation des jeweiligen *Broadcast-Clients*. Jeder *Broadcast-Server* hat dafür Sorge zu tragen, daβ der *Broadcast* zu einem anderen *Broadcast-Server* weitergereicht werden kann. Dazu dient das *nonspecific relay*. Beim *nonspecific relay* selektiert der Kernel, gemäβ seiner Strategie, einen bestimmten *Broadcast-Server*, dem dann die Nachricht zugestellt wird.

Ein *Broadcast* wird aufrechterhalten, indem die *Broadcast*-Nachricht "*relayed*" wird. Prinzipiell bedeutet dies die Verbreitung eines *Rendezvous* im System. Dies ist darauf zurückzuführen, daβ jeder *Broadcast-Server* einen *Broadcast* mittels *receive* in Empfang nimmt und daβ der jeweilige *Broadcast* mittels *relay* weitergereicht wird. Damit werden für den *Broadcast-Client* jeweils mehrere *Rendezvous* zu *Broadcast-Servern* eingerichtet.

3.4.3.3. Terminale Phase

Die Termination eines *Broadcasts* entspricht der Termination eines *Rendezvous* und wird demzufolge mittels *reply* erreicht. Aufgrund des *Rendezvous*-Konzeptes kann diese Termination von jedem Prozeβ herbeigeführt werden, der an dem *Rendezvous* beteiligt ist.

Die *Broadcast-Server* müssen Einvernehmen über ein gemeinsames *Broadcast*-Protokoll erzielen, das festlegt, welcher *Broadcast-Server* den *Broadcast* terminieren darf. Nur so kann sichergestellt werden, daβ der *Broadcast* im System vollständig verbreitet wird. Dieser ausgezeichnete Prozeβ, während der Abarbeitung eines *Broadcasts*, wird als *Broadcast-Master* bezeichnet.

3.4.4. Dynamisch rekonfigurierbare Broadcast-Systeme

Das übliche Vorgehen bei einem *Broadcast*-Protokoll im MOOSE-System würde vorsehen, daβ die initiale- und terminale Phase von dem *Broadcast-Master* kontrolliert wird. Nur die intermediäre Phase eines *Broadcasts* würde von allen *Broadcast-Servern* durchlaufen werden. Während dieser Phase wird ein *Broadcast* jeweils nur "*relayed*". Bild 3.8 skizziert eine Kette von solchen *Broadcast-Servern*.

Der *Broadcast-Master* wird denselben *Broadcast* wieder empfangen, wenn der *Broadcast* von allen *Broadcast-Servern* korrekt "*relayed*" worden ist. Die Selektionsstrategie des Kernels stellt dieses sicher, wenn sich die Attribute des *Broadcast-Masters* zwischenzeitlich nicht verändert haben. Dies gilt insbesondere auch dann, wenn insgesamt nur ein *Broadcast-Server* für den jeweiligen *Broadcast* im System vorhanden ist.

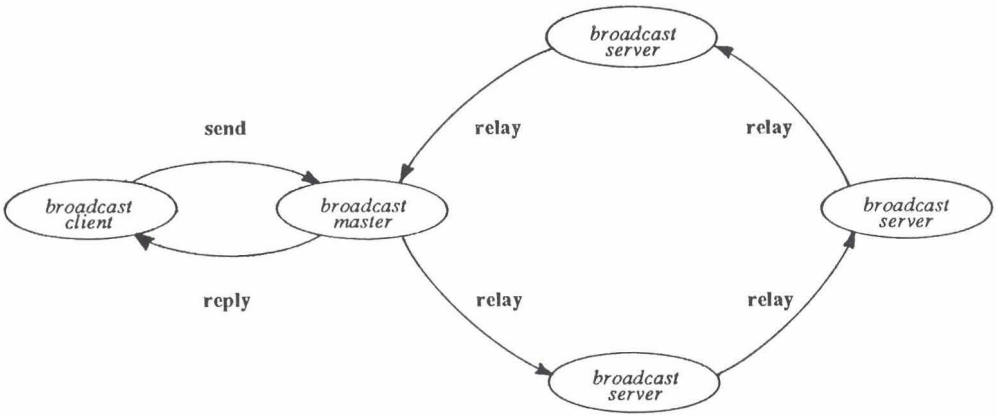

Bild 3.8: Beziehung zwischen Prozessen während eines *Broadcasts*

Gerade der Sonderfall, daß nur ein *Broadcast-Server* im System vorhanden ist, macht zwei wesentliche Aspekte im Zusammenhang mit der Abarbeitung eines *Broadcasts* auf Grundlage des *nonspecific relay* deutlich. Erstens können Prozesse Nachrichten zu sich selbst "*relayen*", und zweitens ist die Anzahl der an der Abarbeitung eines *Broadcasts* beteiligten Prozesse jedem *Broadcast-Server* selbst transparent. Damit ist eine wesentliche Grundlage gegeben, *Broadcast-Server* nachträglich in das System zu integrieren bzw. wieder aus dem System zu entfernen. Die dynamische Rekonfiguration eines Betriebssystems auf Prozeßebene wird hiermit wesentlich unterstützt. Dies ermöglicht es erst, auch Systemprozesse die oftmals *Broadcasts* annehmen müssen, als transient zu betrachten.

3.5. Adoption von Prozessen

Der Übermittlung von Nachrichten wird üblicherweise der Empfang der entsprechenden Nachrichten gegenüberstehen. Diese Interaktion ist typisch für *Message-Passing* und findet insbesondere immer im Zusammenhang mit der Abbildung von *System-Calls* auf entsprechende Nachrichten statt.

Der Nachrichtenaustausch zwischen Prozessen ist u.a. dann von wesentlicher Bedeutung, wenn beachtet wird, daß MOOSE auf Prozeßebene dynamisch rekonfigurierbare Systeme unterstützen soll. Dazu wäre mindestens der Schritt notwendig, das Laufzeitverhalten der auszutauschenden Prozesse nach außen hin, d.h. zu den dienstanfordernden Prozessen, nicht zu verändern. Ein anderer Prozeß muß somit in der Lage sein, mindestens die Nachrichten für den auszutauschenden Prozeß zu empfangen und evtl. auch zu bearbeiten: der auszutauschende Prozeß wird von einem anderen Prozeß adoptiert.

Interressant an dieser Stelle ist, daß der Kernel nur einen Mechanismus zur Verfügung stellen muß, der es erlaubt, daß Prozesse bestimmte Kommunikationspunkte "anzapfen" können. Diese Prozesse könnten somit, stellvertretend für die eigentlichen Empfängerprozesse,

die Nachrichten annehmen und daraufhin bestimmte Reaktionen veranlassen, die die Rekonfiguration eines Systems ermöglichen.

3.5.1. Monitore zur Überwachung von Kommunikationsaktivitäten

Abgesehen von der Unterstützung der dynamischen Rekonfiguration eines Systems, würde der Adoption von Prozessen durch entsprechende Mechanismen des Kernels ein breites Einsatzspektrum in einem prozeβorientierten Betriebssystem offen stehen. Solch ein Mechanismus würde nicht nur allgemein die Funktionalität eines bestimmten Systems erweitern. Er würde im speziellen ebenso helfen, bestimmte aus Effizienzgründen im Kernel nicht realisierte Kontrollmechanismen nachträglich über Prozesse in das System zu integrieren. Allgemein werden Prozesse, die solche Funktionalitäten wahrnehmen, in MOOSE als *Monitore* bezeichnet. Sie überwachen die Kommunikationsaktivitäten des Systems. Die Aufgaben der *Monitore* können sehr vielfältig sein und sind insbesondere für den Kernel vollkommen transparent.

3.5.1.1. Deblocking

Im Zusammenhang mit der Erläuterung des *Rendezvous*-Konzepts von MOOSE wurde dargestellt, daβ der Kernel bei der Interprozeβkommunikation keine Kontrolle durchführt, ob ein Empfängerprozeβ bereits terminiert ist. Dieser Fall wäre eine spezielle Ausnahmesituation im Zuge der Interprozeβkommunikation und ist zugunsten einer laufzeiteffizienten Realisierung der Kommunikationsmechanismen nicht betrachtet worden. Die Konsequenz davon ist jedoch, daβ die Kommunikation mit einem terminierten Prozeβ zur unbefristeten Blockierung des sendenden Prozesses führen würde.

Die Behandlung dieser Ausnahmesituation kann mit Hilfe eines entsprechenden *Monitors* erfolgen. Dieser *Monitor* fungiert als *Garbage-Collector* für die Kommunikation zu bereits terminierten Prozessen. Mit der Termination eines Prozesses, die dem *Monitor* über einen entsprechenden *Broadcast* mitgeteilt werden kann, blendet sich der *Monitor* für den terminierten Prozeβ zur Annahme von Nachrichten ein.

Die Kommunikation mit solch einem *Garbage-Collector* würde somit immer bedeuten, daβ der jeweils sendende Prozeβ mit einem bereits terminierten Prozeβ kommunizieren will. Der *Monitor* kann daraufhin eine applikationsspezifische Behandlung dieser Ausnahmesituation einleiten. Es kann bedeuten, den sendenden Prozeβ ebenfalls zu terminieren. Andere Möglichkeiten beständen darin, den sendenden Prozeβ davon zu informieren, daβ sein Kommunikationspartner bereits terminiert ist.

Dieses Verfahren ermöglicht nicht nur die effiziente Realisierung der Mechanismen zur Interprozeβkommunikation. Es erlaubt ebenso, Behandlungsstrategien für solche Ausnahmesituationen an die jeweiligen Applikationen anzupassen.

3.5.1.2. Swapping

Komplexe Funktionalitäten eines *Monitors*, die mit denen der Aufnahme von Kommunikationsaktivitäten zu bereits terminierten Prozessen vergleichbar sind, ergeben sich z.B. im Zusammenhang mit der Realsierung eines *Swappers*. Auch hier könnte eine Kommunikation mit einem Prozeβ gewollt sein, der jedoch "*geswapped*" ist. Dieser Kommunikationswunsch eines anderen Prozesses kann von einem *Swapper* auf zwei verschiedene Weisen ausgewertet werden. Einerseits kann der kommunizierende Prozeβ ebenfalls ausgelagert werden, da er in jedem Fall blockieren würde. Andererseits kann dies die Dringlichkeit erhöhen, den Prozeβ, mit dem kommuniziert werden soll, wieder in den Hauptspeicher einzulagern.

Um solche Funktionalitäten in einem prozeβorientierten Betriebssystem auf Grundlage von *Message-Passing* realisieren zu können, braucht in dem *Swapper-Team* nur ein *Monitor* angesiedelt zu sein. Der *Monitor* nimmt repräsentativ für alle ausgelagerten Prozesse die Kommunikationswünsche an und initiiert daraufhin eine entsprechende Behandlungsmaβnahme.

Die Behandlungsmaβnahmen des *Swappers* müssen u.a. dafür sorgen, daβ die zurückgestellten Kommunikationswünsche an die jeweils ausgelagerten Prozesse neu zu betrachten sind. Das bedeutet, daβ nach der Einlagerung eines Prozesses, alle zu diesem Prozeβ vermerkten Kommunikationswünsche ihm zugestellt werden müssen. Dies erreicht der *Swapper* in elementarer Weise durch das *specific relay* der Nachrichten, die die jeweiligen Kommunikationswünsche repräsentieren.

3.5.1.3. Monitoring

Andere Funktionalitäten solcher *Monitore* können in dem *monitoren* schlechthin gegeben sein. So könnte die Aufgabe der *Monitore* nur darin bestehen, die Systemaktivitäten bestimmter oder aller Prozesse aufzuzeichnen. Damit würde *Accounting* von Systemaktivitäten unterstützt werden können. Andererseits ist es, z.B. während der *debug*-Phase eines Systemprozesses, sinnvoll, die Kommunikationsaktivitäten zu bestimmten Prozessen nur aufzuzeichnen, um das Schnittstellenverhalten zu den Prozessen analysieren zu können.

3.5.2. Manipulation von Kommunikationsbeziehungen

Der Ansatzpunkt eines *Monitors* ergibt sich unter MOOSE immer im Zusammenhang mit einem *Rendezvous-Server*. Um die Kommunikationsaktivitäten somit *monitoren* zu können, muβ die Abbildung der *process ID* des Empfängerprozesses manipuliert werden können. Diese Manipulation muβ sich dadurch auszeichnen, daβ ein *Monitor* alle Nachrichten, die zu dem betreffenden Prozeβ übermittelt werden, anstelle dieses Prozesses annehmen kann.

Hierzu ist im MOOSE-Kernel ein sehr effizientes Verfahren realisiert, das keine zusätzliche Laufzeit für die jeweiligen Prozesse mit sich bringt. Dieses Verfahren orientiert sich nach dem Prinzip der *Process-Map*. Die *Process-Map* enthält die Referenzen auf PCBs, die wiederum die Prozesse aus Sicht des Kernels eindeutig beschreiben. Entsprechend dieser Tabelle wird zur Realisierung der *Monitor*-Funktionalität eine sogenannte *Receive-Map* eingeführt. Diese Tabelle entspricht vollständig der *Process-Map*, mit der Ausnahme, daβ ihre Einträge

nachträglich modifiziert werden können.

Für die Manipulation der *Receive-Map* stellt der MOOSE-Kernel einen entsprechenden Mechanismus zur Verfügung. Dieser Mechanismus erlaubt es, einen Eintrag in der *Receive-Map*, identifiziert durch die *process ID* des Prozesses der adoptiert werden soll, auf einen anderen PCB zu setzen. Dieser PCB wird der *Process-Map* entnommen. Bild 3.9 zeigt einen Schnappschuß der möglichen Zuordnung zwischen *Receive-Map* und *Process-Table* wie auch zwischen *Process-Map* und *Process-Table*.

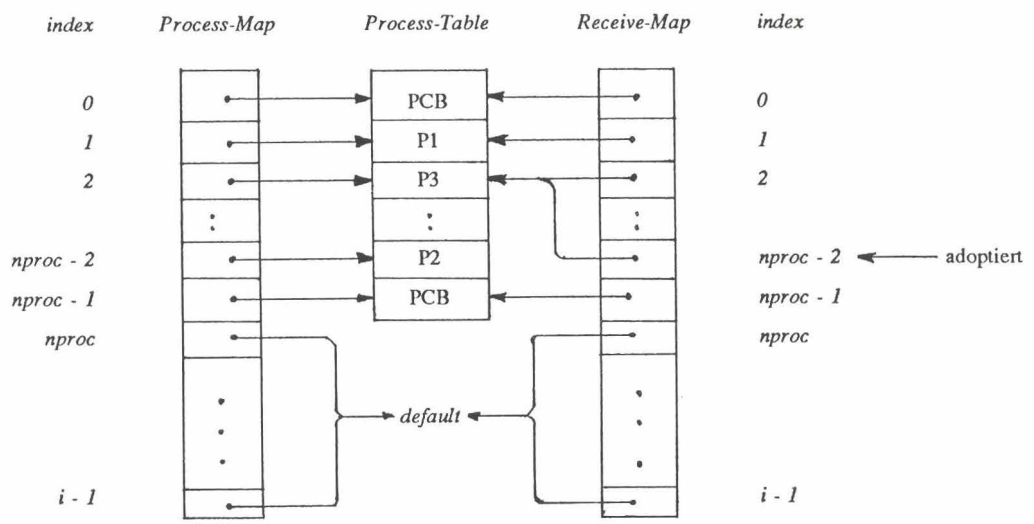

Bild 3.9: Manipulation der *Receive-Map*

Die *Receive-Map* wird von allen Primitiven des MOOSE-Kernels in Anspruch genommen, die zur Übermittlung von Nachrichten beitragen. Dies sind die Primitiven *send* und *relay*. Da die Einträge der *Receive-Map* nicht notwendigerweise die Prozesse identifizieren, die im Zusammenhang mit *send* oder *relay* als Empfangsprozesse angegeben worden sind, findet eine Umleitung der betreffenden Nachricht zu einem anderen Prozeß, dem *Monitor*, statt. Logisch stellt sich diese Situation entsprechend Abbildung 3.10 dar.

Zum Vergleich der kernelinternen mit der logischen Sichtweise eines adoptierten Prozesses, sind in den beiden Abbildungen die entsprechenden Prozeßobjekte mit identischen Bezeichnungen versehen worden. Die ursprüngliche Beziehung ist direkt zwischen P1 und P2 gegeben. Die *Receive-Map* wäre in diesem Beispiel identisch mit der *Process-Map*. Die Adoption von P2 durch P3 führt kernelintern dazu, daß der Eintrag in der *Receive-Map* für P2 auf den PCB von P3 gesetzt wird. Damit wird jedes *send* zu P2 direkt auf P3 abgebildet. Diese Abbildung ist für das *send* vollkommen transparent, da zur Identifikation der *Rendezvous-Server* immer die *Receive-Map* eingesetzt wird.

Der *Monitor* P3 leitet in dem skizzierten Beispiel die von P1 empfangende Nachricht zu P2 weiter; die Nachricht erreicht damit ihren ursprünglichen Bestimmungsort. Die Beendigung des *Rendezvous* zu P1 wird durch P2 durchgeführt, womit P2 die indirekte (über P3 erfolgte) Kommunikation mit P1 abschließt.

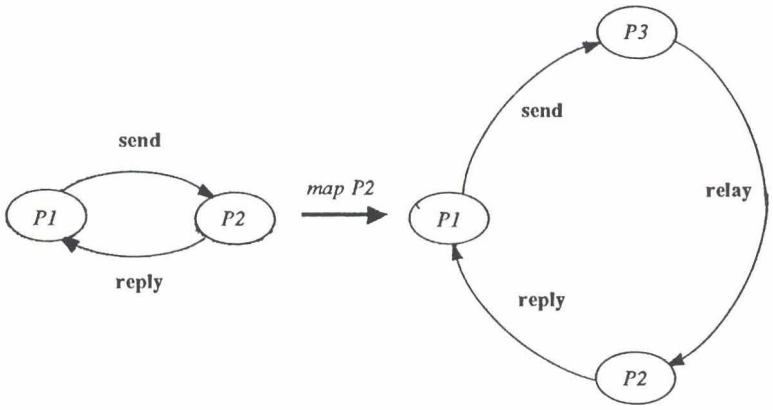

Bild 3.10: Adoption eines Prozesses

3.6. Traps und Interrupts

Eine präzise Definition der Semantik eines Traps bzw. Interrupts ist der einschlägigen Literatur für den Betriebssystembereich nicht zu entnehmen [29]. Dies ist um so mehr bemerkenswerter, als daß die Begriffe "Trap" und "Interrupt" gerade im Betriebssystembereich ihre zentrale Bedeutung besitzen. Die Mechanismen zur Behandlung von Traps und Interrupts basieren vielmehr auf einem bestimmten intuitiven Verständnis über die Semantik eines Traps bzw. Interrupts.

Obgleich die Begriffe "Trap" und "Interrupt" intuitiv verständlich sind, ist deren präzise Definition nicht trivial. Solch eine Definition kann nur erfolgen, wenn bestimmte abstrakte und formale Modelle zur Beschreibung eines Software-Systems zugrunde gelegt werden. In dieser Hinsicht stellt [Schindler 1983] einen Ansatz dar, insbesondere auf Grundlage von [Liskov 1972] und [Parnas 1976], die Semantik der Begriffe "Trap" und "Interrupt" im Betriebssystembereich präzise zu definieren.

3.6.1. Abstraktionsebenen und Prozessoren

Um die Behandlung von Traps und Interrupts unter MOOSE zu verdeutlichen, werden hierzu im wesentlichen vier Abstraktionsebenen betrachtet. Diese Unterteilung stellt dar, welche Ebenen unter MOOSE im Zusammenhang mit der Behandlung von Traps und Interrupts von Bedeutung sind. Es wird damit jedoch weder die Benutztstruktur [Parnas 1976] noch die

[29] Konkret wird hierbei auf [Coffman, Denning 1973], [Tsichritzis, Bernstein 1974], [Shaw 1974], [Habermann 1976] und [Deitel 1984] Bezug genommen. In [Deitel 1984] erfolgt insbesondere eine Klassifikation von Interrupts, die konträr zu der in den anderen Arbeiten und zu der in MOOSE ist. In [Tanenbaum 1984] findet zwar eine sehr anschauliche Erläuterung von Traps und Interrupts statt, die präzise Bedeutung ihrer Behandlung in einem hierarchisch strukturierten Betriebssystem bleibt jedoch ebenfalls offen.

konkrete hierarchische Struktur [Parnas 1974] eines MOOSE-Betriebssystems vorgestellt.

Die einzelnen Abstraktionsebenen stehen in einer eindeutigen hierarchischen Beziehung zueinander. Die unterste Ebene wird hierbei durch einen realen Prozessor und die hierarchisch höher liegenden Ebenen werden jeweils durch einen abstrakten Prozessor repräsentiert. Bild 3.11 skizziert dieses Modell.

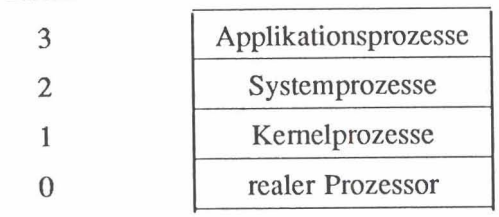

Bild 3.11: Hierarchie von Abstraktionsebenen

Wie in [Schindler 1983] dargestellt, bewirkt jeder dieser Prozessoren die Abarbeitung von Programmen und damit die Ausführung von Prozessen. Die einzelnen Programme der Ebene $n+1$ sind aus Elementaroperationen zusammengestellt, die von einem Prozessor der Ebene n interpretiert werden können – der Prozessor der Ebene n implementiert eine bestimmte Menge von Elementaroperationen. Solch eine Interpretation von Elementaroperationen ermöglicht somit die Abarbeitung der aus diesen Elementaroperationen zusammengesetzten Progamme.

Es ist eine Frage des jeweils betrachteten Abstraktionsniveaus [Schindler 1983], ob die Interpretation von Elementaroperationen durch einen abstrakten oder einen realen Prozessor stattfindet. Die Vorgehensweise bei der Interpretation von Elementaroperationen ist jedoch für beide Kategorien von Prozessoren identisch. Dies gilt insbesondere im Zusammenhang mit der Behandlung von Traps und Interrupts durch einen spezifischen Prozessor.

3.6.1.1. Trap-Behandlung in hierarchischen Systemen

Nach [Schindler 1983] bedeutet ein Trap, daß bei der Interpretation einer Elementaroperation durch den Prozessor der Ebene n (0, 1, 2) eine für die jeweilige Elementaroperation geltende Ausnahmesituation aufgetreten ist. Der Trap signalisiert die durch einen Prozessor festgestellte Ausnahmesituation. Solche Ausnahmesituationen für die einzelnen Elementaroperationen der Ebene n werden jeweils durch den Prozessor der Ebene n definiert.

Der erste Schritt im Zuge der Behandlung eines Traps besteht prinzipiell darin, daß der Prozessor der Ebene n eine entsprechende Behandlungsroutine auf der Ebene n aktiviert. Diese Behandlungsroutine kann selbst als die Implementierung einer bestimmten

Elementaroperation der Ebene n betrachtet werden. Die Aktivierung dieser Routine aufgrund eines Traps entspricht somit dem impliziten Aufruf einer Elementaroperation von der Ebene $n+1$ aus.

Die Erkennung und Behandlung einer Ausnahmesituation auf der Ebene n wird jedoch im allgemeinen der Behebung der durch einen Trap signalisierten Ausnahmesituation im Betriebssystembereich nicht Rechnung tragen können. Vielmehr wird es notwendig sein, die Behandlung der Ausnahmesituation hierarchisch höher angeordneten Ebenen zu übertragen. Die Behandlung der Ausnahmesituation auf der Ebene n bestände somit darin, die jeweilige Ausnahmesituation an die Ebene $n+1$ weiterzureichen. Das Weiterreichen des Ereignisses eines Traps an höher angeordnete Ebenen eines hierarchisch strukturierten Systems, ist eng im Zusammenhang mit dem klassischen Verständnis über *Exception Handling* [Goodenough 1975] zu sehen. In diesem Fall wird von dem Propagieren einer Ausnahmesituation (dem Trap) gesprochen.

Die Propagation eines Traps ist typisch für einen realen Prozessor, d.h. für einen Prozessor auf der physikalischen Ebene. Der reale Prozessor erkennt Traps, kann deren Behandlung jedoch nicht vollständig durchführen. Ein *Page-Fault* Trap, z.B., kann nach Erkennung durch den realen Prozessor nur durch einen abstrakten (hierarchisch höher angeordneten) Prozessor behandelt werden. Die Behandlung eines solchen Traps muß den Kontext des den Trap auslösenden Prozesses betrachten und demzufolge auf bestimmte betriebssysteminterne Informationen zurückgreifen. Diese Informationen sind jedoch nur bestimmten abstrakten Prozessoren bekannt. Um demzufolge die Behandlung eines Traps im Kontext des den Trap verursachenden Prozesses durchführen zu können, muß der abstrakte Prozessor den Trap durch einen hierarchisch tieferliegenden Prozessor, in letzter Konsequenz durch den realen Prozessor, signalisiert bekommen.

3.6.1.2. Interrupt-Behandlung in hierarchischen Systemen

Die im Zusammenhang mit der Behandlung von Traps skizzierte Betrachtungsweise gilt prinzipiell auch für die Behandlung von Interrupts. Nach der Feststellung durch einen bestimmten Prozessor der Ebene n, daß ein Interrupt zur Behandlung ansteht, aktiviert dieser Prozessor eine entsprechende Behandlungsroutine der Ebene n. Diese Behandlungsroutine bestimmt, ob der Interrupt zu einem hierarchisch höher angeordneten Prozessor zu propagieren ist.

Allgemein führen Traps und Interrupts somit zu einer Unterbrechung der durch einen Prozessor der Ebene n ermöglichten normalen Abarbeitung eines Programmes der Ebene $n+1$. Dennoch besteht ein wesentlicher Unterschied zwischen Traps und Interrupts.

Im Gegensatz zu Traps, wird die Behandlung eines Interrupts nicht im Zusammenhang mit der Interpretation von Elementaroperationen stehen. Traps, und damit die Interpretation von Elementaroperationen durch einen Prozessor, stehen immer im Zusammenhang mit der Ausführung eines bestimmten Prozesses. Interrupts dagegen sind auf externe Ereignisse zurückzuführen, die asynchron zur Ausführung des gegenwärtig aktiven Prozesses auftreten und die Aufmerksamkeit eines bestimmten Prozessors für sich erfordern.

Die Unterbrechung der Interpretation einer Elementaroperation durch einen Interrupt ist nicht darauf zurückzuführen, daß für die jeweilige Elementaroperation eine bestimmte Ausnahmesituation vorgegeben ist. Vielmehr sind bestimmte externe Ereignisse für die Unterbrechung verantwortlich. Die Tatsache, daß Interrupts aufgrund externer Ereignisse auftreten, erschwert in besonderen Maße deren Behandlung durch einen bestimmten Prozessor. Mehr noch, die Koordination der im Zusammenhang mit der Behandlung von Interrupts notwendigen Systemaktivitäten, stellt, aufgrund des Asynchronismus' der Interrupts, ein zentrales Problem bei dem Entwurf und der Realisierung von Betriebssystemen dar.

Wie im Zusammenhang mit Traps, wird die Propagation der Interrupts von der Ebene n zur Ebene $n+1$ den Normalfall darstellen. Hiermit wird dem Betriebssystem die Möglichkeit gegeben, auf Interrupts in adäquater Weise reagieren zu können. Dies bedeutet z.B., daß Ein-/Ausgabeaufträge abgeschlossen und die auf die Fertigstellung dieser Aufträge wartenden Prozesse reaktiviert werden können.

3.6.2. Kernel-Calls

Üblicherweise sind die Aktivitäten der Benutzerprozesse von denen des Betriebssystems konkret entkoppelt. Dies wird durch unterschiedliche Adreßräume für die jeweils betrachteten Komponenten erreicht. Für diese Organisation bestehen im wesentlichen zwei Gründe. Einerseits stellt ein Betriebssystem bestimmte Dienste zur Verfügung, die von mehreren Komponenten oberhalb des Betriebssystems angewendet werden können, um ihrerseits bestimmte Funktionalitäten zu erbringen. In diesem Sinne besteht ein Betriebssystem aus einer bestimmten Anzahl von *Moduln*, die für den Großteil der sie benutzenden Komponenten dieselben Schnittstellen anbieten.

Der andere wesentliche Grund für die Entkopplung (der einzelnen Moduln) des Betriebssystems von den Aktivitäten der Benutzerprozesse, ist durch die an ein Betriebssystem gestellten hohen Sicherheitsanforderungen gegeben. Die Zuordnung eines eigenen Adreßraumes für die Moduln eines Betriebssystems verhindert den direkten Zugriff auf betriebssysteminterne Informationen.

Die physikalische Einkapselung eines Betriebssystems ist jedoch nur durch entsprechende Eigenschaften der zugrunde liegenden Hardware möglich. Die logische Entkopplung ist jedoch immer gegeben und wird durch die jeweilige Systemstruktur definiert.

Um aufgrund dieses Hintergrundes Systemdienste in Anspruch nehmen zu können, muß ein (zumindest konzeptioneller) Adreßraumwechsel von der Benutzerebene zur Systemebene erfolgen. Die jeweiligen Systemdienste werden über *System-Calls* abgesetzt. Einen elementaren Mechanismus für solch einen Adreßraumwechsel im Zusammenhang mit *System-Calls* stellen die Trap-Instruktionen eines realen Prozessors dar.

Da Trap-Instruktionen typisch für die technische Realisierung von *System-Calls* sind, wird oftmals mit den *System-Calls* der Trap (bei [Shaw 1974] und [Habermann 1976]) oder gar der Interrupt (bei [Deitel 1984]) selbst in Verbindung gebracht. Diese Betrachtungsweise ist jedoch nicht zutreffend. Sie erweist sich insbesondere durch Systeme wie MULTICS [Organick 1972] und DAS [Isle et al. 1977] als unhaltbar. Der Trap wie auch *capabilities* stellen jeweils

technische Hilfsmittel dar, mittels derer im Zusammenhang mit *System-Calls* ein Adreβraumwechsel vollzogen werden kann. Der Trap ist in diesem Fall nicht aufgrund einer Ausnahmesituation hervorgerufen worden. Dies gilt ebenso für den einen *System-Call* absetzenden und damit einen Trap auslösenden Prozeβ, wie auch für den den Trap behandelnden Prozessor.

Unter MOOSE entspricht der *Kernel-Call* der Sichtweise eines *System-Calls* in herkömmlichen prozedurorientierten Systemen wie UNIX. *Kernel-Calls* dienen dazu, bestimmte Dienste des Kernels den Prozessen zur Anwendung zur Verfügung zu stellen. Hierbei handelt es sich im wesentlichen um die Mechanismen zur Interprozeβkommunikation und zur Annahme von Traps bzw. Interrupts durch spezielle *Server*-Prozesse.

Die *Kernel-Calls* unterscheiden sich jedoch erheblich in ihrer Funktionalität von den üblicherweise im Zusammenhang mit der Erbringung von Systemdiensten genannten *System-Calls*. In MOOSE werden *System-Calls* auf Nachrichten abgebildet und mit Hilfe von *Kernel-Calls* den entsprechenden Systemprozessen zugestellt. Die Aufgabe des Kernels ist es hierbei, die Verbindung zwischen dem dienstanfordernden und diensterbringenden Prozeβ zu ermöglichen.

3.6.3. Clock-Interrupts

Der MOOSE-Kernel behandelt *Clock-Interrupts*, um zwei Funktionalitäten erbringen zu können. Hierbei handelt es sich einerseits um die Verwaltung von Zeitscheiben für *Teams* und andererseits um die Realisierng einer *Realtime-Clock*. Obgleich die Entwurfsentscheidung so ausgelegt ist, *Clock-Interrupts* unter MOOSE auf der Kernelebene zu behandeln, sind *Clock-Interrupts* keine notwendige Voraussetzung, um ein funktionsfähiges MOOSE-System auf der Kernelebene zu ermöglichen. Bestimmte Randbedingungen, um die Systembelastung durch *Clock-Interrupts* zu minimieren, sind für diese Entscheidung verantwortlich.

Auch ohne *Clock-Interrupts* wird die Ausführung von Prozessen möglich sein, zumindestens dann, wenn die Prozesse *System-Calls* bzw. *Kernel-Calls* absetzen. Für die *Realtime-Clock* würde das Fehlen von *Clock-Interrupts* bedeuten, keine Zeitabläufe feststellen zu können. Die mit dem *Clock-Interrupt* in Zusammenhang stehenden Mechanismen des MOOSE-Kernels, sind aus diesem Grunde konsequent von den anderen Mechanismen des Kernels getrennt.

Obgleich der MOOSE-Kernel die *Clock-Interrupts* selbst behandelt, ist dennoch ihre Propagation zu hierarchisch höher angeordneten Ebenen möglich. Dies würde z.B. die Aktivierung eines *Schedulers* ermöglichen, der im Zuge der Behandlung eines *Clock-Interrupts* einen neuen *Dipatch-Set* aufbauen kann. Ebenso kann ein *Clock-Server* dadurch auf *Clock-Interrupts* reagieren.

Diese Sichtweise gilt allgemein für die Behandlung von Traps und Interrupts auf den verschiedenen skizzierten Abstraktionsebenen. Die Behandlung dieser Ereignisse durch einen Prozessor der Ebene n bedeutet somit nicht, daβ eine Behandlung der entsprechenden Ereignisse auf der Ebene $n+1$, d.h. deren Propagation, unterbunden wird.

3.7. Server-Prozesse zur Behandlung von Traps und Interrupts

Der MOOSE-Kernel ist direkt über der physikalischen Ebene der dargestellten vier Abstraktionsebenen angesiedelt. Zur Behandlung von Traps und Interrupts wird hierbei erwartet, daß der direkt untergelagerte reale Prozessor die aufgetretenen Traps bzw. Interrupts zum Kernel propagiert.

Der Normalfall der Trap- und Interrupt-Behandlung durch den MOOSE-Kernel besteht darin, daß er seinerseits die jeweiligen Traps bzw. Interrupts propagiert. In diesem Fall nehmen *Server*-Prozesse die entsprechenden Ereignisse "Trap" und "Interrupt" an und führen ihrerseits eine bestimmte Behandlung dieser Ereignisse durch. Die *Server*-Prozesse bilden in ihrer Gesamtheit die Systemebene eines MOOSE-Systems.

Ob die von den *Server*-Prozessen angenommenen Traps und Interrupts zu den Benutzerprozessen und damit zu der Benutzerebene propagiert werden, ist eine Frage des jeweiligen Modells zur Behandlung von Traps/Interrupts durch die entsprechenden *Server*-Prozesse. Auf der Systemebene, und nicht auf der Kernelebene, wird somit die für die Benutzerprozesse zur Verfügung stehenden Funktionalitäten zur applikationsspezifischen Behandlung von Traps und Interrupts festgelegt.

Die Propagation der Traps und Interrupts durch den MOOSE-Kernel ist jedoch erst dann möglich, wenn dem Kernel die entsprechenden *Server*-Prozesse zur Behandlung der Traps bzw. Interrupts bekannt sind. Der MOOSE-Kernel bietet dazu Mechanismen an, die es *Server*-Prozessen gestattet, sich zur Behandlung von Traps und Interrupts bereitzustellen, indem für diese Prozesse die Anbindung an die entsprechenden Trap-/Interrupt-Vektoren des realen Prozessors ermöglicht wird. Diese Mechanismen erlauben es insbesondere, daß die *Server*-Prozesse zur Behandlung von Trap/Interrupts zur Laufzeit eines MOOSE-Betriebssystems ausgetauscht werden können. Desweiteren ist es den jeweiligen *Server*-Prozessen möglich, verschiedene Traps/Interrupts "gleichzeitig" anzunehmen und zu behandeln.

Sollte keine Propagation der Traps bzw. Interrupts durch den MOOSE-Kernel möglich sein, da kein *Server*-Prozeß für die Behandlung zur Verfügung steht, so stellt der Kernel seine weitere Ausführung ein. Im Sinne eines abstrakten Prozessors nach [Schindler 1983] bedeutet dies, daß der abstrakte Prozessor "Kernel" keine weiteren Elementaroperationen mehr interpretiert. Diese Elementaroperationen stellen die einzelnen *Kernel-Calls* wie auch der *Clock-Interrupt* dar. Aufgrund der Einordnung des Kernels in das skizzierte Modell der Hierarchie von den vier Abstraktionsebenen zur Behandlung von Traps und Interrupts, wird in dieser Situation letztendlich keine weitere Systemaktivität mehr stattfinden[30].

3.7.1. Trap-Messages und Interrupt-Signals

Für alle vom MOOSE-Kernel propagierten Traps und Interrupts sieht es das MOOSE-Konzept vor, *Server*-Prozesse zur Behandlung einzusetzen. Der Kernel ermöglicht es in diesem Zusammenhang, die *Server*-Prozesse von dem Ereignis "Trap" bzw. "Interrupt" zu informieren.

[30] Wie in verschiedenen anderen Systemen auch, z.B. UNIX, so signalisiert der MOOSE-Kernel in diesem Fall ein *Panic*.

Hierbei werden aufgrund der unterschiedlichen Charakteristiken von Traps und Interrupts jeweils verschiedenen Formen der Benachrichtigung eingesetzt.

Traps werden über besondere *Trap-Messages*, durch Anwendung der üblichen Mechanismen zur Interprozeβkommunikation, dem jeweiligen *Trap-Server* mitgeteilt. Die *Trap-Message* enthält als wesentliche Information die Registerinhalte des realen Prozessors. Diese Registerinhalte sind Teil des Prozessorzustandes des einen Trap auslösenden Prozesses.

Interrupts werden durch *Interrupt-Signals* den entsprechenden *Interrupt-Servern* zugesandt. Hierzu wird eine besondere Variante des Signalisierens in Anwendung gebracht, die den Synchronismus der Signalübermittlung sicherstellt.

3.7.2. Dynamische Rekonfiguration von Trap-/Interrupt-Servern

Die Zuordnung von *Server*-Prozessen zu Trap-/Interrupt-Vektoren stellt eine wesentliche Funktionalität des MOOSE-Kernels dar. Diese Funktionalität ermöglicht es, auf Prozeβebene dynamisch rekonfigurierbare Betriebssysteme aufbauen zu können. Die Prozesse sind hierbei der Systemebene eines MOOSE-Betriebssystems zuzuordnen.

Dadurch daβ die Zuordnung von *Server*-Prozessen zu Trap-/Interrupt-Vektoren des realen Prozessors dynamisch ist, kann zur Laufzeit die Funktionalität eines MOOSE-Betriebssystems nachträglich geändert werden. Dies ist um so mehr von Bedeutung, als daβ die Behandlung von Traps und Interrupts typischerweise in Betriebssystemen eine zentrale Rolle einnimmt und aus diesem Grunde oftmals im Kernel angesiedelt ist.

Die Rekonfiguration eines Systems auf der Ebene der Behandlung von Traps/Interrupts ist in den gegenwärtig verbreiteten Betriebssystemen zur Laufzeit nicht möglich. Vielmehr führt in dieser Situation die Integration neuer Komponenten zur Trap- bzw. Interrupt-Behandlung immer zur Generierung eines neuen Systems. Diese Vorgehensweise erschwert im erheblichen Maβe das Austesten neuer Systemkomponenten eines Betriebssystems.

Es wird jedoch sicherlich nicht bedeuten, daβ das *Server*-Modell von MOOSE eine für die im System befindlichen Prozesse vollkommen transparente Rekonfiguration eines MOOSE-Betriebssystems ermöglichen. Gerade das Austesten von Gerätetreibern und damit eines Interrupt-Systems, kann bei auftretenden Fehlern verheerende Auswirkungen auf das Gesamtsystem haben. Dies gilt für alle Klassen von Betriebssystemen, so auch für ein Mitglied der MOOSE-Betriebssystemfamilie. Das *Server*-Modell von MOOSE bietet jedoch die Möglichkeit, Interrupt-Systeme weitestgehend autonom testen zu können.

Die Möglichkeit der konsequenten Anwendung von Prozessen zur Modellierung von Interrupt-Systemen, auf Basis der vom MOOSE-Kernel zur Verfügung gestellten Funktionalitäten, motiviert explizit einen bestimmten Systementwurf. Insbesondere im Zusammenhang mit der Behandlung von Interrupts zahlt sich das Prinzip aus, *Interrupt-Handler* als sehr elementare Bindeglieder zwischen Geräten und *Interrupt-Servern* zu betrachten und den *Interrupt-Servern* ihrerseits die Verantwortung zur Geräteverwaltung und -kontrolle zu übertragen. Dieses Prinzip ermöglicht es, mit einfachen Hilfsmitteln eine Umgebung für den *Interrupt-Server* einzurichten, die das Verhalten von Geräten simuliert. Das Austesten neuer Systemkomponenten ist mit diesem Ansatz fast vollständig möglich und

kann zur Laufzeit – und transparent für alle anderen Aktivitäten – des Systems erfolgen.

3.7.3. Security Kernel

Nicht allein der Umstand der Neugenerierung eines Betriebssystems wird unter MOOSE mit Hilfe von *Trap-* und *Interrupt-Servern* vermieden. Die Neugenerierung stellt dann kein signifikantes Problem mehr dar, wenn auf ein ausgetestetes und für eine konkrete Umgebung bereits konfiguriertes Betriebssystem Bezug genommen wird. Vielmehr bedeutet gerade die durch die *Server*-Prozesse ermöglichte Einkapselung der Behandlung von Traps und Interrupts einen wesentlichen Beitrag zur Sicherheit eines Betriebssystems.

Die Verifikation eines MOOSE-Betriebssystems wird durch das *Server*-Modell zur Behandlung von Traps und Interrupts erheblich unterstützt. Einerseits ist hierfür die bereits genannte Abkapselung der Komponenten zur Trap-/Interrupt-Behandlung von den restlichen Aktivitäten eines MOOSE-Betriebssystems von Bedeutung. Andererseits erleichtert das Prinzip von sequentiell ablaufenden Prozessen die Vermittlung des Gesamtverständnisses über einen bestimmten Betriebssystemkomplex, z.B. den Komplex zur Geräteverwaltung und Interrupt-Behandlung. Die Systemprozesse können weitestgehend unabhängig voneinander formuliert und der Kernel in seiner Komplexität erheblich verkleinert werden.

Mit dieser Organisation sind – mit Ausnahme der Behandlung von *Clock-Interrupts* – keinerlei geräteabhängigen Operationen im MOOSE-Kernel angesiedelt. Dieser Ansatz unterstützt in konsequenter Weise das Konzept eines *Security Kernels* (für einen Überblick über die Konzepte und Entwurfsprinzipien solch eines Kernels sei auf [Deitel 1984] verwiesen). Der MOOSE-Kernel läßt sich dadurch sehr kompakt und übersichtlich gestalten, womit auf Grundlage der heutigen Verifikationstechniken eine notwendige Voraussetzung für die Darstellung der Korrektheit des Kernels gegeben ist.

3.7.4. Emulation von Systemfunktionen

Eine besondere Bedeutung erlangt das *Server*-Modell von MOOSE im Zusammenhang mit der Emulation von Systemfunktionen. Hierbei stellen die Trap-Instruktionen des realen Prozessors oftmals den geeigneten Mechanismus zur Verfügung, Systemdienste von den Benutzerprozessen in Anspruch nehmen zu lassen. Bei der Interpretation solcher Trap-Instruktionen durch den realen Prozessor, wird dieser einen Trap zum MOOSE-Kernel propagieren. Da *Server*-Prozesse solche Traps annehmen und behandeln können, ist in elementarer Weise die Möglichkeit zur Integration von Emulatoren unter MOOSE gegeben.

Die Emulatoren stellen ausgezeichnete *Trap-Server* unter MOOSE dar. Ihre Aufgabe besteht darin, Bindeglieder zwischen unterschiedlichen Betriebssystemdomänen, z.B. UNIX und AX[31], zu repräsentieren. Die zu emulierenden und durch Annahme einer *Trap-Message* mitgeteilten Systemdienste, werden von den Emulatoren auf Systemdienste eines bestimmten Referenzsystems abgebildet.

[31] AX steht für *Advanced UNIX* und repäsentiert den ersten Prototyp eines MOOSE-Betriebssystems.

Die Integration von Emulatoren kann dynamisch gestaltet werden. Emulatoren brauchen erst dann im System zur Verfügung zu stehen, wenn die erste Applikation der durch den Emulator definierten Betriebssystemdomäne gestartet werden soll. Insbesondere kann die Anzahl solcher Emulatoren flexibel gehalten werden. Das bedeutet, daß "zugleich" mehrere Betriebssystemdomänen, aufgrund der flexiblen Handhabung zur Trap-Behandlung durch den MOOSE-Kernel, in einem System vorhanden sein können. Die Modellierung virtueller Maschinen wird hiermit in elementarer Weise und ohne besondere Hardware-Anforderungen ermöglicht.

Die Flexibilität, die die Propagation von Emulator-Traps zu den entsprechenden *Server*-Prozessen ermöglicht, kann ihre letzte Konsequenz in dem Aufbau einer Hierarchie von virtuellen Maschinen besitzen. In diesem Zusammenhang wird nur die zu erwartende Laufzeiteinbuße die Emulation von Systemdiensten –über verschiedene Hierarchien von virtuellen Maschinen– ab einer bestimmten Ebene nicht mehr attraktiv erscheinen lassen. Der MOOSE-Kernel wird hierbei jedoch keine technischen Einschränkungen festlegen.

3.7.5. Anbindung von Server-Prozessen an Trap-/Interrupt-Vektoren

Die wesentliche Funktionalität des MOOSE-Kernels im Zusammenhang mit der Behandlung von Traps und Interrupts besteht darin, die durch den realen Prozessor angezeigten Traps und Interrupts im System zu propagieren. Hierzu muß in erster Linie die Möglichkeit bestehen, auf die vom realen Prozessor signalisierten Traps bzw. Interrupts reagieren zu können, d.h. eine Behandlung auf der Kernelebene anzustreben.

Traps und Interrupts werden von dem realen Prozessor über ausgezeichnete Schnittstellen zur Kernelebene propagiert. Diese Schnittstellen werden durch die Trap-Vektoren und Interrupt-Vektoren des realen Prozessors repräsentiert. Damit die Propagation von Traps/Interrupts zur Systemebene möglich ist, muß der Kernel somit eine Verbindung zwischen dem Trap-/Interrupt-Vektor des realen Prozessors und dem entsprechenden *Server*-Prozeß der Systemebene einrichten.

Die Verbindung zwischen den Trap-/Interrupt-Vektoren und den *Server*-Prozessen wird im wesentlichen durch zwei Komponenten definiert. Diese Komponenten stellen zum einen die Identifikation des jeweiligen *Server*-Prozesses dar und definieren zum anderen den Kommunikationspunkt, über den der entsprechende *Server*-Prozeß erreichbar ist. Auf Grundlage dieser beiden Informationen ist es möglich eine Abbildung anzugeben, die, als Konsequenz eines signalisierten Traps/Interrupts, direkt die Verbindung zu einem *Server*-Prozeß herstellt.

3.7.5.1. Abstraktion von Trap-/Interrupt-Vektoren

Zur Propagation von Traps und Interrupts interpretiert der Kernel lediglich die zur Abbildung auf einen *Server*-Prozeß notwendigen Informationen. Diese Informationen sind in einer Datenstruktur enthalten, die eine Abstraktion von den Trap-/Interrupt-Vektoren des jeweiligen realen Prozessors definiert. Die dazu in Anwendung gebrachte Datenstruktur des MOOSE-Kernels wird als *Gate* bezeichnet. Bild 3.12 skizziert die Lage eines *Gates* zwischen den Trap-/Interrupt-Vektoren des realen Prozessors und den *Server*-Prozessen.

MOOSE-Kernels wird als *Gate* bezeichnet. Bild 3.12 skizziert die Lage eines *Gates* zwischen den Trap-/Interrupt-Vektoren des realen Prozessors und den *Server*-Prozessen.

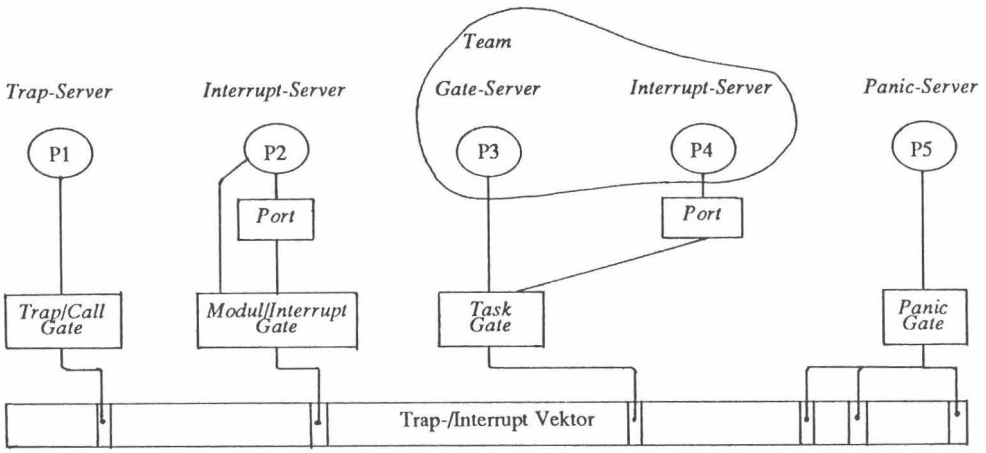

Bild 3.12: Abstraktion von Trap-/Interrupt-Vektoren

Die Identifikation des mit einem *Gate* in Verbindung stehenden *Server*-Prozesses wird durch einen Zeiger auf seinen PCB repräsentiert. Der Kommunikationspunkt wird als Zeiger auf einen *Event-Port* vermerkt. Die Erläuterung der Begriffe, die in diesem Beispiel verwendet worden sind, ist den nachfolgenden Abschnitten zu entnehmen.

Die Assoziation zwischen *Server*-Prozessen und den Trap-/Interrupt-Vektoren ist dynamisch. Dies wird dadurch ermöglicht, indem die Beziehung zwischen Trap-/Interrupt-Vektoren und *Gates* sowie zwischen *Gates* und *Server*-Prozessen bzw. *Event-Ports* manipuliert werden kann. Das bedeutet, daß *Server*-Prozesse den einzelnen Traps bzw. Interrupts nicht statisch zugeordnet sind. Der MOOSE-Kernel stellt hierfür Mechanismen zur Verfügung, die es *Server*-Prozessen ermöglichen, sich an beliebige Trap-/Interrupt-Vektoren des realen Prozessors anbinden zu können. Ebenso ist es den *Server*-Prozessen möglich, die Verbindung zu bestimmten Trap-/Interrupt-Vektoren aufzulösen.

3.7.5.1.1. Server-Gates

Die *Server-Gates* dienen ausschließlich zur Propagation der Traps/Interrupts durch den MOOSE-Kernel, die ihrerseits von dem realen Prozessor zum MOOSE-Kernel signalisiert worden sind. Je nach Typ der zu propagierenden Ereignisse, ergibt sich für die *Server-Gates* die nachfolgend skizzierte Klassifikation:

Trap-Gates dienen zur Anbindung von *Trap-Servern* an Trap-Vektoren des realen Prozessors. Über diese *Gates* werden alle vom realen Prozessor signalisierten Traps vom MOOSE-Kernel an den jeweils angebundenen *Server*-Prozeß weitergeleitet;

Modul-Gates dienen zur Anbindung von *Interrupt-Servern* an Interrupt-Vektoren des realen Prozessors. Über diese *Gates* wird die prozedurale Aktivierung der *Interrupt-Handler* durch den MOOSE-Kernel gesteuert. Die jeweiligen *Interrupt-Handler* sind dem Adreβraum ihres *Teams* zugeordnet;

Task-Gates dienen zur Anbindung von *Interrupt-Servern* an Interrupt-Vektoren des realen Prozessors. Über diese *Gates* wird die Aktivierung von Prozessen, den *Gate-Servern*, aufgrund des jeweils vom realen Prozessor signalisierten Interrupts durch den MOOSE-Kernel gesteuert.

Werden *Server-Gates* zur Anbindung von Prozessen an Trap-/Interrupt-Vektoren des realen Prozessors verwendet, so erfolgt die Benachrichtigung der entsprechenden Ereignisse zu den jeweiligen *Server*-Prozessen ohne weitere Eingriffe des Kernels. Die Benachrichtigung wird mit Hilfe der üblichen Mechanismen zur Interprozeβkommunikation des MOOSE-Kernels erreicht.

Die *Trap-Gates* ermöglichen die Identifikation von *Trap-Servern*, denen eine *Trap-Message* zugesandt wird, wenn der reale Prozessor einen Trap signalisiert hat. Die *Modul-* und *Task-Gates* ermöglichen die Identifikation von *Interrupt-Servern*, denen ein *Interrupt-Signal* zugesandt wird, wenn der reale Prozessor einen Interrupt signalisiert hat. Die *Task-Gates* ermöglichen die Identifikation des Prozesses, der zur Abkapselung eines *Interrupt-Handlers* dient und der mit dem jeweils signalisierten Interrupt gestartet werden soll.

3.7.5.1.2. Kernel-Gates

Die *Kernel-Gates* ermöglichen eine Vorverarbeitung der vom realen Prozessor signalisierten Traps/Interrupts durch den MOOSE-Kernel, bevor diese Ereignisse zu den entsprechenden *Server*-Prozessen propagiert werden. Je nach Typ der zu propagierenden Ereignisse, ergibt sich für *Kernel-Gates* die nachfolgend skizzierte Klassifikation:

Call-Gates dienen zur Aufnahme von *Kernel-Calls*. Über diese *Gates* wird die Übergabe der Argumente vom Benutzer- zum Kerneladreβraum und die Rückgabe der Resultate vom Kernel- zum Benutzeradreβraum gesteuert;

Interrupt-Gates dienen zur Anbindung von *Interrupt-Servern* an Interrupt-Vektoren des realen Prozessors. Über diese *Gates* wird die prozedurale Aktivierung von *Interrupt-Handlern* durch den MOOSE-Kernel gesteuert. Die jeweiligen *Interrupt-Handler* sind dem Kerneladreβraum zugeordnet.

Die Aufgabe der *Kernel-Gates* kann sehr vielfältig interpretiert werden. Zum einen sind die *Call-Gates* hervorzuheben, die schlieβlich einen kontrollierten Wechsel zwischen Benutzer-/Systemebene und Kernelebene ermöglichen. Über diese *Gates* werden sämtliche Dienste des MOOSE-Kernels zur Verfügung gestellt. Das bedeutet insbesondere für den Kernel die Möglichkeit zur Abstraktion von der initialen- und terminalen Phase eines *Kernel-Calls*.

Die *Interrupt-Gates* steuern die Aktivierung von *Interrupt-Handlern*, die, aus Gründen zur Verbesserung der Laufzeiteffizienz, der Kernelebene zugeordnet sind. Über diese Klasse von *Gates* werden mindestens die *Clock-Interrupts* zum Kernel propagiert. Desweiteren ermöglichen diese *Gates* weiterhin die Identifikation von *Interrupt-Servern*, denen ein

Interrupt-Signal übermittelt werden kann, nachdem der reale Prozessor den Interrupt zum Kernel propagiert und der Kernel seinerseits eine entsprechende Behandlung des Interrupts durchgeführt hat.

3.7.5.2. Assoziation zwischen Trap-/Interrupt-Vektoren und Server-Prozessen

Das *Gate*-Modell des MOOSE-Kernels stellt die Grundlage dar, dynamisch rekonfigurierbare Systeme zur Behandlung von Traps und Interrupts technisch ermöglichen zu können. Dies findet im wesentlichen in zwei Tatsachen seine Begründung. Zum einen werden *Gates* dynamisch vergeben, nämlich genau dann, wenn ein *Server*-Prozeβ die Anbindung an einen Trap-/Interrupt-Vektor anfordert. Die Freigabe des *Gates* erfolgt analog, d.h. mit der Anforderung durch einen *Server*-Prozeβ zur Auflösung einer Verbindung zum jeweiligen Trap-/Interrupt-Vektor. Zum anderen können die frei zur Disposition stehenden *Gates* mit beliebigen Trap-/Interrupt-Vektoren des realen Prozessors in Verbindung gebracht werden.

Die *Server*-Prozesse selbst entscheiden, ob das über einen bestimmten Trap-/Interrupt-Vektor signalisierte Ereignis als Trap oder Interrupt propagiert werden soll. Je nach dem in welcher Form die Propagation des Ereignisses stattfinden soll, plaziert der MOOSE-Kernel ein entsprechend klassifiziertes *Gate* zwischen dem betreffenden Trap-/Interrupt-Vektor und dem behandelnden (und anfordernden) *Server*-Prozeβ. Dies erfolgt zum Zeitpunkt der Anforderung eines *Server*-Prozesses zur Anbindung an einen bestimmten Trap-/Interrupt-Vektor.

Die Auflösung einer Verbindung zu einem Trap-/Interrupt-Vektor bedeutet, daβ das jeweilige *Gate* von dem ihm zugeordneten Trap-/Interrupt-Vektor abgekoppelt wird. Um jedoch zu vermeiden, daβ in diesem Fall nachfolgend auftretende Traps-/Interrupts nicht in unkontrollierter Weise den Kernel aktivieren, wird der betreffende Trap-/Interrupt-Vektor mit einem ausgezeichneten *Gate* in Verbindung gebracht. Dieses *Gate* wird als *Panic-Gate* bezeichnet. Durch einen *Server*-Prozeβ, der sich mit dem *Panic-Gate* assoziiert, können somit Traps und Interrupts, für die eigentlich keine *Server*-Prozesse im System mehr vorhanden sind, abgefangen werden. In besonderen Fällen ist es solch einem *Panic-Server* möglich, eine angemessene Behandlung dieser extremen Ausnahmesituation so durchführen zu können, daβ die Funktionalität des Systems weiterhin gewährleistet werden kann.

3.8. Propagation von Traps und Interrupts

Die Behandlung von Traps und Interrupts durch den MOOSE-Kernel wird durch die mit den entsprechenden Trap-/Interrupt-Vektoren assoziierten *Gates* kontrolliert. Ist dem jeweiligen Trap-/Interrupt-Vektor ein *Kernel-Gate* vorgeschaltet, so führt der MOOSE-Kernel eine eigenständige Behandlung des Traps/Interrupts durch, bevor eine Propagation dieser Ereignisse zur Systemebene erfolgt. Ist dagegen ein *Server-Gate* vorgeschaltet, so wird der MOOSE-Kernel nur die Propagation des entsprechenden Traps/Interrupts und damit keine eigenständige Behandlung durchführen.

Für jede Klasse von *Gates* steht ein bestimmter *Gate-Handler* innerhalb des MOOSE-Kernels zur Verfügung. Dieser *Gate-Handler* wird immer dann aktiviert, wenn über den mit dem betreffenden *Gate* assoziierten Vektor ein Trap oder Interrupt signalisiert wird. Die Semantik des *Gate-Handlers* bestimmt damit, wie zur weiteren Behandlung des Traps bzw.

Interrupts vorgegangen werden soll, d.h., ob die Traps/Interrupts direkt zu *Server*-Prozessen propagiert oder zuerst im Kernel selbst behandelt werden sollen.

Die Aktivitäten der jeweiligen *Gate-Handler* finden unter Kontrolle des einen Trap auslösenden bzw. des durch einen Interrupt unterbrochenen Prozesses statt. Da die *Gate-Handler* jedoch im Kerneladreβraum angesiedelt sind, finden ihre Aktivitäten im privilegierten *Kernelmode* statt. Dies entspricht dem Prinzip der Kernelprozesse im Zusammenhang mit prozedurorientierten Betriebssystemen. Die *Gate-Handler* laufen somit immer unter Kontrolle eines bestimmten Kernelprozesses. Dennoch besteht ein wesentlicher Unterschied zwischen Kernelprozessen in prozeβorientierten- und prozedurorientierten Systemen. Der Unterschied ergibt sich dadurch, daβ die Kernelprozesse in prozeβorientierten Systemen üblicherweise erheblich geringer in ihrer Komplexität sind, als es Kernelprozesse in prozedurorientierten Systemen darstellen. Dieser Komplexitätsunterschied hat seine Ursache in den verschiedenen Funktionalitäten der jeweiligen Kernel.

Zur Propagation von Traps und Interrupts sind verschiedene Strategien notwendig. Diese Strategien unterscheiden sich je nach der Semantik des jeweils aktivierten *Gate-Handlers,* d.h. nach der Klasse der mit den Traps/Interrupts assoziierten *Gates*.

3.8.1. Interaktion mit dem Trap-Server

Trap-Server nehmen die vom MOOSE-Kernel propagierten Traps über *Trap-Gates* an. Sobald ein Trap ausgelöst wird, formuliert der im Kernel residente *Gate-Handler* eines *Trap-Gates* eine *Trap-Message*. Diese Nachricht wird dem zur Behandlung des Traps zuständigen *Server*-Prozeβ mit Hilfe der üblichen Mechanismen zur Interprozeβkommunikation des MOOSE-Kernels zugesandt. Die Behandlung des in dieser Weise propagierten Traps, d.h. die Abarbeitung der entsprechenden *Trap-Message*, übernimmt der zuständige *Trap-Server*.

Die im Zusammenhang mit der Propagation eines Traps ablaufenden Aktivitäten bedeuten, daβ der gegenwärtig aktive Kernelprozeβ als *Rendezvous-Client* dem *Trap-Server*, der selbst den *Rendezvous-Server* repräsentiert, gegenüberstehen wird. Da die Identifikation dieses Kernelprozesses jedoch identisch mit der des aktiven Benutzerprozesses ist, erfährt der *Trap-Server* mit dem Empfang der *Trap-Message* gleichzeitig die Identifikation des den Trap verursachenden Prozesses.

Prinzipiell besteht die Möglichkeit mit jedem Trap einen eigenen *Trap-Server* zu assoziieren. Ebenso kann aber auch derselbe *Trap-Server* verschiedene Traps annehmen, d.h. die Propagation von unterschiedlichen Traps durch den MOOSE-Kernel kann zur Identifikation nur eines behandelnden *Trap-Servers* führen. Bild 3.13 skizziert hierzu die mögliche Beziehung zwischen *Gates* und *Trap-Server*.

Die Unterscheidung welcher Trap dem *Trap-Server* propagiert worden ist, wird anhand der *Trap-Message* selbst möglich sein. Neben dem Prozessorzustand des den Trap auslösenden Prozesses, sind in einer *Trap-Message* noch andere Informationen enthalten. Diese Informationen werden von dem jeweiligen *Gate-Handler* hinzugefügt und enthalten u.a. die Identifikation des Trap-Vektors, über den der Trap vom realen Prozessor signalisiert worden ist.

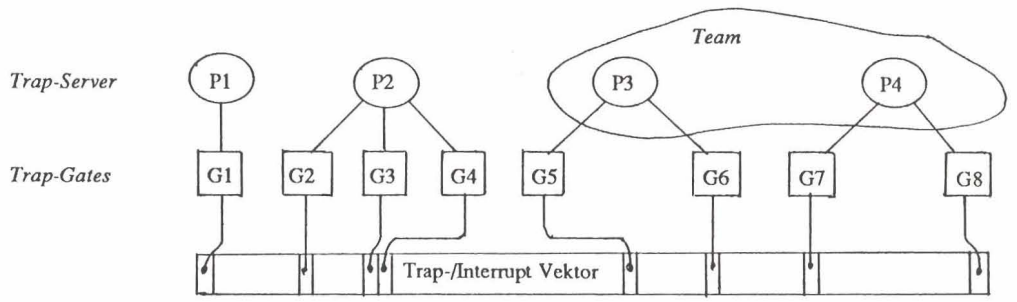

Bild 3.13: Beziehung zwischen *Gates* und *Trap-Server*

3.8.1.1. Die Bedeutung des Rendezvous-Konzeptes bei der Behandlung von Traps

Die Kommunikation mit dem jeweiligen *Trap-Server* basiert auf Anwendung der üblichen vom MOOSE-Kernel zur Verfügung gestellten Mechanismen zur Interprozeβkommunikation. Dies bedeutet erstens, daβ der *Trap-Server* für den Empfang der *Trap-Message* keinen speziellen Mechanismus benötigt. Und zweitens bedeutet dies, daβ der den Trap auslösende Prozeβ mit der Übermittlung der *Trap-Message* blockiert, da dieser Prozeβ in ein *Rendezvous* mit dem jeweiligen *Trap-Server* geführt wird. Der einen Trap auslösende Prozeβ muβ somit explizit, mit Beendigung eines *Rendezvous*, reaktiviert werden.

Der Kernel selbst ist während der Behandlung des Traps nicht blockiert und kann somit jederzeit die normale wie auch die durch Ausnahmesituationen bedingte Kommunikation zwischen Prozessen ermöglichen. Insbesondere können mit dieser Grundlage mehrere Emulator-Traps "zugleich" behandelt werden, die jeweils unterschiedlichen Betriebssystemdomänen zuzuordnen sind.

Die Reaktivierung eines "getrappten" Prozesses erfolgt durch die übliche Beendigung eines *Rendezvous*. Hierzu gibt der *Trap-Server* eine *Reply-Message* an den den Trap auslösenden Prozeβ zurück. Mit der Rückgabe dieser *Reply-Message* wird dann zuerst der entsprechende Kernelprozeβ und damit der jeweilige *Gate-Handler* des Traps reaktiviert. Erst mit Beendigung der Aktivitäten des *Gate-Handlers* wird der Benutzerprozeβ reaktiviert.

Diese Vorgehensweise ermöglicht es, daβ der *Gate-Handler* noch spezifische Maβnahmen für den jeweiligen Benutzerprozeβ durchführen kann. Diese Maβnahmen bedeuten insbesondere den Prozessorzustand eines Prozesses, nämlich den des zu reaktivierenden Benutzerprozesses, wieder herzustellen. Hierzu wird der jeweilige Prozessorzustand der vom *Trap-Server* übermittelten *Reply-Message* entnommen. Das bedeutet, daβ der Prozessorzustand für einen "getrappten" Prozeβ jeweils vom *Trap-Server* bestimmt wird.

Die für einen Prozeß vollständig transparente Behandlung eines Traps durch einen *Trap-Server* würde bedeuten, daß der vom *Gate-Handler* aufgebaute und zum *Trap-Server* übermittelte Prozessorzustand des "getrappten" Prozesses, ohne Modifikation durch den *Trap-Server*, in Form einer *Reply-Message* an den *Gate-Handler* zurückgesendet wird. Das bedeutet, daß der *Gate-Handler* die von ihm aufgebaute *Trap-Message* zurückerhält. Im Zusammenhang mit der Behandlung bestimmter Traps ist jedoch die Manipulation des Prozessorzustandes durch einen *Trap-Server* notwendig.

Die Behandlung eines *Page-Faults* ist in diesem Zusammenhang ein typisches Beispiel dafür. Die Reaktivierung des einen *Page-Fault* auslösenden Prozesses muß zur erneuten Ausführung der Instruktion führen, die den *Page-Fault* verursacht hat. Hierzu ist oftmals die Modifikation des Prozessorzustandes für diesen Prozeß notwendig, d.h. der *Instruction-Pointer* des Prozesses ist auf die betreffende Instruktion zurückzusetzen.

Die Emulation von Systemdiensten stellt ein anderes Beispiel dar, das die Manipulation des Prozessorzustandes motiviert. Die Argumente und Resultate der zu emulierenden *System-Calls* werden oftmals in Registern gehalten, um einen problemlosen Austausch von Informationen zwischen Benutzer- und Systemebene bei prozedurorientierten Systemen zu ermöglichen. Demzufolge muß der Prozessorzustand nicht nur gelesen werden, um die Argumente der *System-Calls* zu extrahieren. Es muß ebenso die Möglichkeit bestehen, den Prozessorzustand zu modifizieren, damit die Resultate an den Benutzerprozeß zurückgeliefert werden können.

3.8.2. Modelle zur Behandlung von Interrupts

Ebenso wie bei der Trap-Behandlung, findet im MOOSE-System die Behandlung von Interrupts durch *Server*-Prozesse statt, d.h. die *Interrupt-Server* können auf den Eintritt des Interrupts warten. Aufgrund des asynchronen Charakters eines Interrupts, erfolgt die Kommunikation zwischen Kernel und *Server*-Prozeß bzw. zwischen Gerät und *Server*-Prozeß jedoch nach anderen Prinzipien.

Da kein Gerät direkt mit Prozessen kommunizieren kann, muß dafür ein durch Software realisiertes Verbindungsglied, der *Interrupt-Handler*, eingesetzt werden. Die Funktionalität des *Interrupt-Handlers* sollte so elementar wie möglich sein [Loehr 1980]. Im Idealfall bedeutet dies, nur den Interrupt zu quittieren, d.h. die "Hardware" darüber zu informieren, daß der Interrupt angenommen worden ist, und es bedeutet auch gleichzeitig, die Kommunikation mit dem entsprechenden *Server*-Prozeß zu ermöglichen, d.h. die "Software" darüber zu informieren, daß ein Interrupt aufgetreten ist.

Die Realisierung von *Interrupt-Handlern* wird im Zusammenhang mit dem MOOSE-System durch zwei voneinander unabhängige Mechanismen ermöglicht. Neben einer prozedurorientierten Beziehung zwischen Kernel und *Interrupt-Handler*, dargestellt durch *Gate-Links*, existiert als Alternative die prozeßorientierte Beziehung, durch *Gate-Server* repräsentiert. Gemeinsam haben diese Varianten, daß die *Interrupt-Handler* Zugriff auf den Adreßraum des *Server*-Prozesses erhalten und somit direkt und effizient Daten des *Server*-Prozesses manipulieren, d.h. Ein-/Ausgabeanforderungen abarbeiten können. Der Unterschied zwischen beiden Varianten liegt in der Identifikation des *Interrupt-Handlers*. Unter MOOSE kann der *Interrupt-Handler* durch die Realisierung als *Gate-Server* eine eigene

Prozeβidentifikation erhalten und würde somit nicht mit der Identifikation des durch ihn unterbrochenen Prozesses laufen. Hierdurch können somit jedem *Interrupt-Handler* eigene Zugriffsrechte zugeordnet werden.

3.8.2.1. Prozedurale Beziehung zwischen Gate- und Interrupt-Handler

Die übliche Art und Weise, Interrupts zu behandeln, stellt eine prozedurale Beziehung zwischen Kernel (genauer: zwischen dem vom Interrupt unterbrochenen Prozeβ) und *Interrupt-Handler* dar. Die Verbindung zwischen diesen beiden Komponenten erfolgt unter MOOSE über den *Gate-Link*. Als wesentliche Funktionalität eines *Gate-Handlers* zur prozeduralen Aktivierung des *Interrupt-Handlers*, muβ die Identifikation und Definition des Adreβraumes für den zu aktivierenden *Interrupt-Handler* erbracht werden.

3.8.2.1.1. Interrupt-Behandlung auf der Systemebene

Zur Interrupt-Behandlung auf der Systemebene, erfolgt die Adreβraumdefinition nicht durch einen Prozeβwechsel. Die Definition des Adreβraumes orientiert sich lediglich an dem *Team*, das für den *Interrupt-Handler* zuständig bzw. in dem der entsprechende *Server*-Prozeβ angesiedelt ist. Sobald sich ein *Interrupt-Server* beim Kernel identifiziert hat, ist dieses *Team* sowie der *Interrupt-Handler* dem Kernel bekannt.

Die Selektion des *Interrupt-Handlers* durch einen *Gate-Handler* für *Modul-Gates* erfolgt in zwei Schritten. Der erste Schritt entspricht der Aktivierung des *Gate-Handlers* selbst. Dieser identifiziert den Adreβraum des *Interrupt-Handlers*, selektiert den *Interrupt-Handler* und ermöglicht die Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server*. Der *Gate-Handler* steht in direkter Beziehung zu dem Interrupt-Vektor des realen Prozessors.

Der zweite Schritt führt zur Aktivierung des geräteabhängigen *Interrupt-Handlers* auf der Systemebene inklusive Adreβraumdefinition. Dieser *Interrupt-Handler* wird über einen sogenannten virtuellen Interrupt-Vektor selektiert, der im Adreβraum des dem *Interrupt-Server* zugeordneten *Teams* angelegt ist. Die Größe dieses Vektors entspricht der Größe des Interrupt-Vektors des realen Prozessors. Um den *Interrupt-Handler* durch den *Gate-Handler* schnell identifizieren zu können, entspricht jeder Eintrag *i* des realen Interrupt-Vektors einem Eintrag *i* des virtuellen Interrupt-Vektors. Bild 3.14 skizziert dieses Modell.

Zum Zeitpunkt des Interrupts verfügt der *Gate-Handler* über Informationen, die eine eindeutige Identifikation des Interrupt-Vektors zulassen, über den der Interrupt signalisiert worden ist. Aufgrund der klaren Beziehung zwischen beiden Vektoren, kann direkt der entsprechende virtuelle Interrupt-Vektor und somit der *Interrupt-Handler* identifiziert werden. Die Definition eines virtuellen Interrupt-Vektors wird duch den *Interrupt-Server* selbst durchgeführt: die Adresse des *Gate-Links* für den jeweiligen Interrupt wird in den entsprechenden Eintrag *i* des virtuellen Interrupt-Vektors eingetragen. Diese Maβnahme erfolgt üblicherweise bevor sich der *Interrupt-Server* an den Eintrag *i* des realen Interrupt-Vektors zur Annahme von Interrupts anbindet.

Der *Gate-Link* ist direkt der Komponente zur Interrupt-Behandlung zugeordnet und nicht dem Adreβraum des Kernels zuzurechnen. Er übernimmt für den Ablauf der Interrupt-Behandlung zwei wesentliche Funktionalitäten. Erstens sorgt er für eine entsprechende

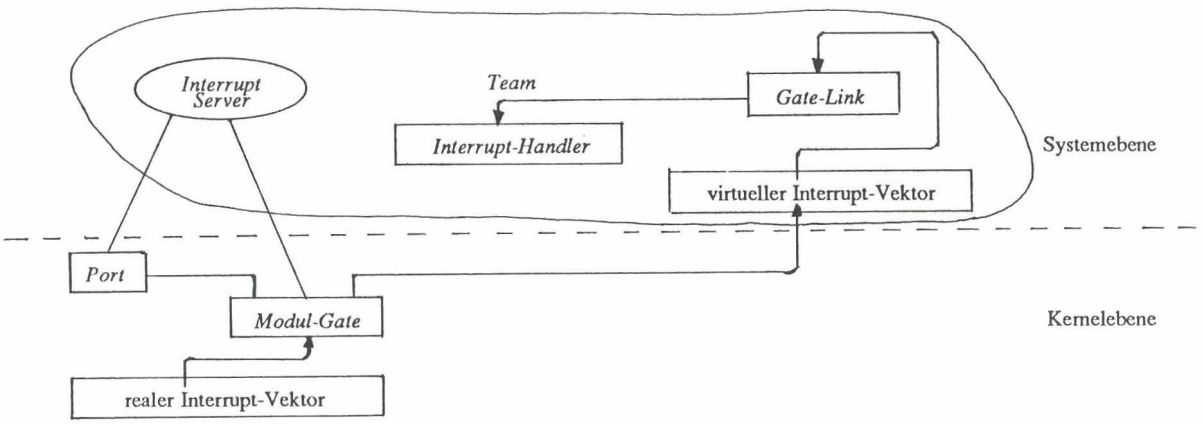

Bild 3.14: Propagation von Interrupts zur Systemebene

Parameterisierung des *Interrupt-Handlers*, so daß die Möglichkeit besteht, einem *Interrupt-Handler* z.B. mehrere Geräte zuordnen zu können. Zweitens stellt er entsprechend der zugrunde liegenden Hardware den technischen Rahmen zur Verfügung, einen, wenn auch in stark idealisierter Form, *remote procedure call* [Nelson 1982] auf der vom Kernel bereits entfernten Seite anzunehmen und von diesem zurückzukehren.

Aus Sichtweise der *Interrupt-Handler*, erfolgt deren Aktivierung unabhängig von ihrer Adreßraumzugehörigkeit. Ihre Schnittstelle würde sich nicht ändern, wenn der *Gate-Link* dem Kerneladreßraum zugeordnet werden würde und somit nur noch eine übliche prozedurale Beziehung zur Aktivierung des *Interrupt-Handlers* notwendig wäre.

3.8.2.1.2. Interrupt-Behandlung auf der Kernelebene

Bestimmte hardware-technische Randbedingungen können die vollständige Behandlung von Interrupts auf der Systemebene als nicht zweckmäßig erscheinen lassen. Hierbei sind insbesondere Systeme zu betrachten, die auf sehr komplexer MMU-Hardware basieren[32]. Die Behandlung von Interrupts auf der Systemebene bedeutet schließlich die Einblendung des entsprechenden Adreßraumes für den *Interrupt-Handler*. Je nach zugrunde liegender Hardware kann dieser Vorgang erhebliche Laufzeiten mit sich bringen, die im Zuge der Behandlung von Interrupts nicht akzeptabel sind.

Der MOOSE-Kernel unterstützt deshalb eine weitere Variante der Interrupt-Behandlung auf Grundlage des *Gate-Link* Modells. Diese Variante vermeidet die zusätzliche Adreßraumdefinition für den *Interrupt-Handler*. Das bedeutet jedoch auch, daß die Interrupt-Behandlung im Kerneladreßraum stattfindet und daß demzufolge die *Interrupt-Handler* auf der

[32] Ein typisches Beispiel stellt in diesem Zusammenhang die mc68451 dar. Diese MMU findet mit dem mc68000/10 eine sehr weite Verbreitung, erschwert jedoch im erheblichen Maße generell die Realisierung von prozeßorientierten Betriebssystemen.

Kernelebene angesiedelt sind. Dies bedingt jedoch nicht notwendigerweise den Verlust an Flexibilität eines Interrupt-Systems. Zur Ein-/Ausgabe von Daten besitzen die *Interrupt-Handler* mit ihren *Interrupt-Servern* durch *Shared-Memory* den direkten Zugriff auf gemeinsame Datenstrukturen.

Die Identifikation der *Interrupt-Handler* orientiert sich auch hier anhand einer zweiten Abbildungstabelle. Diese Tabelle entspricht in ihren Ausmaßen und ihrem Aufbau jedoch nicht dem virtuellen Interrupt-Vektor. Jeder Eintrag dieses *Handler*-Vektors enthält zum einem die Adresse des *Interrupt-Handlers* und ein Argument für diesen *Interrupt-Handler*. Die Identifikation des *Interrupt-Handlers* anhand des *Handler*-Vektors richtet sich nach der mit einem *Interrupt-Gate* assoziierten Vektornummer. Diese Vektornummer unterscheidet sich im allgemeinen von der des Interrupt-Vektors des realen Prozessors. Bild 3.15 skizziert dieses Modell.

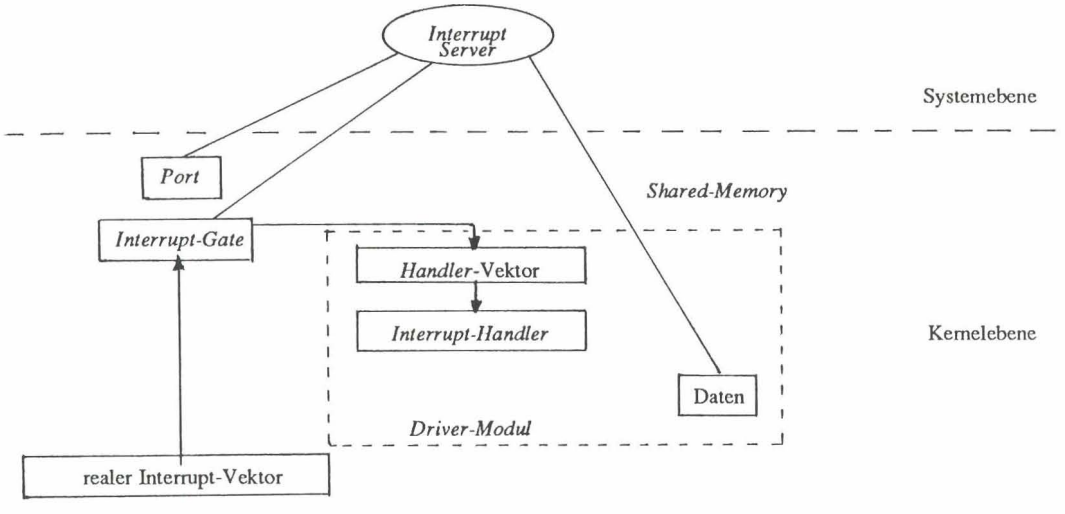

Bild 3.15: Propagation von Interrupts zur Kernelebene

Die Aktivierung des geräteabhängigen *Interrupt-Handlers* auf der Kernelebene erfolgt mit der Übergabe eines Argumentes für den jeweiligen *Interrupt-Handler*. Die logische Entkopplung vom MOOSE-Kernel ist durch diese Organisation immer noch gegeben. Ebenso erfolgt die Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* nach den gleichen Prinzipien wie bei der Aktivierung eines *Interrupt-Handlers* über *Modul-Gates*.

3.8.2.2. Prozeßbeziehung zwischen Gate- und Interrupt-Handler

Die *Gate-Server* stellen im MOOSE-System die Alternative bei der Aktivierung eines *Interrupt-Handlers* dar. Im Vergleich mit den *Gate-Links*, ermöglichen die *Gate-Server* dieselbe Flexibilität bei der Entwicklung und Realisierung eines Ein-/Ausgabesystems. In beiden Fällen sind die jeweiligen *Teams*, in denen *Interrupt-Server* angesiedelt sind, unabhängig vom Kernel und können zur Laufzeit des Systems rekonfiguriert werden. Der wesentliche Unterschied zwischen *Gate-Server* und *Gate-Links* ergibt sich in der Identifikation des *Interrupt-Handlers*. *Gate-Server* stellen immer denselben Kontext für einen *Interrupt-*

Handler zur Verfügung. Bei den *Gate-Links* findet nur eine Adreβraumdefinition, jedoch kein Prozeβwechsel statt. Das bedeutet, der *Interrupt-Handler* läuft zwar innerhalb des Adreβraumes seines *Teams*, ist jedoch mit der Identifikation des durch ihn unterbrochenen Prozesses behaftet. Die *Gate-Server* erlauben es den *Interrupt-Handlern* eine eigene Prozeβidentifikation zukommen zu lassen, d.h. für den *Interrupt-Handler* wird ein eigener Prozeβ gestartet.

Der Vorteil im Zusammenhang mit den *Gate-Servern* liegt darin, daβ die Auswirkungen von Fehlern, die von *Interrupt-Handlern* produziert werden, sich im wesentlichen auf einen Prozeβ beschränken [33]. Die *Interrupt-Handler* sind konkret von den restlichen Aktivitäten eines MOOSE-Betriebssystems abgekapselt. Insbesondere wird durch diesen Mechanismus der Kernel in die Lage versetzt, Zugriffsrechte der *Interrupt-Handler* überprüfen zu können. Diese Zugriffsrechte resultieren von den jeweiligen Zugriffsattributen des *Gate-Servers* her, der einen bestimmten *Interrupt-Handler* kontrolliert.

Gate-Server werden von ihrem *Server*-Prozeβ erzeugt und beim Kernel angemeldet. Der *Server*-Prozeβ gibt vor Erzeugung des *Gate-Servers* dem Kernel bekannt, mit welcher Variante der Interrupt behandelt und über welchen realen Vektor dieser erwartet wird. Nach seiner Identifikation beim Kernel, wird sich der *Gate-Server* auf die von ihm kontrollierten Interrupts synchronisieren. Sobald die Zuständigkeit des *Gate-Servers* für einen aufgetretenen Interrupt vom *Gate-Handler* für *Task-Gates* erkannt worden ist, wird dieser Prozeβ gestartet und dadurch der *Interrupt-Handler* aktiviert. Abbildung 3.16 skizziert das Modell eines *Teams* bestehend aus zwei *Gate-Servern* und einem *Interrupt-Server*.

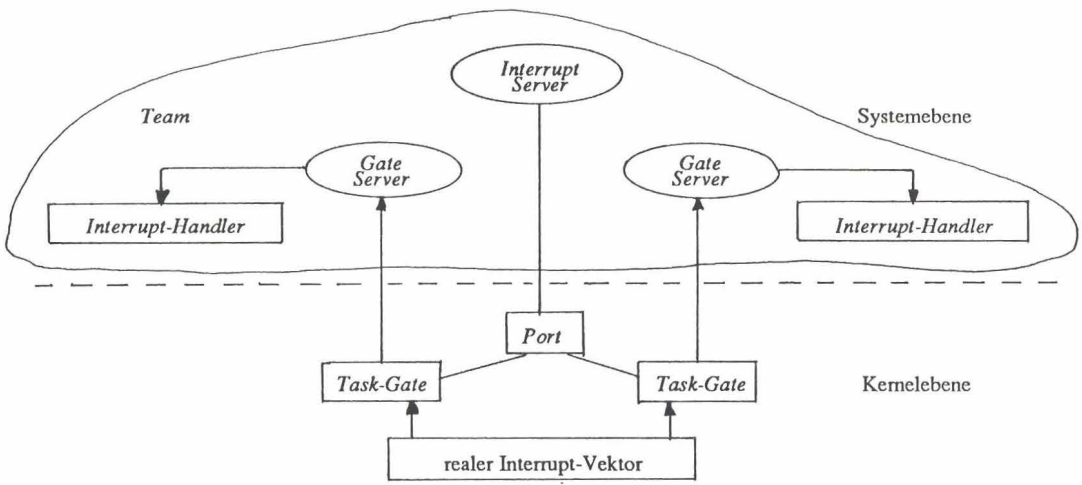

Bild 3.16: Propagation von Interrupts zu *Gate-Servern*

[33] Das prinzipielle Problem, daβ z.B. DMA-Geräte zwecks Ein-/Ausgabe mit absoluten Adressen des Systems versorgt werden müssen, bleibt in jedem Fall bestehen. Zumindest können logische Fehler der *Interrupt-Handler* eingegrenzt werden.

Die *Gate-Server* können jeweils verschiedene oder denselben *Interrupt-Handler* abkapseln. Wird durch mehrere *Gate-Server* derselbe *Interrupt-Handler* abgekapselt, so entspricht dies der Organisation eines Systems, in dem mehrere Geräte von nur einem *Interrupt-Handler* gesteuert werden.

Die Realisierung über einen eigenen Prozeß beeinflußt den *Interrupt-Handler* in keiner Weise, so daß allein durch den *Server*-Prozeß festgelegt wird, mit welcher Laufzeitumgebung (*Gate-Link* oder *Gate-Server*) der *Interrupt-Handler* zu versehen ist.

3.8.3. Interaktion mit dem Interrupt-Server

Bevor ein Gerät Interrupts produzieren kann, muß es üblicherweise erst einen Ein-/Ausgabeauftrag erhalten, d.h. es muß in einen entsprechenden Arbeitsmodus gebracht werden. Je nach Konzeption des Ein-/Ausgabesystems, übernimmt diese Aufgabe ständig der *Interrupt-Server*, oder sie wird ihm nur dann aufgetragen, wenn das Gerät neu programmiert werden muß, d.h. wenn eine Folge von Ein-/Ausgabeoperationen des Gerätes abgeschlossen ist. In jedem Fall muß der *Interrupt-Server* über den jeweiligen Arbeitszustand des Gerätes informiert sein, um jederzeit entsprechende Maßnahmen zur Kontrolle des Gerätes einleiten zu können.

Es ist Aufgabe des *Interrupt-Handlers*, Gerätezustände festzuhalten und dem *Interrupt-Server* die entsprechenden Zustandsinformationen zukommen zu lassen. Da ebenso mit dem Quittieren eines Interrupts zusätzlich die Ein-/Ausgabe von Daten verknüpft sein kann, besteht die Kommunikationsverbindung zwischen *Interrupt-Handler* und *Interrupt-Server* aus einem Kontroll- und Datenweg. Über den Kontrollweg synchronisiert sich der *Interrupt-Handler* mit dem *Interrupt-Server*, d.h. der *Interrupt-Handler* signalisiert den (evtl. nicht erfolgreichen) Abschluß eines Ein-/Ausgabeauftrags. Der Datenweg dient zum Austausch von Informationen zwischen *Interrupt-Handler* und *Interrupt-Server*.

3.8.3.1. Kommunikation zwischen Interrupt-Handler und Interrupt-Server

Die *Message-Passing* Primitiven des MOOSE-Kernels ermöglichen zwar gleichzeitig den Kontroll- und Datentransfer, können jedoch trotzdem nicht zur Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* angewendet werden. Die Gründe hierfür sind vielfältig und nicht nur auf Primitiven des MOOSE-Kernels beschränkt, sondern eher konzeptioneller Natur.

3.8.3.1.1. Technische Aspekte

Ein erster Aspekt gegen die Anwendung von *Message-Passing* zur Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* ergibt sich aus der Tatsache, daß der Aufwand im Zusammenhang mit den *Message-Passing* Primitiven relativ hoch ist. Die Folge davon ist, daß die Laufzeiten der *Interrupt-Handler* bei Anwendung der *Message-Passing* Primitiven ebenso entsprechend hoch sein würden. Da die Nachrichten zusätzlich noch unterschiedlich in ihrer Länge sein können, würden desweiteren die Laufzeiten entsprechend variieren, d.h. es könnte keine feste obere Grenze für die Dauer der Interrupt-Behandlung bestimmt werden.

Der zweite Aspekt, der gegen die Anwendung der *Message-Passing* Primitiven spricht, ist in der Tatsache begründet, daß diese Primitiven des MOOSE-Kernels blockierend auf den jeweils anwendenden Prozeß wirken. Die Anwendung solcher Primitiven durch den *Interrupt-Handler* würde somit in jedem Fall zur Blockierung des Prozesses führen, unter dessen Identifikation der *Interrupt-Handler* aktiv ist. Je nach dem betrachteten Modell zur Interrupt-Behandlung, ergeben sich in diesem Zusammenhang zwei wesentliche Problempunkte. Diese Problempunkte beziehen sich einerseits auf die Anwendung von *Message-Passing*, zur Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server*, und auf die Einordnung eines *Interrupt-Handlers* in eine bestimmte Systemstruktur, wenn zur Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* blockierende Mechansimen zur Anwendung gebracht werden.

Message-Passing bedeutet einen Datentransfer vom Adreßraum des sendenden in den Adreßraum des empfangenden Prozesses. Mit Ausnahme bei Anwendung des *Gate-Server* Modells, würde insbesondere der Adreßraum des sendenden Prozesses nicht mit dem Adreßraum des *Interrupt-Handlers* übereinstimmen; der Datentransfer müßte jedoch effektiv aus dem Adreßraum des *Interrupt-Handlers* erfolgen. Eine Sonderbehandlung des Datentransfers für diesen Fall würde allgemein die Laufzeiten der *Message-Passing* Primitiven erhöhen und sich zu Kosten der Interrupt-Behandlung selbst, wie auch zu Kosten der Interprozeßkommunikation zwischen den Prozessen, auswirken.

Nur im Fall der Interrupt-Behandlung auf Grundlage der *Gate-Server* wäre der Adreßraum des sendenden Prozesses mit dem des *Interrupt-Handlers* identisch. Dennoch scheiden auch hierfür die *Message-Passing* Primitiven zur Kommunikation mit dem *Interrupt-Server* aus. Der Grund liegt darin, daß *Gate-Server* grundsätzlich nicht reentrant sind[34)]. Die Blockierung eines *Gate-Servers* bedeutet, daß er nicht mehr in der Lage ist, weitere Interrupts anzunehmen. Es ist hierbei insbesondere zu beachten, daß ein *Gate-Server* aufgrund eines Interrupts direkt gestartet wird. Würde sich der *Gate-Server* in einem *Rendezvous* mit seinem *Interrupt-Server* befinden, hätte ein Interrupt somit die Konsequenz, das *Rendezvous* abzubrechen. Der *Gate-Server* würde in diesem Fall jedoch an einer Stelle reaktiviert werden, an der er nicht in der Lage wäre, sofort den Interrupt zu behandeln. Der Interrupt hätte für den *Gate-Server* die Semantik einer Ausnahmesituation, die von dem *Gate-Server* erst entsprechend interpretiert werden muß. Dieser Ablauf findet in dem klassischen Verständnis über die Reaktivierung von Prozessen seine Begründung (siehe auch [Schindler 1983]).

3.8.3.1.2. Konzeptionelle Aspekte

Nicht nur technische Aspekte sprechen gegen die Anwendung von blockierendem *Message-Passing* zur Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server*. Aus software-technischer Sicht kann ein *Interrupt-Handler* in seiner Struktur nicht mehr als elementar bezeichnet werden, wenn die Kommunikation mit seinem *Interrupt-Server* über

[34)] Das bedeutet jedoch nicht, daß die von einem *Gate-Server* abgekapselten *Interrupt-Handler* ebenfalls nicht reentrant sind. Mehrere *Gate-Server* können zur Abkapselung desselben *Interrupt-Handlers* eingesetzt werden, womit, nach Eintritt eines Interrupts, mehrere *Gate-Server* aktiv sein können; die *Gate-Server* werden sich in diesem Fall gegenseitig unterbrochen haben.

blockierende Mechanismen stattfindet. Dies ist im wesentlichen darin begründet, daß blockierende Kommunikationsprimitiven eine spezielle Ausprägung benutzthierarchischer Beziehungen darstellen. Alle Aktivitäten die der *Interrupt-Server* nach seiner Reaktivierung durch den *Interrupt-Handler* durchführt, sind in diesem Zusammenhang auf die Benutzung von Operationen des *Interrupt-Servers* durch den *Interrupt-Handler* zurückzuführen. Der *Interrupt-Handler* ist in dieser Hinsicht auf einer höheren Abstraktionsebene angesiedelt als der *Interrupt-Server*.

Generell widerspricht das Modell eines *Interrupt-Handlers*, der auf blockierendes *Message-Passing* zur Kommunikation mit seinem *Interrupt-Server* basiert, somit der Forderung, daß die Laufzeiten eines *Interrupt-Handlers* kurz und der *Interrupt-Handler* selbst elementar aufgebaut sein sollte. Nur der Entwurf eines reentranten Interrupt-Systems würde helfen, die Antwortzeiten für die *Interrupt-Handler* zu minimieren. Die Komplexität eines solchen Systems würde hierbei jedoch weiter zunehmen – insbesondere verstärkt mit der Gefahr von *Deadlocks* behaftet sein –, so daß dieser Ansatz keine generelle Lösung der skizzierten Problematik darstellen kann. Das Grundprinzip zum Entwurf eines Ein-/Ausgabesystems, in [Loehr 1980] exemplarisch veranschaulicht, lautet damit, einen *Interrupt-Handler* funktional sehr einfach zu gestalten.

3.8.3.1.3. Trennung zwischen Kontroll- und Datenfluß

Aufgrund der skizzierten technischen und konzeptionellen Aspekte, kann die Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* nur über nichtblockierende Primitiven erfolgen, die insbesondere keinen Datentransfer unterstützen. Der beidseitige Datenaustausch zwischen *Interrupt-Handler* und *Interrupt-Server* muß dadurch ermöglicht werden, daß beide Komponenten Zugriff auf denselben Adreßraum besitzen und somit auf gemeinsame Daten zugreifen können. Daraus ergibt sich eine "natürliche" Trennung zwischen Primitiven, die den Kontrollfluß und solchen, die den Datenfluß bei der Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* regeln.

Unter MOOSE wird die Kommunikation mit dem *Interrupt-Server* zum Zeitpunkt der Termination des *Interrupt-Handlers* ermöglicht. Der *Interrupt-Handler* wird über den *Gate-Handler* des Kernels gestartet und kehrt bei seiner Termination zu dem *Gate-Handler* zurück. Mit der Termination des *Interrupt-Handlers* ist ein Resultat verknüpft, das die nachfolgenden Aktionen des *Gate-Handlers* steuert. Dieses Resultat bestimmt, ob dem *Interrupt-Server* ein *Interrupt-Signal* zuzustellen ist oder direkt die Ausführung des von dem Interrupt unterbrochenen Prozesses wieder aufgenommen werden soll.

Die Benachrichtigung des *Interrupt-Servers* basiert auf zwei Varianten, die jeweils die Übermittlung eines Signals bedeuten. Diese Varianten unterscheiden sich in der Identifikation des Kommunikationspunktes (*Ports*), über den der *Interrupt-Server* erreichbar ist. Diese Identifikation kann implizit erfolgen, in diesem Fall wird der mit dem jeweiligen *Gate* assoziierte *Port* betrachtet, oder sie kann explizit vorgegeben werden, in diesem Fall bestimmt der *Interrupt-Handler* über welchen *Port* ein *Interrupt-Signal* übermittelt werden soll. Verlangt somit der *Interrupt-Handler* bei seiner Termination die Benachrichtigung des *Interrupt-Servers*, wird dies durch den entsprechenden *Gate-Handler* über synchronisierte Kommunikationsprimitiven erreicht, die zur Übermittlung von Signalen führen.

3.8.3.2. Synchronisation der Kommunikationsprimitiven

Der zentrale Punkt bei der Propagation von Interrupts durch den MOOSE-Kernel stellt, neben der Durchführung der Kommunikation mit den *Server*-Prozessen, die Synchronisation aller Kommunikationsaktivitäten dar. Die im Zusammenhang mit der Behandlung von Interrupts auftretende systemglobale Synchronisationsproblematik ist aus den *Server*-Prozessen herausgezogen und im MOOSE-Kernel konzentriert. Die tatsächliche Behandlung der Interrupts, d.h. die dazu notwendigen Operationen im Zusammenhang mit den Geräten, findet im Kontext der *Server*-Prozesse statt. Der Kernel propagiert die Interrupts nur, behält aber die Kontrolle über die Kommunikationsaktivitäten bei der Behandlung der Interrupts.

Dies stellt den wesentlichen Unterschied zu prozedurorientierten Betriebssystemen dar. In diesen Systemen sind die Komponenten zur Interrupt-Behandlung (in MOOSE als *Server*-Prozesse realisiert) in den Kernel integriert und haben somit die Möglichkeit der direkten Manipulation des Kernels. Diese Manipulation äußert sich z.B. in UNIX darin, daß die entsprechenden Komponenten direkt die Synchronisationsoperationen des Kernels anwenden, in jedem Fall aber auf zentrale Datenstrukturen des Kernels zugreifen können. Die Konsequenz daraus ist die Forderung nach zusätzlichen Leistungen bei dem Entwurf, der Verifikation und der Realisierung solcher Komponenten, um fehlerhaft operierende *Interrupt-Handler* ausschließen zu können.

Durch die in MOOSE realisierte technische Entkopplung von Kernel und *Interrupt-Handler* wird die Verifikation des Gesamtsystems erheblich vereinfacht. Dies gilt umsomehr bei der Realisierung der *Interrupt-Handler* durch *Gate-Server*, da hiermit in jedem Fall eine eigene Identifikation und ein vom Kernel vollständig entkoppelter Adreßraum erlangt werden kann. In jedem Fall abstrahiert der *Interrupt-Handler* von den Prinzipien der Synchronisationsmaßnahmen des Kernels und wird somit nicht mit der Problematik der Synchronisation und Verifikation nebenläufiger Aktivitäten konfrontiert.

3.8.3.2.1. Klassisches Monitor-Konzept

Um die Kommunikationsprimitiven für ihre häufige Anwendung so effizient wie möglich zu gestalten, wird der gesamte MOOSE-Kernel als kritischer Abschnitt betrachtet. Der MOOSE-Kernel entspricht in dieser Hinsicht einem Monitor im Sinne von [Hoare 1974]. Da die Laufzeiten innerhalb des Kernels deterministisch sind und eine obere Grenze festgelegt werden kann[35], ist diese Lösung für einen elementaren *Message-Passing* Kernel äußerst günstig. Die Konsequenz davon ist jedoch, daß zur Kommunikation mit ihren *Server*-Prozessen den *Interrupt-Handlern* die Anwendung der entsprechenden Primitiven des Kernels untersagt ist, da innerhalb des Kernels keine Synchronisation der Kommunikationsprimitiven stattfindet.

Die kritische Situation bei der Durchführung der Kommunikation ist dann gegeben, wenn ein Kernelprozeß von dem Interrupt unterbrochen worden ist. Die Synchronisation findet in diesem Zusammenhang über eine spezielle Semaphore statt. Die Besonderheit an dieser

[35] Die Ausnahme bilden zyklische Operationen, die z.B. bei der Realisierung des Datentransfers zum Nachrichtenaustausch ihre Anwendung finden. Da jedoch solche Primitiven innerhalb des MOOSE-Kernels nicht als kritisch betrachtet werden, wird während ihrer Ausführung der kritische Abschnitt verlassen und erst dann wieder betreten, wenn die entsprechende Operation beendet worden ist.

Semaphore ist jedoch, daß der Kernelprozeß, unter dessen Identifikation der *Gate-Handler* aktiv ist, nicht blockieren darf. Dies wird dadurch erreicht, daß der an der Kommunikation beteiligte *Event-Port* und nicht der von dem Interrupt unterbrochene Prozeß durch die Semaphore kontrolliert wird.

3.8.3.2.2. Synchronisation auf der Kernelebene

Wird ein Kernelprozeß von einem Interrupt unterbrochen und fordert der *Interrupt-Handler* bei seiner Termination die Kommunikation mit seinem *Interrupt-Server* an, so wird der dem Interrupt zugeordnete *Event-Port* in eine *Event-List* eingetragen. Der jeweilige *Event-Port* ist jedoch nur einmal in der *Event-List* enthalten. Wie oft ein Interrupt über diesen *Event-Port* signalisiert worden ist, wird über einen *Event-Count* in dem jeweiligen *Event-Port* vermerkt.

Mit der Termination eines Kernelprozesses, d.h. bevor von der Kernelebene zur System- oder Benutzerebene zurückgekehrt wird, erfolgt die Abarbeitung der *Event-List*. Damit werden die mit den jeweiligen *Event-Ports* assoziierten *Interrupt-Server* nachträglich von der Termination ihrer *Interrupt-Handler* informiert. Die in dem jeweiligen *Event-Port* vermerkten *Interrupt-Signals*, angegeben durch den *Event-Count*, werden dem *Interrupt-Server* zugestellt.

Alle Aktivitäten, die auf der Kernelebene stattfinden –sei es ein *Kernel-Call*, die Behandlung/Propagation von Interrupts oder die Propagation von Traps–, müssen über *Gate-Handler* gestartet worden sein. Demzufolge sind die *Gate-Handler* prädestiniert dafür, den Auf- und Abbau der *Event-List* zu kontrollieren. Auf diese Weise kann für den MOOSE-Kernel eine ausgezeichnete Stelle angegeben werden, an der die Synchronisationsproblematik konzentriert ist. Dies ist eine wesentliche Voraussetzung zur Durchführung eines Korrektheitsnachweises für den Kernel.

Zur Verwaltung der *Event-List* ist es notwendig, darüber Informationen zu besitzen, ob ein Kernelprozeß unterbrochen worden ist. Jede Aktivierung eines Kernelprozesses durch einen Trap oder Interrupt wird von den jeweiligen *Gate-Handlern* vermerkt. Hierzu findet ein *Lock-Count* seine Anwendung. Bei jedem Eintritt in den Kernel wird der *Lock-Count* inkrementiert, und jeder Austritt aus den Kernel führt zur Dekrementierung des *Lock-Counts*. Nur bei einem bestimmten Wert des *Lock-Counts* ist die Kommunikation des *Interrupt-Handlers* mit seinem *Interrupt-Server* direkt möglich, ansonsten muß dieser Kommunikationswusch zurückgestellt und über die *Event-List* vermerkt werden.

Um zur Verwaltung der *Event-List* ein elementares und laufzeiteffizientes Verfahren angeben zu können, ist der *Lock-Count* mit -2 vorinitialisiert. Die Aktivierung eines Kernelprozesses führt dazu, daß *Lock-Count* den Wert -1 annimt und damit anzeigt, daß ein Kernelprozeß aktiv ist. Wird der Kernelprozeß durch einen Interrupt unterbrochen, führt dies dazu, daß *Lock-Count* von dem entsprechenden *Gate-Handler* abermals inkrementiert wird und damit einen positiven Wert annimt.

Positive Werte von *Lock-Count* bedeuten, daß ein Kernelprozeß durch einen Interrupt unterbrochen worden ist und daß die Kommunikation zwischen dem *Interrupt-Handler* und dem *Interrupt-Server* zurückgestellt werden muß. Der entsprechende *Event-Port* wird in die *Event-List* engetragen. Der negative Wert (-1) von *Lock-Count* bedeutet, daß kein Kernelprozeß, sondern ein Benutzer-/Systemprozeß durch einen Interrupt unterbrochen worden

ist. Die Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server* kann direkt stattfinden.

Die Verwaltung von *Lock-Count* basiert auf drei elementaren Operationen. Diese Operationen bedeuten das Inkrementieren, Dekrementieren und Abfragen von *Lock-Count*. Hierzu stellt jeder reale Prozessor unteilbare Operationen zur Verfügung, zumindest, wenn Monoprozessorarchitekturen betrachtet werden. Da der MOOSE-Kernel in einem Multiprozessorsystem jedoch jeweils auf jeder Prozessorkomponente angesiedelt sein wird, ist auch in diesen Systemen die Unteilbarkeit dieser drei Operationen sichergestellt. Es ist hierbei zu beachten, daß *Lock-Count* jeweils lokal in jedem Kernel verwaltet wird.

3.8.3.2.3. Synchronisation auf der physikalischen Ebene

Die im MOOSE-Kernel verwaltete Semaphore zur Synchronisation der Kommunikationsaktivitäten zwischen *Interrupt-Handler* und *Interrupt-Server*, wird in erster Linie durch die *Event-List* repräsentiert. Da auf diese Liste jedoch immer im Zusammenhang mit dem Abschluß einer Interrupt-Behandlung zugegriffen wird, müssen die jeweiligen Zugriffsoperationen selbst synchronisiert werden. Hierzu werden, während der kritischen Operationsfolgen auf diese Liste, die Interrupts durch den realen Prozessor unterbunden. Dies stellt eine von zwei Stellen des MOOSE-Kernels dar, an der Interrupts physikalisch nicht zugelassen sind. Die andere Stelle betrifft die Prozeßumschaltung.

Um die Phase, während der Interrupts physikalisch unterbunden sind, sehr kurz zu fassen, ist die Verwaltung der *Event-List* selbst sehr elementar. Die kritischen Operationsfolgen auf diese Liste setzen sich jeweils aus sehr wenigen Schritten zusammen und führen zu einer extrem kurzen Dauer der physikalischen Unterbindung von Interrupts durch den realen Prozessor. Die kritischen Operationsfolgen sind im einzelnen:

- Aufnahme eines *Event-Ports* in die *Event-List*, sofern dieser *Port* noch nicht in der *Event-List* enthalten ist. Der *Event-Port* wird jeweils an das Ende der *Event-List* angehangen;
- Übertrag des *Event-Counts* auf den *Signal-Count* des jeweiligen *Event-Ports*. Der *Signal-Count* wird um *Event-Count* erhöht und *Event-Count* wird anschließend auf 0 gesetzt;
- Austragen eines *Event-Ports* aus der *Event-List*. Dies erfolgt immer im Anschluß an die Propagation der in dem *Event-Port* vermerkten *Interrupt-Signals*. Ausgetragen wird jeweils der *Event-Port* am Kopf der *Event-List*. Die Abarbeitungsreihenfolge der *Event-List* erfolgt demnach entsprechend des *first-come-first-serve* Prinzips.

3.8.3.2.4. Konsequenz des Monitor-Konzeptes für den Kernel

Das Monitor-Konzept, zur Synchronisation der Kommunikation zwischen *Interrupt-Handler* und *Interrupt-Server*, ist im MOOSE-Kernel sehr effizient realisiert. Im einzelnen lassen sich hierdurch folgende Konsequenzen für den Kernel und die Systemprozesse abzeichen:

- die sehr häufig angewendeten Primitiven zur Interprozeßkommunikation sind nicht mit zusätzlichen Maßnahmen zur Interrupt-Synchronisation verbunden. Damit wird eine

sehr laufzeiteffiziente Realisierung der Mechanismen zur Interprozeβkommunikation unter MOOSE ermöglicht;

- die Synchronisation wird vollständig über die jeweiligen *Gate-Handler* gesteuert. Alle *Gate-Handler* verwalten den *Lock-Count*, jedoch nur die *Gate-Handler* für Interrupts, müssen die Synchronisation der Kommunikationsaktivitäten gewährleisten;
- unabhängig davon, ob ein Kernelprozeβ aktiv ist oder nicht aktiv ist, wird die Behandlung von Interrupts, bis auf sehr kurze Zeitabschnitte, immer möglich sein. Auf Grundlage der gegenwärtigen Mikroprozessortechnologie ergeben sich hierbei Laufzeiten, während der die Interrupts unterbunden sind, die ca. zwischen 10 und 20 Mikrosekunden liegen;
- die Realisierung der Synchronisation von Kommunikationsaktivitäten ist nicht über den gesamten MOOSE-Kernel verteilt, sondern durch die *Gate-Handler* konzentriert. Insbesondere sind die *Interrupt-Handler* nicht mit der Synchronisationsproblematik im Zusammenhang der Kommunikation mit ihren *Interrupt-Servern* behaftet. Einzig die problemorientierte Synchronisation, im Zusammenhang mit der Verwaltung von Geräten, bleibt bestehen.

Im Vergleich zu anderen *Message-Passing* Systemen, z.B. THOTH [Cheriton 1982] und QNX[36] [Quantum 1984], begründen die eben skizzierten Konsequenzen erhebliche Leistungsunterschiede zugunsten des MOOSE-Kernels. Die Leistungsunterschiede sind hierbei nicht nur im Zusammenhang mit den Mechanismen zur Interprozeβkommunikation zu sehen. Sie ergeben sich insbesondere aufgrund des im MOOSE-Kernel realisierten Verfahrens zur Interrupt-Synchronisation.

3.9. Prozeβattribute

Die Attribute eines Prozesses setzen sich aus prozeβ- und *team*-lokalen Attributen zusammen. Die *team*-lokalen Attribute besitzen ihre Gültigkeit für alle Prozesse desselben *Teams*, wohingegen die prozeβlokalen Attribute jeweils nur einem Prozeβ zugeordnet sind.

Die Zuordnung und Interpretation der Attribute eines Prozesses findet jeweils auf der Kernel- und Systemebene statt. Entsprechend der hierarchischen Ordnung dieser Ebenen – die Systemebene benutzt die Kernelebene – ist der MOOSE-Kernel jedoch unabhängig von der durch die Systemprozesse verwalteten Prozeβattribute. Das Zuordnen bestimmter Attribute zu einem Prozeβ steuert allgemein die Ausführung des entsprechenden Prozesses durch den Kernel.

3.9.1. Attributverwaltung auf der Systemebene

Die auf der Systemebene erreichte Zuordnung von Attributen zu Prozessen ist immer im Zusammenhang mit der Modellierung des Adreβraumes eines Prozesses bzw. *Teams* zu sehen. Hierbei ergeben sich im einzelnen folgende Attribute:

[36] QNX ist ein Warenzeichen der *Quantum Software Systems Ltd*.

custodian der entsprechende Prozeβ ist als kontrollierender Prozeβ eines *Teams* erzeugt worden. Der durch ihn definierte Adreβraum stellt den Bezugsrahmen zum Aufbau eines *Teams* dar;

captive der entsprechende Prozeβ ist als unterstützender Prozeβ eines *Teams* erzeugt worden und besitzt direkten Zugriff auf das Code- und Datensegment eines *custodian*;

enclosed für den entsprechenden Prozeβ gilt eine bestimmte relative Adreβraumbeziehung;

swapped das entsprechende *Team* ist auf einem Hintergrundspeicher ausgelagert ("*geswapped*");

locked das entsprechende *Team* ist fest einem bestimmten Speicherbereich zugeordnet und darf nicht verschoben bzw. "*geswapped*" werden.

Bedeutende Gesichtspunkte ergeben sich hierbei im Zusammenhang der Kombination der Attribute *enclosed* und *custodian* bzw. *captive*. Ein *enclosed custodian* repräsentiert ein *Team*, dessen Datensegment physikalisch einer bestimmte Gruppe von Datensegmenten anderer *Teams* zugeordnet ist. Damit soll sichergestellt werden, daβ Applikationen, die aus mehreren *Teams* aufgebaut sind, von ihrer Speicherabbildung her gesehen, in einem Block verwaltet werden können. Hierbei wird insbesondere das "*Swappen*" von Applikationen unterstützt.

Ein *enclosed captive* repräsentiert einen Prozeβ innerhalb eines *Teams*, dessen Stacksegment einem logischen Adreβbereich zugeordnet ist, der von dem seines erzeugenden Prozesses unterschiedlich ist. Dieser logische Adreβbereich wird nicht zwischen verschiedenen physikalischen Adreβbereichen gemultiplext. Damit ist sichergestellt, daβ das *Team*-Konzept, auch ohne besondere Anforderungen an einen realen Prozessor, der keine Abbildung von logischen auf realen Adressen unterstützt, stellen zu müssen, realisiert werden kann.

3.9.2. Attributverwaltung auf der Kernelebene

Die durch den MOOSE-Kernel erreichte Zuordnung von Attributen zu Prozessen ist immer im Zusammenhang mit der Kontrolle der Ausführung eines Prozesses bzw. *Teams* zu sehen. Hierbei ergeben sich im einzelnen folgende Attribute:

server der entsprechende Prozeβ ist ein *Broadcast-Server*. Die *Broadcast*-Attribute dieses Prozesses geben an, von welchen Systemereignissen der Prozeβ über einen *Broadcast* benachrichtigt werden soll;

client der entsprechende Prozeβ repräsentiert einen *Broadcast-Client*. Seine *Broadcast*-Attribute definieren das durch einen *Broadcast* zu verbreitende Systemereignis für den betreffenden Prozeβ;

indivisible der entsprechende Prozeβ (und damit sein *Team*) wird die Kontrolle nicht durch einen *Clock-Interrupt* abgeben. Der Prozeβ ist unteilbar;

interrupt der entsprechende Prozeβ wird bei seiner Deblockierung an den Kopf der *team*-lokalen *Ready-List* plaziert. Dieser Prozeβ wird als *Interrupt-Task* bezeichnet;

emulated der entsprechende Prozeβ wird emuliert und ist einer bestimmten Betriebssystemdomäne zugeordnet;

system das entsprechende *Team* repräsentiert einen Systemprozeβ. Bestimmte privilegierte Operationen des MOOSE-Kernels sind erlaubt;

privileged das entsprechende *Team* ist privilegiert. Sämtliche Operationen des MOOSE-Kernels sind erlaubt;

excluded das entsprechende *Team* ist aus dem *Dispatch-Set* ausgeschlossen;

informed das entsprechende *Team* soll bei der Verarbeitung von *Broadcasts* mit berücksichtigt werden.

Insbesondere mit der Verarbeitung von *Broadcasts* besitzen die jeweiligen Attribute eines Prozesses eine zentrale Bedeutung. Zur Auswahl eines *Broadcast-Servers* orientiert sich der MOOSE-Kernel in erster Linie nach einem *informed custodian*. In dem von diesen Prozeβ kontrollierten *Team* (evtl. der *custodian* selbst) werden alle *server* betrachtet, die mit dem jeweils aktiven *Broadcast* attributiert sind. Dem ersten Prozeβ dieses *Teams* wird der jeweilige *Broadcast* zugestellt.

Broadcasts stehen immer in Verbindung mit Prozessen, dem *Broadcast-Client* und *Broadcast-Server*. Zur Spezifikation, welcher *Broadcast* für einen *Broadcast-Client* aktiv ist, werden die Prozeβattribute des *Broadcast-Clients* entsprechend aufgesetzt. Zur Selektion eines entsprechenden *Broadcast-Servers* werden die Prozeβattribute des *Servers* eines *informed custodian* betrachtet. Die entsprechenden Prozeβattribute zur Spezifikation des *Broadcasts* stellen die *Broadcast*-Attribute des jeweiligen Prozesses dar. Hierbei unterscheidet der MOOSE-Kernel vier verschiedene *Broadcast*-Attribute, die repräsentativ für die einzelnen *Broadcast*-Klassen stehen. Zur Selektion eines *Broadcast-Servers* müssen seine *Broadcast*-Attribute mit denen des jeweiligen *Broadcast-Clients* übereinstimmen.

3.9.3. Manipulation und Bedeutung der Attribute eines Prozesses

Die Attribute eines Prozesses beeinflussen zum groβen Teil die Laufzeiteigenschaften und Zugriffsrechte des jeweiligen Prozesses. Demzufolge ist die Modifikation bestimmter Attribute nur dann möglich, wenn der modifizierende Prozeβ über entsprechende Zugriffsrechte verfügt. Die nachfolgende Aufzählung gibt an, welche Klassen von Prozessen die Modifikation bestimmter Prozeβattribute erreichen können:

- Benutzerprozesse
  die Attribute *emulated* und *interrupt*. Desweiteren das Attribut *custodian*, jedoch nur von einem *custodian* selbst;
- Systemprozesse
  die Attribute *system*, *server*, *client*, *informed*, *excluded* und die *Broadcast*-Attribute eines Prozesses, sowie alle Attribute, die Benutzerprozesse manipulieren dürfen;
- privilegierte Prozesse
  die Attribute *privileged* und *indivisible*, sowie alle Attribute, die Systemprozesse manipulieren dürfen.

Die Attribute eines Prozesses ermöglichen es, Zugriffsrechte des jeweiligen Prozesses auf Objekte, die vom MOOSE-Kernel verwaltet werden, zu kontrollieren. Diese Objekte stellen jeweils *Ports*, *Gates* und den realen Prozessor (die CPU) dar.

Um vordefinierte *Ports* mit dem *specific attach* zu belegen, muß der jeweilige Prozeß mindestens als Systemprozeß oder als privilegierter Prozeß ausgezeichnet sein. Um *Gates* zu belegen, d.h. um die Anbindung an Trap-/Interrupt-Vektoren des realen Prozessors zu erreichen, muß der jeweilige Prozeß privilegiert sein. Die gleiche Anforderung gilt im Zusammenhang mit der exklusiven Belegung der CPU durch einen Prozeß, d.h. daß der jeweilige Prozeß unteilbar ausgeführt werden soll. Auch in diesem Fall muß sich der Prozeß als privilegiert auszeichnen können.

Der kontrollierende Prozeß eines *Teams* ist jeweils mit *custodian* attributiert. In einem *Team* wird es deshalb nur genau einen *custodian* geben. Die Erzeugung neuer *Teams* ist allen oben genannten Klassen von Prozessen möglich, sofern der erzeugende Prozeß selbst als *custodian* ausgezeichnet ist.

Kapitel 4

Prozesse und Dienste

Die im vorigen Kapitel vorgestellten Mechanismen des MOOSE-Kernels dienen im wesentlichen dazu, eine Kommunikation und Kooperation zwischen Prozessen zu ermöglichen. Dabei wurde davon ausgegangen, daß Prozesse bereits zur Verfügung stehen und damit vom MOOSE-Kernel zur Ausführung auf einen realen Prozessor abgebildet werden können.

In diesem Kapitel steht die Modellierung von Prozessen im Vordergrund. Es wird dargestellt, nach welchen Prinzipien Prozesse erzeugt werden und über welche Mechanismen eine Abstraktion von den Prozessen zu erreichen ist. Hierbei wird zur Abstraktion von Prozessen der Dienstbegriff zugrunde gelegt. Mit den hier vorgestellten Prinzipien wird, neben dem MOOSE-Kernel selbst, die Grundlage angegeben, die zum Aufbau einer Betriebssystemfamilie für MOOSE von Bedeutung ist.

4.1. Prozeßmodelle

Zur Ausführung von Prozessen, ist durch den MOOSE-Kernel für jeden Prozeß eine Abbildung auf den realen Prozessor zu realisieren. Diese Abbildung wird durch bestimmte vorgegebene Datenstrukturen für jeden Prozeß definiert. Der Kernel interpretiert diese Datenstrukturen dahingehend, daß mit jeder Aktivierung eines Prozesses eine Adreßraumzuordnung für den betreffenden Prozeß getroffen wird.

Die Definition dieser Datenstrukturen erfolgt nicht durch den Kernel, sondern wird durch bestimmte Systemprozesse eines MOOSE-Betriebssystems erreicht. Diese Systemprozesse bestimmen damit das jeweils durch das entsprechende Betriebssystem zugrunde gelegte Prozeßmodell. Daß z.B. mehreren Prozessen derselbe Adreßraum zugeordnet ist und sie damit ein *Team* bilden, wird durch den MOOSE-Kernel nicht festgelegt. Einzig die logische Sichtweise des MOOSE-Kernels über die Prozesse, bedingt das *Team*-Konzept. Der *Context*- und *Image-Descriptor* ist jeweils so aufgebaut, daß aus diesen Datenstrukturen nicht hervorgeht, ob einem Prozeß exklusiv ein bestimmter Adreßraum zugeordnet ist, oder ob sich mehrere Prozesse denselben Adreßraum teilen ("*sharen*").

4.1.1. Zuordnung von Adreßräumen

Das jeweils zugrunde liegende Prozeßmodell ist wesentlich für die Funktionalität eines Systems. Es gibt aber auch gleichzeitig das Einsatzspektrum der Systeme vor, die auf einem bestimmten Prozeßmodell basieren. Wenn z.B. keine Abbildungsmöglichkeit für die Prozesse auf den jeweiligen realen Prozessor gegeben ist – weil zur Realisierung des Prozeßmodells bestimmte hardware-technischen Voraussetzungen erfüllt sein müssen –, wird das Einsatzgebiet

der durch die Prozesse konstituierten Applikationen eingeschränkt. Auch wenn ein entsprechender realer Prozessor geschaffen wird, der die Abbildung der Prozesse ermöglicht, bedingt dies nicht notwendigerweise die Erweiterung des Einsatzgebietes der jeweiligen Applikation. Die Kosten und Verfügbarkeit des entsprechenden Prozessors werden hierbei jeweils eine entscheidende Rolle spielen.

Das Grundprinzip bei MOOSE ist es daher, Prozeßmodelle zur Verfügung zu stellen, die keine besonderen Anforderungen an einen bestimmten realen Prozessor stellen. Das bedeutet insbesondere, die Grundlage für ein prozeßorientiertes Betriebssystems zu schaffen, dessen Einsatzspektrum die für UNIX bestehende Lücke zwischen Ein- und Mehrbenutzersystemen schließen soll. Um dieses Grundprinzip zu verwirklichen, werden zwei Verfahren angewendet, um Adreßräume für die zu erzeugenden Prozesse herleiten zu können. Diese Verfahren unterscheiden sich darin, ob Adreßräume als Duplikate anderer Adreßräume implizit oder ob sie durch eine Vorintialisierung explizit entstehen.

4.1.1.1. Duplikation von Kontexten

Mit UNIX ist eng eine bestimmte Form der Prozeßerzeugung verbunden. Neue Prozesse werden bei UNIX erzeugt, indem eine vollständige Kopie des erzeugenden Prozesses angelegt wird. Dupliziert wird hierbei das Daten- und Stacksegment des erzeugenden Prozesses. Ein evtl. vorhandenenes Codesegment (in UNIX als Textsegment bezeichnet) wird nicht dupliziert, sondern zwischen dem erzeugenden und dem erzeugten Prozeß geteilt. Dupliziert werden somit immer die Bereiche, bei denen davon ausgegangen wird, daß sie während der Ausführung eines Prozesses modifiziert werden.

Die unter UNIX einzige Form der Prozeßerzeugung wird unter MOOSE als eine besondere Variante betrachtet. Diese Form bietet verschiedene Vorteile, die sich im wesentlichen daraus ergeben, daß der erzeugte Prozeß den jeweiligen Laufzeitkontext seines erzeugenden Prozesses übertragen bekommen hat.

Die Duplikation des Laufzeitkontextes bedeutet, daß der erzeugte Prozeß seine Ausführung genau an der Stelle beginnt, an der seine Erzeugung initiiert worden ist. Ab dieser Stelle durchlaufen somit zwei verschiedene Prozesse – der erzeugende und der erzeugte Prozeß – den gleichen Kontext (siehe auch [Lions 1977] und [Kernighan, McIlroy 1979]). Es wird jedoch dafür Sorge getragen, daß sich der erzeugende und der erzeugte Prozeß jeweils als solche identifizieren können und dadurch mit der Ausführung unterschiedlicher Programmbereiche fortfahren können. Bildlich gesprochen entspricht die Stelle, an der ein neuer Prozeß erzeugt wird, der Gabelung (fork) eines Prozesses.

Zum Vorteil dieses Verfahrens sind im wesentlichen zwei Aspekte anzuführen. Zum einen ist die Übergabe von Argumenten an den erzeugten Prozeß prinzipiell keinen Einschränkungen unterworfen. Der erzeugte Prozeß besitzt schließlich eine Kopie sämtlicher Datenbestände seines erzeugenden Prozesses. Zum anderen können in einfacher Weise Applikationen unterstützt werden, die in bestimmten Intervallen die Sicherung ihrer Datenbestände erfordern. Ein Prozeß bestimmt mit der Erzeugung eines neuen Prozesses den jeweiligen *Checkpoint* für die durch den erzeugenden Prozeß kontrollierte Applikation. Die Aufgabe des erzeugten Prozesses besteht dann darin, die zum Zeitpunkt des *Checkpoints* (die Erzeugung des Prozesses) gültigen Datenbestände zu sichern. Diese Maßnahme erfolgt nebenläufig zum

weiteren Ablauf der eigentlichen Applikation und ist insbesondere mit keinen Synchronisationsproblemen behaftet.

4.1.1.2. Erzeugung neuer Kontexte

Das Verfahren der Duplikation eines Adreβraumes, um einem zu erzeugenden Prozeβ einen Adreβraum implizit zuordnen zu können, ist nur durchführbar, wenn eine wesentliche Voraussetzung erfüllt ist. Diese Voraussetzung bedeutet die Forderung an einen realen Prozessor, gleiche logische Adressen eines Prozesses auf unterschiedliche reale Adressen während der Ausführung eines Prozesses abzubilden.

Für die Abbildung gleicher logischer Adressen auf unterschiedliche reale Adressen ergeben sich prinzipiell zwei Ansätze. Der erste Ansatz basiert auf *Relocation-Hardware* (z.B. eine MMU oder Segmentregister). Der zweite Ansatz erfordert von dem realen Prozessor, den jeweiligen Prozeβ positionsunabhängig, hinsichtlich eines bestimmten realen Adreβraumes, ausführen zu können. Dieser Ansatz wird insbesondere dann notwendig sein, wenn keine *Relocation-Hardware* zur Verfügung steht. In diesem Zusammenhang gilt die zusätzliche Forderung nach einem Sprachverarbeitungssystem, das in der Lage ist, positionsunabhängigen Code zu erzeugen. Im Zusammenhang mit diesem zuletzt genannten Aspekt wird insbesondere durch OS-9 [Microware 1984] gezeigt, wie auch ohne spezielle Hardware-Anforderungen das Prozeβmodell von UNIX realisiert werden kann.

Zur Realisierung von *Teams*, auf der Grundlage der Duplikation von Stacksegmenten für die innerhalb des *Teams* angesiedelten Prozesse, ist jedoch in jedem Fall *Relocation-Hardware* von dem realen Prozessor zu fordern. Die Duplikation eines Stacksegmentes innerhalb eines *Teams* bedeutet, daβ gleiche logische Adressen von Objekten innerhalb des Stacksegmentes auf verschiedene physikalische Adressen abgebildet werden müssen. In diesem Fall ermöglicht auch nicht die positionsunabhängige Ausführung von Prozessen das *fork*-Prinzip. Die Positionsunabhängigkeit kann nur durch Mechanismen des Prozessors zur relativen Adressierung von Objekten erreicht werden. Objekte duplizierter Stacksegmente besitzen jedoch immer die gleichen relativen (logischen) Adressen. Demzufolge müssen in diesem Fall die einzelnen Stacksegmente jeweils auf verschiedene physikalische Adreβbereiche abgebildet werden.

Um insbesondere *Teams* –und damit Prozesse allgemein– auch auf sehr einfacher Hardware unterstützen zu können, muβ neben der Duplikation des Laufzeitkontextes noch eine andere Variante der Prozeβerzeugung zur Verfügung stehen. Diese Variante muβ davon ausgehen, daβ keine bestehenden Laufzeitkontexte für den zu erzeugenden Prozeβ referenziert werden können. Vielmehr muβ der erzeugte Prozeβ, nach seiner erstmaligen Aktivierung, selbst dafür sorgen, sich einen bestimmten Laufzeitkontext aufzubauen. Entsprechend dieser Variante zur Erzeugung neuer Laufzeitkontexte, wird zwischen der Erzeugung eines neuen *Teams* (eines *custodian*) und eines neuen Prozesses (dem *captive*) innerhalb eines bestehenden *Teams* unterschieden.

Zur Erzeugung eines *captive* kann davon ausgegangen werden, daβ bereits ein Adreβraum, d.h. Code- und Datensegmente, verfügbar ist. Zur Angabe der Stelle, an der der *captive* mit seiner Ausführung beginnen soll, eignet sich, aufgrund des zur Verfügung stehenden Codesegmentes, die Adresse einer Prozedur. Diese Prozedur kann parameterisiert sein, so daβ

die Erzeugung des *captive* mit der Übergabe von Argumenten durch den erzeugenden Prozeß verbunden sein kann.

Die Erzeugung eines *custodian* dagegen bedeutet, daß auf keinen bestehenden Adreßraum zurückgegriffen werden kann. Die entsprechenden Code- und Datensegmente für den *custodian* müssen erst verfügbar gemacht werden. Hierzu bietet es sich an, ein neues Programm, das über ein Filesystem geladen werden kann, zur Ausführung durch einen neuen Prozeß zu bringen.

Dieses Verfahren der Erzeugung eines neuen Prozesses, verbunden mit der Ausführung eines neuen Progammes, ermöglicht die Verschmelzung der unter UNIX üblichen Vorgehensweise zur Erzeugung von Applikationen. Unter UNIX sind hierbei immer im wesentlichen zwei Schritte notwendig; erstens die Erzeugung eines neuen Prozesses nach dem *fork*-Prinzip und zweitens die Überlagerung des Adreßraumes des neuen Prozesses mit einem neuen Programm.

Unter MOOSE ist dieses Verfahren ebenfalls möglich – da ebenfalls das *fork*-Prinzip zur Erzeugung von Prozessen unterztützt wird. Die Verschmelzung der in UNIX üblichen Schritte zur Ausführung eines Programms durch einen neuen Prozeß, bringt jedoch zwei Vorteile mit sich. Zum einen ist es mit dieser Technik möglich, auch Prozesse erzeugen zu können, ohne spezielle Anforderungen an den realen Prozessor zur Ausführung des Prozesses zu stellen (*Relocation-Hardware* oder die Möglichkeit der positionsunabhängigen Abarbeitung eines Prozesses). Zum anderen ergibt sich zum Starten neuer Applikationsprogramme ein nicht zu unterschätzender Geschwindigkeitsvorteil: das Duplizieren des Daten- und Stacksegmentes entfällt. Dieser Aspekt führte insbesondere bei UNIX 4.2BSD dazu, eine überarbeitete Form der Erzeugung eines Prozesses einzuführen, die das Duplizieren des Daten-/Stacksegmentes des erzeugenden Prozesses vermeidet (zum *virtual fork* siehe [Joy et al. 1983]). Ein anderes Beispiel stellt der *run System-Call* von LOCUS [Popek et al. 1981] dar.

4.1.2. Aufbau eines Teams

Ein *Team* faßt mehrere Prozesse innerhalb desselben Adreßraumes zusammen. Hierzu stellt der entsprechende Adreßraum die notwendige Voraussetzung (den Bezugsrahmen) dar, ein *Team* aufbauen zu können. Mit den dargestellten Modellen zur Erzeugung eines neuen Prozesses, ist dazu der entsprechende Grundstein gelegt. Der Aufbau eines *Teams* ist demnach in zwei Schritte unterteilt:

a) Die Definition des Adreßraumes für das *Team*. Dies wird durch die Erzeugung eines Prozesses erreicht, der das Duplikat seines erzeugenden Prozesses darstellt bzw. dem ein vollständig neuer Laufzeitkontext zugeordnet worden ist. Der erzeugte Prozeß stellt den Bezugsrahmen für den Adreßraum eines *Teams* her und ist gleichzeitig der *custodian* des betreffenden *Teams*.

b) Die Zuordnung weiterer Prozesse (*captives*) zu einem *Team*. Dies wird durch die Erzeugung entsprechender Prozesse erreicht, denen einzig ein Stacksegment zugeordnet wird. Das Stacksegment ist entweder das Duplikat des Stacksegmentes vom erzeugenden Prozeß oder es wird vom erzeugenden Prozeß explizit vorinitialisiert.

Der wesentliche Unterschied hinsichtlich der Adreβraumzuordnung zwischen *custodian* und *captive* besteht somit darin, daβ zur Erzeugung eines *captives* nur ein neues Stacksegment eingerichtet werden muβ. Da mit der Duplikation dieses Segmentes der aktuelle Laufzeitkontext des erzeugenden Prozesses übernommen wird, verhält sich der *captive* bei seiner erstmaligen Aktivierung entsprechend des *fork*-Prinzips. Dieses Prinzip kann somit zu Erzeugung neuer *custodians* wie auch zum Aufbau von *Teams*, durch die Zuordnung weiterer *captives* zu einem *custodian*, angewendet werden.

Da zur Erzeugung des *captives* das Datensegment des *custodian* nicht dupliziert wird, behält jeder *captive* den direkten Zugriff auf dieses Segment. Damit ist implizit *Shared-Memory* zwischen allen Prozessen eines *Teams* realisiert.

Die Erzeugung eines *captives* ist nur dem *custodian* bzw. anderen *captives* desselben *Teams* gestattet. Der erzeugte *captive* wird damit demselben *Team* zugeordnet, das durch den jeweils erzeugenden Prozeβ vorgegeben ist.

Die Erzeugung eines *custodians* ist nur einem *custodian* selbst gestattet. Damit ist gleichzeitig die Möglichkeit zum Aufbau eines neuen *Teams* gegeben. Bild 4.1 skizziert eine mögliche Hierarchie von *custodians* und *captives*.

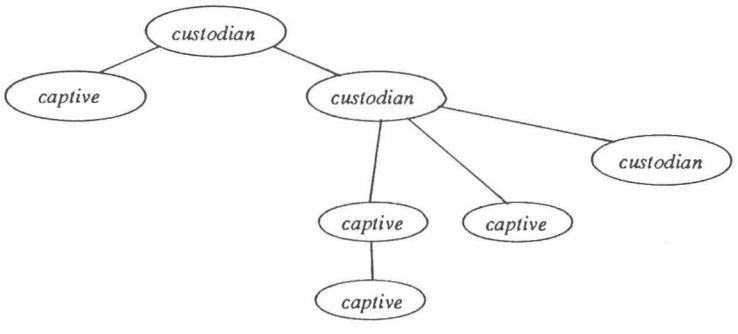

Bild 4.1: Hierarchie zwischen *custodians* und *captives*

Die Erzeugung eines neuen *custodians* bedeutet jedoch nicht, die dem erzeugenden *custodian* evtl. zugeordneten *captives* in das neue *Team* mit zu übernehmen. Unabhängig von der Anzahl der *captives*, die dem erzeugenden *custodian* zugeordnet sind, wird ein neuer *custodian* (und damit evtl. ein neues *Team*) immer als einzelner Prozeβ eines bestimmten Adreβraumes erzeugt.

4.1.3. Objektorientierte Verwaltung

Die Modellierung von Prozessen splittet sich in zwei grundlegende Bereiche auf. Zum einen die logische Modellierung der Prozesse. Dieser Bereich ist mit der Verwaltung der Beziehung zwischen TCB und PCB verbunden. Zum andereren die physikalische Modellierung der Prozesse. Hierbei wird jeweils der *Context-* und *Image-Descriptor* so definiert, daß für den realen Prozessor eine direkte Abbildung des Adreßraumes eines Prozesses auf die zugrunde liegende Hardware möglich ist.

4.1.3.1. Prozeßobjekte

Prozeßobjekte werden jeweils über den PCB beschrieben. Die Identifikation eines Prozeßobjektes erfolgt anhand der mit dem PCB direkt in Beziehung stehenden *process ID* .

Die Verwaltung von Prozeßobjekten ist nicht mit der Abbildung eines Prozesses auf den realen Prozessor konfrontiert. Das bedeutet, es findet noch keine Zuordnung eines Adreßraumes für den jeweils zu erzeugenden Prozeß statt. Mit der Verwaltung von Prozeßobjekten steht vielmehr die logische Beziehung dieser Objekte untereinander im Vordergrund.

Prozeßobjekte werden in einem *Team* zusammengefaßt, um durch den MOOSE-Kernel eine bestimmte Ausführungsstrategie für die in dem *Team* angesiedelten Prozessen zu definieren. Desgleichen werden damit prozeßübergreifende Zugriffsrechte, die für das gesamte *Team* ihre Gültigkeit besitzen, festgelegt. Mit dieser Zusammenfassung von Prozeßobjekten zu einem *Team* wird die Zuordnung von *captives* zu einem *custodian* geregelt. Das bedeutet, einem bereits vorhandenen TCB werden weitere PCBs zugeordnet. Die Erzeugung eines neuen *custodian* bedeutet dagegen, daß ein TCB und ein PCB jeweils alloziiert und in Beziehung zueinander gebracht werden müssen.

4.1.3.2. Kontextobjekte

Die physikalische Modellierung von Prozessen, über entsprechende Kontextobjekte, orientiert sich an dem konkret zugrunde liegenden realen Prozessor. Zur Erzeugung von Prozessen muß ein Adreßraum für den jeweiligen Prozeß eingerichtet werden. Ebenso muß der Dynamik von Prozessen dahingehend Rechnung getragen werden, daß die Erweiterung/Verkleinerung des Adreßraumes eines Prozesses möglich ist, wenn Anforderungen zur Speichervergabe bzw. -freigabe von dem jeweiligen Prozeß erfolgen.

Kontextobjekte werden in einem *Team* zusammengefaßt, um durch den MOOSE-Kernel eine bestimmte Adreßraumdomäne für die in dem *Team* angesiedelten Prozesse zu definieren. Damit werden prozeßübergreifende Zugriffsrechte, auf einen bestimmten durch das *Team* festgelegten Adreßraum, für alle Prozesse innerhalb des *Teams* vergeben. Mit dieser Zusammenfassung von Kontextobjekten zu einem *Team* wird die implizite Definition von *Shared-Memory* zwischen dem *custodian* und den jeweiligen *captives* eines *Teams* ermöglicht. Der gemeinsame Datenbereich aller Prozesse eines *Teams* wird durch den *Image-Descriptor* vorgegeben. Der prozeßlokale Datenbereich (das Stacksegment) wird durch den *Context-Descriptor* definiert. Die Erzeugung eines neuen *custodian* bedeutet somit die Alloziierung eines *Image-* und *Context-Descriptors*, wohingegen die Erzeugung eines *captives* nur noch mit

der Vergabe eines neuen *Context-Descriptors* verbunden ist.

4.1.4. Broadcasts bei der Erzeugung/Zerstörung von Objekten

Die vollständige Erzeugung eines neuen Prozeβobjektes wird im System durch einen entsprechenden *Broadcast* verbreitet. Erst nach Beendigung des *Broadcasts* wird der erzeugte Prozeβ erstmalig gestartet. Für den erzeugenden Prozeβ besteht hierbei die Möglichkeit, auf die Beendigung des *Broadcasts* zu warten oder bereits, nach Vergabe einer *process ID* für den neuen Prozeβ, reaktiviert zu werden. Im letzteren Fall ist der erzeugte Prozeβ nur als "*Zombie*" vorhanden, der noch über keinen Adreβraum verfügt, logisch jedoch bereits modelliert ist.

Die Reaktivierung des erzeugenden Prozesses erst nach Beendigung des *Broadcasts* für den erzeugten Prozeβ, ist in Systemen von Bedeutung, in denen Systemprozesse die Vererbung bestimmter Objekte des Vaterprozesses an den erzeugten Sohnprozeβ durchführen müssen. Ein typisches Beispiel ist hier durch die Vererbung der File-Deskriptoren unter UNIX gegeben. Die Blockierung des Vaterprozesses während des *Broadcasts* für den Sohnprozeβ verhindert, daβ der Vaterprozeβ bereits Objekte wieder freigibt, bevor der Sohnprozeβ vollständig (systemglobal) erzeugt worden ist.

Die Zerstörung eines Prozesses bedeutet mindestens die Freigabe seines prozeβlokalen Speicherbereiches. Diese Maβnahme ist ausreichend bei der Zerstörung eines *captives*. Die Zerstörung eines *custodian* dagegen bedeutet zusätzlich die Freigabe des durch das jeweilige *Team* belegten Speicherbereiches. Dieser Speicherbereich wird jedoch erst dann vollständig freigegeben, wenn der *custodian* den einzigen Prozeβ des *Teams* darstellt. Das bedeutet, die Termination eines *custodian* wird solange verzögert, bis alle anderen *captives* desselben *Teams* ebenfalls ihre Termination angezeigt haben.

Bevor ein Prozeβ vollständig zerstört, d.h. bevor sein gesamter Speicherbereich frei zur Disposition gestellt wird, erfolgt die systemglobale Benachrichtigung über die bevorstehende Zerstörung des Prozesses. Zu diesem Zeitpunkt gilt der jeweilige Prozeβ bereits als terminiert. Erst mit Beendigung des *Broadcasts* für den terminierten Prozeβ, wird die vollständige Zerstörung des Prozesses durchgeführt. Der durch ihn belegte Speicherbereich, sowie alle zur Modellierung des Prozesses notwendig gewesenen Datenstrukturen werden dem System zurückgegeben.

4.2. Identifikation von Diensten

Zur Identifikation von Prozessen ist jedem Prozeβ eine eindeutige *process ID* zugeordnet. Diese *process ID* wird im Zusammenhang mit den Mechanismen zur Interprozeβkommunikation des MOOSE-Kernels angewendet, um den jeweiligen Kommunikationspartner eines Prozesses zu adressieren. Demzufolge ist von allen miteinander kommunizierenden Prozessen gefordert, daβ sie eine Zuordnung von einer *process ID* zu ihrem jeweiligen Kommunkationspartner treffen müssen.

4.2.1. Benennung von Prozeβfunktionalitäten

Eine *process ID* selbst ist semantikfrei. Das bedeutet, daβ von einer *process ID* nicht abgeleitet werden kann, welche Funktionalität der darüber identifizierte Prozeβ für einen bestimmten Systemkomplex erbringt. Dadurch, daβ Prozesse dynamisch erzeugt und zerstört werden können, ist es insbesondere möglich, daβ dieselbe Funktionalität zu unterschiedlichen Zeitpunkten mit anderen *process IDs* in Verbindung gebracht werden muβ und umgekehrt. Die Identifikation eines Kommunikationspartners muβ somit darauf basieren, Funktionalitäten eines Prozesses benennen zu können. Diese Forderung gilt besonders für dezentral organisierte bzw. verteilte Systeme.

Die Benennung der Funktionalitäten richtet sich unter MOOSE nach den jeweiligen von einem Prozeβ zur Verfügung gestellten Diensten. Die Erbringung eines Dienstes durch einen Prozeβ definiert die mit dem Prozeβ assoziierte Funktionalität.

Die Benennung einer Funktionalität erfolgt durch die Angabe eines frei wählbaren Namens durch den die jeweilige Funktionalität erbringenden Prozeβ. Eine in dieser Form ausgezeichnete Funktionalität wird unter MOOSE als Dienst des betreffenden Prozesses bezeichnet. Ein Prozeβ kann in dieser Weise einen einzigen Dienst oder mehrere verschiedene Dienste unter verschiedenen Namen zur Verfügung stellen. Ebenso kann er verschiedene Namen für denselben Dienst vergeben, mit demselben Namen können jedoch nicht verschiedene Dienste assoziiert werden. Zur Verfügung gestellte Dienste können für ungültig erklärt oder anderen Prozessen übertragen bzw. von anderen Prozessen adoptiert werden. Wird ein Dienst für ungültig erklärt, bedeutet dies, daβ der betreffende Dienst nicht weiter zur Verfügung gestellt wird. Dies bedeutet jedoch nicht, daβ die mit dem Dienst assoziierte Funktionalität eines Prozesses verloren geht. Diese Funktionalität kann lediglich nicht mehr angefordert werden, da die Identifikation des entsprechenden Prozesses nur anhand des Dienstnamens möglich ist[37].

4.2.2. Abbildung von Diensten auf Process IDs

Die Abbildung eines Dienstes auf eine *process ID* basiert darauf, daβ Prozesse die durch sie erbrachten Dienste global bekannt geben. Das bedeutet die Exportierung von Diensten durch einen Prozeβ. Prozesse, die Dienste in Anspruch nehmen wollen, importieren die jeweiligen Dienste und erfahren damit die Identifikation des entsprechenden diensterbringenden Prozesses. Diese Identifikation entspricht der *process ID* des jeweiligen Prozesses.

Die Stelle, an der ein Dienst von einem bestimmten Prozeβ in Anspruch genommen werden kann, wird als *service access point* (SAP) [Schindler 1980] bezeichnet. Der *service access point* verbirgt vollständig die Implementierung des jeweiligen Dienstes. Damit ist eine wesentliche Abstraktionsschicht eingeführt worden, die die Identifikation von und die

[37] Hiervon ausgenommen sind Prozeβkomplexe, bei denen implizit bekannt ist, welcher Prozeβ für die Erbringung einer bestimmten Funktionalität verantwortlich ist. Solche Prozeβkomplexe werden sich dadurch auszeichnen, daβ die einzelnen konstituierenden Prozesse von anderen Prozessen desselben Komplexes erzeugt worden sind und damit ihre Identifikation bereits bekannt ist. Die Erzeugung eines Prozesses liefert immer die Identifikation des neuen Prozesses.

Kommunikation zwischen Prozessen unabhängig von der jeweils zugrunde liegenden Systemtopologie ermöglicht. Für alle Prozesse, die Dienste über einen ausgezeichneten *service access point* in Anspruch nehmen, ist die Tatsache eines zugrunde liegenden verteilten oder nicht verteilten Systems – zumindest für die Inanspruchnahme des jeweiligen Dienstes – nicht von Bedeutung.

Bevor die Kommunikation zwischen Prozessen möglich ist, müssen sich die Prozesse jeweils den Zugang zu einem *service access point* verschaffen. Der Zugang zu einem *service access point* wird durch ein bestimmtes Protokoll, dem *service access protocol*, zwischen dem dienstanfordernden und dem diensterbringenden Prozeβ ermöglicht. Dieses Protokoll basiert auf unterster Ebene darauf, daβ der dienstanfordernde Prozeβ die Funktionalität benennt, die er von einem bestimmten diensterbringenden Prozeβ in Anspruch nehmen will. Der anfordernde Prozeβ importiert damit den angeforderten Dienst.

Die Verbindung zwischen dem dienstanfordernden und diensterbringenden Prozeβ kommt dann zustande, wenn der diensterbringende Prozeβ seine Dienste global bekannt gemacht und damit exportiert hat. Bild 4.2 skizziert das grundlegende Modell zwischen einem dienstanfordernden und diensterbringenden Prozeβ.

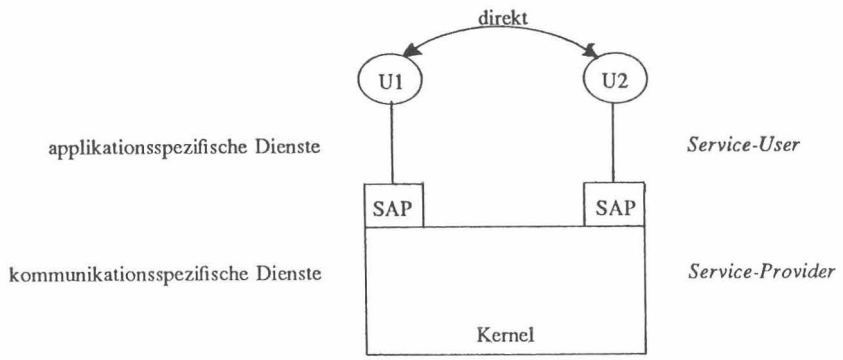

Bild 4.2: Beziehung zwischen *Service-User* und *Service-Provider*

In diesem Beispiel stellen U1 und U2 jeweils miteinander kommunizierende Prozesse dar. Hierbei wird davon ausgegangen, daβ beide Prozesse applikationsspezifische Dienste des jeweils anderen Prozesses in Anspruch nehmen wollen und deshalb eine Kommunikation zwischen den beiden Prozessen stattfinden muβ. Die Kommunikation zwischen U1 und U2 wird jedoch selbst auf die Inanspruchnahme von Diensten eines bestimmten, tieferliegenden Kommunikationssystems basieren. Die kommunikationsspezifischen Dienste werden in diesem Beispiel durch die Mechanismen zur Interprozeβkommunikation des MOOSE-Kernels und durch die Mechanismen zur Identifikation von Prozessen, auf Grundlage der Benennung von Prozeβfunktionalitäten, realisiert. In diesem Sinne (und nach [Schindler 1980]) stellen U1 und U2 beide die *Service-User* der kommunikationsspezifischen Dienste des

Kommunikationssystems dar, und der Kernel sowie die konkrete Ausprägung der SAPs repräsentieren den *Service-Provider*, d.h. das Kommunikationssystem selbst.

Das *service access protocol* kann den jeweiligen Prozessen gegenüber (U1 und U2) transparent gehalten werden. Insbesondere mit der Erbringung von Systemdiensten (*System-Calls*) läßt sich die prozedurale Sichtweise des *System-Calls* durch entsprechende Bibliotheksfunktionen auf die Interprozeßkommunikation zwischen dienstanfordernden und diensterbringenden Prozeß leicht abbilden. In dieser Hinsicht ist ein *System-Call*, wenn auch in sehr idealisierter Form, als *remote procedure call* [Nelson 1982] realisiert. Mit dem *System-Call* ist unter MOOSE in jedem Fall ein Prozeßwechsel und damit die nichtlokale Erbringung eines Dienstes verbunden.

4.2.3. Bedeutung des Dienstbegriffs in netzwerkorientierten Systemen

Das Prinzip des *service access points* ermöglicht es dem dienstanfordernden Prozeß, von der Lage des diensterbringenden Prozesses zu abstrahieren. Entsprechendes gilt für den diensterbringenden Prozeß selbst, der von der Lage des dienstanfordernden Prozesses abstrahieren kann.

Die Problematiken von *remote procedure calls* im Sinne von [Nelson 1982] werden jedoch erst in netzwerkorientierten Systemen voll zum Tragen kommen. Mit der Zuordnung von Diensten zu einzelnen *remote procedure calls* ist unter MOOSE der Ansatz gegeben, diesen Problematiken durch eine systemglobale Identifikation von Prozessen zu begegnen. Der *service access point* trägt bereits in einem lokalen System zur Abstraktion bei, wie die Dienste eines Prozesses in Anspruch genommen werden können. Der Übergang zu einem dezentralen- bzw. verteilten System würde auf der dienstanfordernden Seite bedeuten, einen Vermittler für die netzwerkglobale Dienstanforderung einzusetzen. Zusätzlich wird der *service access point* auf der diensterbringenden Seite durch einen Prozeß unterstützt, der alle vom Netzwerk eintreffenden Dienste den entsprechenden diensterbringenden Prozessen seines lokalen Systems zustellt. Bild 4.3 skizziert in groben Zügen ein entsprechendes Prozeßmodell.

Dieses Beispiel stellt das grundlegende Prinzip unter MOOSE vor, wie dezentrale/verteilte (Betriebs-, Applikations-) Systeme unterstützt werden. Spezielle Systemprozesse werden eingesetzt, die die Verbindung zwischen den einzelnen Rechnersystemen realisieren. Dieser Ansatz, der vergleichbar mit dem in [Akkoyunlu et al. 1974] favorisierten Modell ist, wird unter MOOSE sehr konsequent verfolgt, so daß er auch in eng gekoppelten Multiprozessorsystemen seine Gültigkeit besitzt. Der MOOSE-Kernel ist von jeglichen Verwaltungsmaßnahmen in Netzwerk- oder Multiprozessorsystemen entlastet und beschränkt sich rein auf die Interprozeßkommunikation in lokalen Systemen. Demzufolge wird ein dezentral organisiertes oder verteiltes MOOSE-Betriebssystem darauf basieren, daß der MOOSE-Kernel auf jedem Rechnersystem des jeweils betrachteten Netzwerkkomplexes als Basis zur Verfügung gestellt wird.

Das vorgestellte Modell zeigt, wie ein *service access point* unter MOOSE in einem nichtlokalen System repräsentiert werden kann. Hierbei wird auf das Modell der *distributed abstract machine* (DAM) [Schindler 1980] Bezug genommen. Die DAM wird durch die Systemprozesse S1 und S2 sowie durch das Netzwerk realisiert. Die zentrale Aufgabe von S1

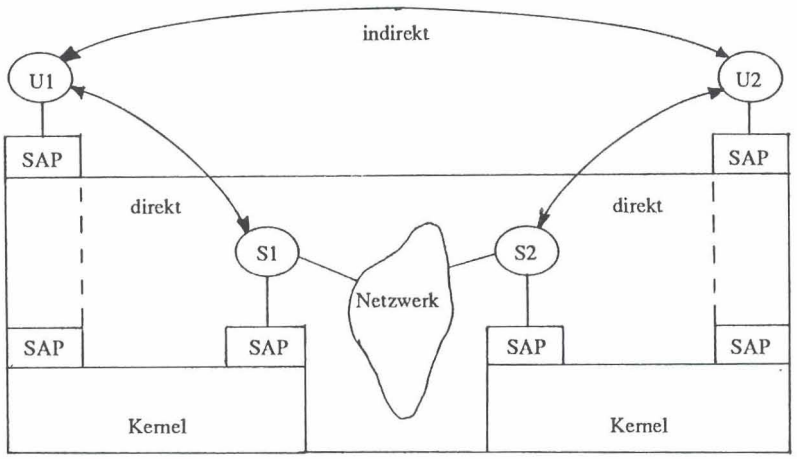

Bild 4.3: Der *service access point* in dezentralen Systemen

und S2 wäre es hierbei, ein *remote procedure call protocol*, etwa nach [Nelson 1982] oder [Panzieri, Shrivastava 1982], zur Verfügung zu stellen, daß die für U1 netzwerktransparente Sichtweise eines *System-Calls* ermöglicht. Die lokalen Systeme werden in diesem Modell jeweils durch das Paar (U1, S1) und (S2, U2) definiert. Die Kommunikation zwischen den Prozessen dieser Systeme wird direkt durch den MOOSE-Kernel unterstützt.

Dieses Modell verdeutlicht noch einen weiteren wesentlichen Sachverhalt. Der MOOSE-Kernel kann als eine spezielle Ausprägung der DAM – nämlich nur im Sinne einer abstrakten und nicht verteilten Maschine – betrachtet werden. Auch in einem nicht verteilten System werden die in [Schindler 1980] dargestellten Prinzipien ihre Gültigkeit besitzen.

Das *service access protocol* zwischen U1 und U2 wird auf ein entsprechendes lokales Protokoll zwischen (U1, S1) und (S2, U2) abgebildet. Für U1 ist der diensterbringende Prozeß in diesem Fall durch S1 repräsentiert, da S1 die für U1 entfernt zugänglichen Dienste von U2 zur Verfügung stellt. Für U2 ist der dienstanfordernde Prozeß durch S2 repräsentiert. S2 nimmt in seinem lokalen System alle entfernt gestellten Dienstanforderungen an U2 entgegen und sorgt damit für die Übermittlung der Dienstanforderung von U1 an U2. S1 und S2 repräsentieren damit *Server*-Prozesse, die den *Service-Provider* einer DAM dahingehend vervollkommnen, daß ein verteiltes (Betriebs-, Applikations-) System verwirklicht werden kann.

Die Funktionalität von S2, z.B., kann verallgemeinert werden. So muß dieser Prozeß nicht unbedingt nur U2 zugeordnet sein, sondern er kann gundsätzlich alle entfernt gestellten applikationsspezifischen Dienstanforderungen an alle diensterbringenden Prozesse seines lokalen Systems annehmen. Damit besteht jedoch für S2 das typische Problem, jederzeit entfernt gestellte Dienstanforderungen entgegennehmen zu können und "gleichzeitig" die lokale Erbringung dieser Dienste einzuleiten. Eine mit der *Newcastle Connection* [Brownbridge et al. 1982] [38] vergleichbare Lösung dieses Problems – für jeden entfernten und dienstanfordernden

[38] Für eine kurze Skizzierung der Struktur der *Newcastle Connection* siehe [Loehr 1985].

Prozeβ wird ein lokal zum diensterbringenden Prozeβ angesiedelter Repräsentant eingerichtet– wäre unter MOOSE sicherlich ebenfalls durchführbar, wird jedoch nicht als einziges und grundsätzliches Modell betrachtet. Vielmehr bietet die konsequente Anwendung des *Team*-Konzeptes von MOOSE die Möglichkeit, mit einer minimalen Anzahl von Prozessen eine Lösung des skizzierten Problems anzugeben. Diese Lösung basiert auf der Grundlage nichtblockierender und problemorientierter Kommunikationsmechanismen[39].

4.2.4. Migration von Diensten

Mit der Benennung der Dienste eines Prozesses ist die eindeutige systemglobale Zuordnung dieses Prozesses zu den von ihm zur Verfügung gestellten Funktionalitäten erfolgt. Anhand des jeweiligen Namens der einzelnen Funktionalitäten –bzw. des der jeweiligen Funktionalität entsprechenden Dienstes– kann die Identifikation des entsprechenden diensterbringenden Prozesses erfolgen.

Bei diesem Verfahren besteht keine Notwendigkeit, die Zuordnung eines Dienstes zu dem jeweiligen diensterbringenden Prozeβ statisch zu realisieren, d.h. daβ durch die Exportierung eines Dienstes statisch immer derselbe Prozeβ für die Erbringung dieses Dienstes verantwortlich ist. Insbesondere bei dezentralen- oder verteilten Systemen ist es notwendig, diese Zuordnung dynamisch gestalten zu können. Die Migration von Prozessen, etwa nach den in DEMOS/MP [Powell, Miller 1983] realisierten Prinzipien, bedeutet in solchen Systemen oftmals, daβ die von den Prozessen zur Verfügung gestellten Dienste ebenfalls migriert werden müssen. Dies ist darauf zurückzuführen, daβ die Migration eines Prozesses im wesentlichen zur entfernten Erzeugung eines repräsentativen und zur lokalen Zerstörung des zu migrierenden Prozesses führt. Damit ändert sich zwangsläufig die *process ID* des migrierten und diensterbringenden Prozesses.

Unter MOOSE wird jedoch bereits in lokalen Systemen die Migration von Diensten notwendig sein. Der Grund hierfür ist durch das Konzept einer Familie von Betriebssystemen vorgegeben. Diese Notwendigkeit ist jedoch keinesfalls typisch für MOOSE, sie ist vielmehr allgemeingültig, so auch in FAMOS [Habermann et al. 1976]. Die Modellierung einer Betriebssystemfamilie verlangt ein bestimmtes Maβ an Dynamik der in der Familie zusammengefaβten Systemkomponenten, die unter MOOSE als Systemprozesse realisiert werden. Die Integration eines neuen Systemprozesses kann bedeuten, Dienste anderer Systemprozesse mit zu übernehmen oder bestehende Systemprozesse (in zwei oder mehrere Systemprozesse) aufzusplitten. Ebenso kann die Herausnahme eines Systemprozesses aus einem bestehenden Systemkomplex bedeuten, die von diesem Prozeβ zur Verfügung gestellten Dienste anderen Systemprozessen –und sei es nur zum Zwecke einer möglichen Ausnahmebehandlung– zu übertragen.

Die Techniken der Migration eines Dienstes unter MOOSE unterstützen die dynamische Rekonfiguration eines bestimmten Systemkomplexes dahingehend, daβ immer, gemäβ der jeweils aktuellen Systemkonfiguration, eine eindeutige Identifikation eines diensterbringenden

[39] Eine Diskussion der Modelle nichtblockierender Kommunikationsmechanismen unter MOOSE findet im nachfolgenden Kapitel statt.

Prozesses erreicht wird. Die Migration eines Dienstes erfolgt unteilbar und basiert darauf, den betreffenden Dienst eines Prozesses zu adoptieren bzw. zu vererben. Die hierzu notwendigen Verwaltungsmaßnahmen sind dennoch sehr elementar. Sie beschränken sich, wie im Zusammenhang mit der Importierung und Exportierung von Diensten, im wesentlichen auf die Assoziation einer *process ID* mit einem Dienstnamen.

Die Adoption eines Dienstes bedeutet die Übernahme des betreffenden Dienstes durch den adoptierenden Prozeß. Die Vererbung eines Dienstes bedeutet die Zuordnung des betreffenden Dienstes zu einem anderen Prozeß. In beiden Fällen gilt jedoch, daß nach erfolgter Migration in mindestens zwei Prozessen die gleiche Funktionalität angesiedelt ist. Damit behalten alle noch bestehenden Beziehungen zu den ursprünglichen diensterbringenden Prozessen ihre Gültigkeit. Jede nach der Migration eines Dienstes erfolgte Abbildung des Dienstes auf eine *process ID* wird die Identifikation des neuen diensterbringenden Prozeß liefern[40].

4.3. Prozeßabstraktion

Die Verfahren unter MOOSE zur Erzeugung, Zerstörung und systemglobalen Identifikation von Prozessen ermöglichen es, Prozesse als Bausteine für Betriebs- und Applikationssysteme einsetzen zu können. Diese Verfahren werden durch drei Systemprozesse realisiert. Die Systemprozesse definieren hierbei einen abstrakten Prozessor [Schindler 1983], der Prozesse als dynamische Objekte betrachtet, die erzeugt und wieder zerstört werden können. Damit wird die statische Sichtweise des MOOSE-Kernels um eine Funktionalität erweitert, die den Aufbau und die dynamische Rekonfiguration einer Betriebssystemfamilie ermöglicht.

Die nachfolgend vorgestellten Systemprozesse stehen stellvertretend für eine bestimmte, mit MOOSE verbundene Klasse von Prozessen, den *Administratoren*. Die *Administratoren* nehmen für ein MOOSE-Betriebssystem wesentliche Funktionalitäten wahr und dienen dazu, eine bestimmte, funktional zusammenhängende Gruppe von Prozessen begrifflich fassen zu können. Die in einem *Administrator* zusammengefaßten Systemprozesse können selbst in einem einzigen *Team* konzentriert oder über mehrere *Teams* verteilt sein. Mit dieser Sichtweise ergibt sich eine etwas andere Begriffsbildung als sie in [Gentleman 1981] getroffen worden ist.

4.3.1. Team-, Context- und Service-Administrator

Die drei Systemprozesse zur Realisierung der Prozeßabstraktion unter MOOSE, stehen in einer eindeutigen hierarchischen Beziehung zueinander. Bild 4.4 skizziert das Modell des durch die Systemprozesse realisierten abstrakten Prozessors.

Der *Team-Administrator* übernimmt die logische Verwaltung von Prozeßobjekten. Er kontrolliert die Erzeugung und Zerstörung von Prozessen und sorgt dafür, daß Prozesse bestimmten *Teams* zugeordnet werden. In diesem Zusammenhang ist insbesondere die Kontrolle des *Broadcasts* für die zu erzeugenden und zu zerstörenden Prozesse hervorzuheben.

[40] Wie der ursprüngliche diensterbringende Prozeß aus dem Systemkomplex entfernt werden kann, wird im Zusammenhang mit der dynamischen Rekonfiguration unter MOOSE dargestellt.

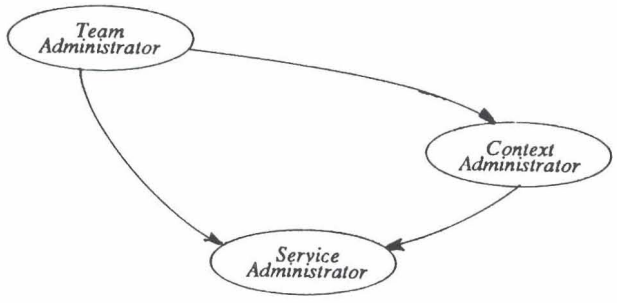

Bild 4.4: Zentrale Systemprozesse zur Realisierung der Prozeβabstraktion

Das bedeutet, die Funktion des *Broadcast-Masters* wird vom *Team-Administrator* zusätzlich übernommen.

Der *Context-Administrator* übernimmt die physikalische Verwaltung von Prozessen. Dieser Systemprozeβ ordnet jedem Prozeβ bzw. *Team* einen bestimmten Adreβraum zu. Er sorgt dafür, daβ die zur Ausführung der Prozesse notwendige Abbildung auf einen realen Prozessor durch den MOOSE-Kernel ermöglicht wird. Damit ist der *Context-Administrator* abhängig von der Hardware-Organisation des jeweiligen realen Prozessors.

Der *Service-Administrator* ermöglicht die systemglobale Identifikation von Prozessen anhand der von den Prozessen zur Verfügung gestellten Dienste. Dieser Systemprozeβ verzeichnet alle exportierten Dienste und ermöglicht die Abbildung von Diensten auf *process IDs*. Mit dieser Funktionalität des *Service-Administrators* wird die Realisierung eines *service access points* und eines *service access protocols* unter MOOSE unterstützt.

4.3.2. Beziehung zwischen den zentralen Systemprozessen

Die dargestellte Beziehung zwischen den drei Systemprozessen ergibt sich aufgrund verschiedener Tatsachen. Zum einen stellen diese Prozesse jeweils diensterbringende Prozesse dar, d.h. sie exportieren Dienste, die von anderen Prozessen in Anspuch genommen werden können. Aus diesem Grunde besteht vom *Team-* und *Context-Administrator* jeweils die hierarchische Beziehung zum *Service-Administrator*.

Zum anderen benutzt der *Team-Administrator* Dienste des *Context-Administrators*. Diese Dienste bewirken das Einrichten eines Adreβraumes und Laufzeitkontextes für einen zu erzeugenden Prozeβ. Die Identifikation des *Context-Administrators* erfährt der *Team-Administrator* über den *Service-Administrator*.

4.3.3. Identifikation des Service-Administrators

Die Identifikation von Prozessen anhand der von diesen Prozessen exportierten Dienste, erfolgt durch die Erbringung eines entsprechenden Dienstes des *Service-Administrators*. Um diese Dienste des *Service-Administrators* in Anspruch nehmen zu können, kann jedoch der *Service-Administrator* selbst nicht zur Abbildung eines Dienstnamens auf die *processID* des entsprechenden diensterbringenden Prozesses, nämlich der *Service-Administrator* selbst, herangezogen werden. Damit ergibt sich ein offensichtliches Identifikationsproblem, da diese Abbildung nur durch Dienste des *Service-Administrators* erbracht wird.

Um das Identifikationsproblem zu umgehen, erfolgt die Identifikation des *Service-Administrators* nach einem anderen Verfahren, als die Identifikation der restlichen Prozesse des MOOSE-Systems. Hierzu wird auf bestimmte Dienste des MOOSE-Kernels zurückgegriffen, die im Zusammenhang mit der Verwaltung von *Ports* stehen. Bestimmte Prozesse können sich über ein *specific attach* an einen vorgegebenen *Port* anbinden, und ebenso kann die Identifikation des jeweils an einen *Port* angebundenen Prozesses in Erfahrung gebracht werden. Der *Service-Administrator* macht sich diese Funktionalität des MOOSE-Kernels zunutze und belegt einen bestimmten *Port*, über den dann die Identifikation des *Service-Administrators* nachgefragt werden kann.

Damit ergeben sich für die Identifikation von Prozessen generell zwei Ebenen für das *service access protocol*:

Ebene 0 die Identifikation des *Service-Administrators* anhand des durch ihn belegten *Ports*. Hierzu werden Dienste des MOOSE-Kernels in Anspruch genommen;

Ebene 1 die Identifikation der jeweiligen Prozesse anhand der durch sie erbrachten und benannten Funktionalitäten. Hierzu werden Dienste des *Service-Administrators* in Anspruch genommen.

Dieses zweistufige Verfahren ermöglicht einerseits die effiziente Identifikation des *Service-Administrators* und erlaubt andererseits, daß der MOOSE-Kernel mit der Problematik der systemglobalen Identifikation von Prozessen nicht konfrontiert wird. Insbesondere dadurch, daß die Identifikation von Prozessen durch einen Prozeß selbst durchgeführt wird, ist die Möglichkeit gegeben, weitergehende Identifikationsverfahren in ein bestehendes System zu integrieren.

Kapitel 5

Das Familienkonzept

Die in den vorigen Kapiteln vorgestellten Mechanismen erlauben einerseits die konsequente Anwendung von Prozessen zur Modellierung und Realisierung von Software-Systemen, andererseits werden sehr laufzeiteffiziente Verfahren zur Interprozeβkommunikation durch den MOOSE-Kernel zur Verfügung gestellt. Mit diesen Mechanismen ist die wesentliche Grundlage gegeben, spezifische Betriebs- und Applikationssysteme als eine gemeinsame Familie realisieren zu können.

In diesem Kapitel wird vorgestellt, welche typischen Komponenten eines Betriebssystems als *Building Blocks* unter MOOSE verstanden werden, um sie jeweils als Komponenten zum Aufbau eines spezifischen, applikationsorientierten Systems einsetzen zu können. Prozesse werden als Strukturierungshilfsmittel zum Betriebssystementwurf in den Vordergrund gestellt. Hierbei wird die Linie von [Cheriton 1979] konsequent weiterverfolgt.

Der Leitfaden, der sich durch das in MOOSE verfolgte Konzept einer Familie von Betriebssystemen hervorhebt, läβt sich kurz und aussagekräftig durch ein Zitat aus [Liskov 1981] darstellen:

"*...a major concern is what ideas to exclude from the design.*"

In [Liskov 1981] ist dieser Leitfaden im Zusammenhang mit der Vorstellung der grundlegenden Konzepte von THOTH und dem sich daraus entwickelnden V-System [Cheriton 1984] genannt worden. MOOSE entspricht vollständig dieser Maxime und den in [Lampson 1983] skizzierten Entwurfsprinzipien komplexer Software-Systeme.

5.1. Prozeβdomänen

Mit dem *Team*-Konzept wird vom MOOSE-Kernel bereits eine bestimmte Form der Gruppierung von Prozessen erreicht. Mehrere *captives* werden hierbei einem *custodian* zugeordnet. Die *captives* besitzen dabei den Zugriff auf dasselbe Code- und Datensegment des jeweiligen *Teams*. Das *Team* definiert damit eine Adreβraumdomäne für eine bestimmte Gruppe von Prozessen.

Neben der physikalischen Beziehung zwischen Prozessen (dargestellt durch das *Team*-Konzept) ist deren gegenseitiger logischer Zusammenhang in einem Betriebssystem ebenso von wesentlicher Bedeutung. Die Gruppierung von Prozessen ermöglicht es, nicht nur einzelne Prozesse mit bestimmten Systemaktivitäten in Verbindung zu bringen. Oftmals ist es sinnvoll, ganze Prozeβkomplexe hierbei zu betrachten. Beispiele, die den logischen Zusammenschluβ von Prozessen motivieren, sind das *Scheduling* und/oder *Swapping* von Prozeβgruppen, die Vergabe von Zugriffsrechten an Applikationsprozesse sowie die applikationsorientierte

Behandlung systemspezifischer Ereignisse (etwa die Termination eines Prozesses oder die Propagation von anderen Ausnahmesituation).

5.1.1. Benutzergruppen

Mit "Benutzer" wird unter MOOSE ein sehr weit gefaßter Begriff verstanden, der Personen, Geräte und Prozesse als Bestandteile einer bestimmten Applikation zu betrachten erlaubt. Eine Applikation stellt hierbei einen in sich abgeschlossenen Komplex dar, der zur Erbringung einer bestimmten Funktionalität beiträgt. Hierzu muß immer mindestens ein Prozeß vorhanden sein, der den Ablauf der Applikation ermöglicht. Üblicherweise baut sich eine Applikation jedoch aus mehreren Prozessen auf, die jeweils miteinander kooperieren und damit die Gesamtfunktionalität der durch sie konstituierten Applikation erbringen. In diesem Zusammenhang sind insbesondere Prozesse als Repräsentanten für Personen und Geräte zu verstehen, um somit einheitlich, auf der Ebene von Prozessen, die Abarbeitung einer bestimmten Applikation durch einen abstrakten/realen Prozessor zu ermöglichen.

Um Applikationen unabhängig voneinander modellieren und vor nicht authorisierten Zugriffen von Prozessen anderer Applikationen schützen zu können, ist mit jedem Prozeß ein bestimmtes Zugriffsrecht assoziiert. Dieses Zugriffsrecht richtet sich nach der Identifikation der jeweiligen Benutzergruppe eines Prozesses. Diese Identifikation wird als *user group ID* bezeichnet.

5.1.2. Prozeßgruppen

Mit der Unterteilung einer Applikation in verschiedene miteinander kooperierende Prozesse ist es sinnvoll, einen bestimmten Teilkomplex der Applikation von einer bestimmten Anzahl von Prozessen kontrollieren zu lassen. Die Prozesse, die den jeweiligen Teilkomplex einer Applikation repräsentieren, werden in eine Prozeßgruppe zusammengefaßt. Mit jeder Prozeßgruppe ist eine eindeutige Identifikation verbunden, die *process group ID*, die es gestattet, den Teilkomplex einer Applikation explizit bestimmen zu können.

Prozeßgruppen stellen unter MOOSE Gebilde dar, die im Zusammenhang mit bestimmten Systemaktivitäten als Einheit betrachtet werden. Solche Systemaktivitäten sind insbesondere die Termination, die Prozessorvergabe (*Scheduling*) an *Teams*, das Aus-/Einlagern (*Swapping*) von *Teams* und die benutzerspezifische Behandlung von Ausnahmesituationen.

5.1.3. Familiengruppen

Prozesse sind unter MOOSE immer einer bestimmte Betriebssystemdomäne zugeordnet. Diese Domäne wird jeweils durch das zum Einsatz gebrachten MOOSE-Betriebssystem oder durch ein Emulationssystem bestimmt. Die Emulatoren stellen hierbei, basierend auf einem bestimmten Referenzsystem, Dienste anderer Betriebssysteme zur Verfügung. Sie stellen damit die Verbindungsglieder zwischen unterschiedlichen Betriebssystemdomänen dar.

Unabhängig von ihrer Zuordnung zu bestimmten Betriebssystemdomänen, wird die Ausführung der einzelnen Prozesse immer durch den MOOSE-Kernel ermöglicht. Dies bedeutet im wesentlichen, daß die Zuordnung von Adreßräumen zu den jeweiligen Prozessen, nach den durch den MOOSE-Kernel vorgegebenen Prinzipien erfolgt. Hierbei werden insbesondere die Datenstrukturen des MOOSE-Kernels zur Modellierung von Prozessen, unabhängig von ihrer Betriebssystemdomäne, einheitlich eingesetzt, d.h. der TCB, PCB, *Image-* und *Context-Descriptor* findet jeweils seine entsprechende Anwendung. Dies führt dazu, daß die Prozesse, unabhängig ihrer Zugehörigkeit zu bestimmten Betriebssystemdomänen, vom MOOSE-Kernel bei ihrer Ausführung unterstützt werden können.

Für bestimmte Aktivitäten im Zusammenhang mit der Ausführung eines Prozesses, ist dessen Zugehörigkeit zu einer bestimmten Betriebssystemdomäne jedoch von wesentlicher Bedeutung. Insbesondere im Zusammenhang mit der Behandlung von Traps, kann für jede Betriebssystemdomäne eine unterschiedliche Reaktion auf diese für einen Prozeß entstandene Ausnahmesituation vorgegeben sein. Beispiele sind z.B. *Core-Dumps*, die jeweils unterschiedliche Formate aufweisen können, oder Strategien zur Propagation der Traps zu Benutzerprozessen.

Eine Betriebssystemdomäne wird durch eine bestimmte Anzahl von Prozessen definiert, die derselben Familiengruppe zugeordnet sind. Eine Familiengruppe wird eindeutig durch die mit jedem Prozeß verbundene *family group ID* identifiziert. Die *family group ID* ist der Schlüssel dafür, einzelne Mitglieder der MOOSE-Betriebssystemfamilie zu unterscheiden.

5.1.4. Bedeutung einer Prozeßdomäne

Der Begriff der "Domäne" wird unter MOOSE verwendet, um für jeden Prozeß einen zu einem bestimmten Zeitpunkt gültigen Laufzeitkontext bestimmen zu können. Dieser Laufzeitkontext wird hierbei nicht nur durch die jeweilige Zugehörigkeit des Prozesses zu einem bestimmten *Team* und damit zu einem bestimmten Adreßraum definiert. Die Domäne eines Prozesses ist vielmehr die Erweiterung dieses Laufzeitkontextes und verbindet den Prozeß mit bestimmten Zugriffsrechten auf systemglobale Objekte (z.B. Files und Prozesse). Ebenso ermöglicht die jeweilige Domäne eines Prozesses die Entscheidung, welche systemspezifischen Maßnahmen für einen Prozeß zu treffen sind, wenn bestimmte Systemereignisse (z.B. die Termination eines Prozesses) aufgetreten sind. Mit jeder Domäne kann eine eigene Behandlungsstrategie solcher Systemereignisse erzielt werden.

Nur in einem Fall ist ein Prozeß während seiner "Lebensdauer" fest einer bestimmten Domäne zugeordnet. Diese Domäne wird durch das *Team* vorgegeben, das für den Prozeß die jeweilige Adreßraumdomäne definiert. Ein Wechsel dieser Domäne während der Ausführung eines Prozesses, wie mit dem DAS-System [Mueller et al. 1980] beispielhaft vorgestellt, bildet unter MOOSE nicht den zentralen Hintergrund zur Verwaltung eines Prozesses sowie zur Definition des Adreßraumes für einen Prozeß.

Dem gegenüber ist die Zuordnung von Benutzer-, Prozeß- und Familiengruppen zu einzelnen Prozessen nicht statisch festgelegt. Prozesse können somit während ihrer Ausführung jeweils unterschiedlichen Prozeßdomänen zugeordnet werden. Die jeweilige Prozeßdomäne ergibt sich hierbei aus einer beliebigen Kombination der einzelnen Gruppierungsmöglichkeiten

für den Prozeβ. Bild 5.1 stellt modellhaft dar, wie unterschiedliche Prozeβdomänen aufgebaut sein können.

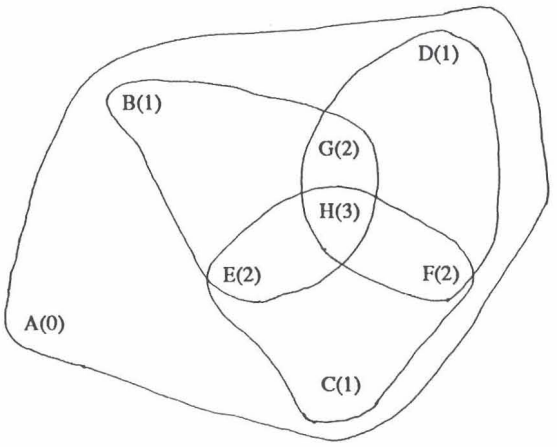

Bild 5.1: Modellierung verschiedener Prozeβdomänen

Diese Grafik zeigt 8 verschiedene Prozeβdomänen, die sich jeweils aufgrund entsprechender Schnittmengenbildungen der einzelnen Gruppen der Prozesse ergeben. Die Numerierung der einzelnen Bereiche repräsentiert jeweils einen bestimmten *Domain-Level*. Ein *Domain-Level* zeichnet sich durch die Anzahl signifikanter Gruppenidentifikationen zur Bildung der jeweiligen Prozeβdomäne aus. Die Tabelle 5.1 gibt hierzu einen Überblick über die verschiedenen *Domain-Level* und wie sich eine Prozeβdomäne im einzelnen als Schnittmenge der verschiedenen Gruppenidentifikationen versteht.

Domain-Group	A	B	C	D	E	F	G	H
user group ID	x	*u*	x	x	*u*	x	*u*	*u*
process group ID	x	x	*p*	x	*p*	*p*	x	*p*
family group ID	x	x	x	*f*	x	*f*	*f*	*f*
Domain-Level	0	1			2			3

Tabelle 5.1: Zusammensetzung der *Domain-Level*

Das x in der dargestellten Tabelle kennzeichnet jeweils eine privilegierte Gruppenidentifikation [41]. Die Gruppenidentifikation x zeigt an, daβ ein durch sie erfaβter Prozeβ keiner Einschränkung beim Zugriff auf Objekte der jeweiligen Gruppe unterlegen ist. Dieser Prozeβ operiert damit gruppenübergreifend und wird als *Group-Master* bezeichnet.

[41] Im Sinne von UNIX würde ein mit x bezeichneter Prozeβ als *Super-User* betrachtet werden, der administrative Aufgaben der betreffenden *Domain-Group* wahrnimmt.

Eine Prozeβdomäne wird durch das Tripel (*u,p,f*) eindeutig festgelegt. Hierbei sind *u*, *p* und *f* jeweils nicht fest, sondern können den Prozessen dynamisch zugeordnet werden. Die Wertänderung einer der Komponenten des Tripels (*u,p,f*) entspricht einem Domänenwechsel des durch das Tripel erfaβten Prozesses. Dazu gelten für die jeweilige Änderung der Benutzer-, Prozeβ- und Familiengruppe eines Prozesses folgende Regelungen:

Benutzergruppe — nur dem *Group-Master* ist erlaubt die *user group ID* eines bestimmten Prozesses zu ändern;

Prozeβgruppe — Die Modifikation der *process group ID* eines anderen Prozesses ist nur dem *Group-Master* erlaubt oder dann möglich, wenn zwischen beiden involvierten Prozessen eine Vater-Sohn-Beziehung besteht. Der modifizierende Prozeβ muβ in diesem Fall der Vater des jeweils von der Modifikation betroffenen Prozesses sein. Jeder Prozeβ kann eigenständig seine *process group ID* modifizieren, wenn

a) die *process group ID* noch nicht vergeben ist, oder

b) die bereits vergebene *process group ID* mit einer *user group ID* verbunden ist, die mit der des jeweils betrachteten Prozesses übereinstimmt;

Familiengruppe — nur mit der Ausführung eines neuen Programmes durch einen Prozeβ wird die *family group ID* des betrachteten Prozesses definiert.

5.1.5. Der Domain-Administrator

Die Verwaltung von Prozeβdomänen übernimmt ein bestimmter Systemprozeβ, der *Domain-Administrator*. Die Aufgabe dieses Prozesses beschränkt sich darauf, Domänen für Prozesse zu definieren und die domänenspezifische Behandlung bestimmter Systemereignisse zu ermöglichen. Bild 5.2 skizziert die Position des *Domain-Administrators* innerhalb des MOOSE-Systems.

Die Beziehung des *Domain-Administrators* zum *Service-Administrator* ermöglicht das Importieren der Dienste anderer Systemprozesse und führt selbst dazu, daβ eigene Dienste des *Domain-Administrators* exportiert werden. Diese Dienste bedeuten, neben der Zuordnung von Domänen für die Prozesse, im wesentlichen, beliebige Systemdienste domänenglobal zu initiieren. Da diese Systemdienste üblicherweise über die Mechanismen zur Interprozeβkommunikation aufgerufen werden, entspricht die domänenglobale Initiierung solcher Dienste einem *one-to-many-send*, ähnlich wie es im Zusammenhang mit dem V-System [Cheriton 1984] vorgestellt worden ist. Beispiele von Systemdiensten, die diese Funktionalität des *Domain-Administrators* in Anwendung bringen können, sind die Zerstörung aller Prozesse einer bestimmten Prozeβdomäne (in 4.2BSD das *killgrp*) und die Freigabe des gesammten Speicherbereiches einer Domäne.

Die Beziehung des *Domain-Administrators* zum *Process-* und *Context-Administrator* ergibt sich einerseits durch die domänenglobale Initiierung von Diensten, die von diesen beiden Systemprozessen jeweils dazu zur Verfügung gestellt werden. Andererseits wird der *Domain-Administrator* zur Erbringung eigener Funktionalitäten selbst Dienste des *Process-* und

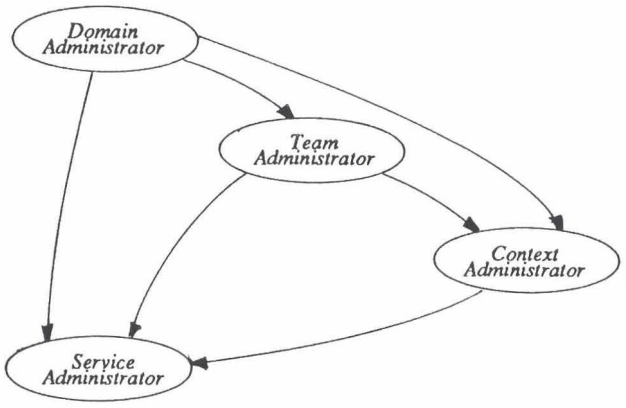

Bild 5.2: Einordnung des *Domain-Administrators*

Context-Administrators in Anspruch nehmen müssen.

5.2. Domänenspezifische Behandlung von Ausnahmesituationen

Ausnahmesituationen, auch als *Exceptions* bezeichnet, spielen bei dem Entwurf und der Realisierung komplexer Software-Systeme eine zentrale Rolle. Ihre Definition, wie auch ihre Klassifikation, steht im Rahmen dieser Arbeit jedoch nicht im Vordergrund. Hierzu sei auf [Goodenough 1975] verwiesen. Bemerkenswert ist jedoch, daß sich insbesondere die Darstellung der Behandlung von Ausnahmesituationen jeweils nur auf die programmiersprachliche Ebene bezieht. Für das Signalisieren bestimmter Ausnahmesituationen zu dieser Ebene sind jedoch oftmals sehr "abenteuerliche" Vorgänge innerhalb eines Betriebssystems verantwortlich. Das Betriebssystem muß dafür sorgen, daß Ausnahmesituationen, z.B. durch *Hardware-Traps* angezeigt, auf die programmiersprachliche Ebene hochgereicht wird. Desweiteren kann das Betriebssystem, als abstrakter Prozessor [Schindler 1983], selbst bestimmte Ausnahmesituationen anzeigen, die zur Behandlung durch Benutzerprozesse gedacht sind.

5.2.1. Verschiedene Techniken zum Signalisieren einer Ausnahmesituation

Die Behandlung einer Ausnahmesituation (*Exception*) wird durch einen *Exception-Handler* durchgeführt. Da der Zeitpunkt, wann eine Ausnahmesituation eintritt, nicht von vornherein bekannt ist – Ausnahmesituationen werden lediglich erwartet und sind im allgemeinen nicht vorbestimmt –, muß die Aktivierung des *Exception-Handlers*, vergleichbar mit einem Trap oder Interrupt, jederzeit möglich sein. Im programmiersprachlichen Sinne entspricht die Aktivierung des *Exception-Handlers* einem nicht programmierten Prozeduraufruf.

Auf der programmiersprachlichen Ebene – genauer, durch das jeweilige Laufzeitsystem – läßt sich die Aktivierung eines *Exception-Handlers* sehr einfach gestalten. Dies ist im wesentlichen dadurch begründet, daß die eine Ausnahmesituation anzeigende Ebene und die eine Ausnahmesituation behandelnde Ebene gemeinsam in demselben Adreßraum angesiedelt sind. Komplizierter stellt sich bereits die Situation dar, wenn die anzeigende und die behandelnde Ebene jeweils verschiedenen Prozessen und damit unterschiedlichen Adreßräumen zugeordnet sind. Das Anzeigen der Ausnahmesituation ist in diesem Fall nur durch Zuhilfenahme geeigneter Mechanismen zur Interprozeßkommunikation möglich.

Eine besondere Problematik ergibt sich in der Situation, wo zwar eine prozedurale Beziehung zwischen der signalisierenden und der behandelnden Ebene besteht, jedoch jeweils unterschiedliche Adreßräume mit den einzelnen Ebenen assoziiert sind. Dies ist die Standardsituation in prozedurorientierten Betriebssystemen (z.B. UNIX). Aber auch prozeßorientierte Betriebssysteme (z.B. QNX) können sehr wohl mit dieser Problematik konfrontiert sein. Die Aktivierung eines *Exception-Handlers* ist mit diesem Hintergrund üblicherweise nur durch die Manipulation des Laufzeitkontextes (des Stacks) des Prozesses möglich, unter dessen Kontrolle der *Exception-Handler* die Ausnahmesituation behandeln soll.

5.2.2. Typisches Modell für prozedurorientierte Betriebssysteme

Um das unter MOOSE zur Verfügung stehende Modell zur Aktivierung eines *Exception-Handlers* besser bewerten zu können, soll an dieser Stelle das entsprechende Modell von UNIX einer kurzen Erläuterung unterzogen werden. Dieses Modell besitzt für alle UNIX-Versionen seine Gültigkeit, obgleich insbesondere mit UNIX 4.2 BSD [Joy et al. 1983] funktional erheblich angereicherte Mechanismen zur Signalbehandlung [42)] zur Verfügung gestellt worden sind. Ebenso findet dieses Modell seine Anwendung in dem prozeßorientierten Betriebssystem QNX [Quantum 1984].

Das angewendete Modell basiert darauf, für einen bestimmten Prozeß einen "Interrupt" zu simulieren und damit eine Ausnahmesituation für einen Prozeß zu signalisieren. Dieser Prozeß wird verantwortlich für die Ausführung des *Exception-Handlers* sein. Die Aktivierung des *Exception-Handlers* erfolgt mit der nächsten Aktivierung des den *Exception-Handler* kontrollierenden Prozesses. Mit der Termination des *Exception-Handlers* setzt der Prozeß an der Stelle seine Ausführung fort, an der er durch die Ausnahmesituation unterbrochen worden ist.

Technisch wird dieser Ablauf durch den Betriebssystemkernel ermöglicht. Das eigentliche Kernproblem besteht hierbei in der Simulation des "Interrupts". Der Laufzeitkontext des betreffenden Prozesses muß so manipuliert werden, daß bei der nächsten Aktivierung dieses Prozesses zuerst der *Exception-Handler* gestartet wird. Der alte Laufzeitkontext muß jedoch beibehalten werden, so daß mit der Termination des *Exception-Handlers* der Prozeß dort mit seiner Ausführung weiter fortfährt, wo er zuletzt unterbrochen worden ist. Bild 5.3 skizziert,

[42)] Signale besitzen unter UNIX die Bedeutung von Ausnahmesituationen, die von Prozessen oder dem Kernel signalisiert worden sind. Diese Signale stehen in keiner Beziehung zu den Signalen des MOOSE-Kernels.

wie sich der Laufzeitkontext des betreffenden Prozesses unter Einwirkung des Kernels prinzipiell verändert.

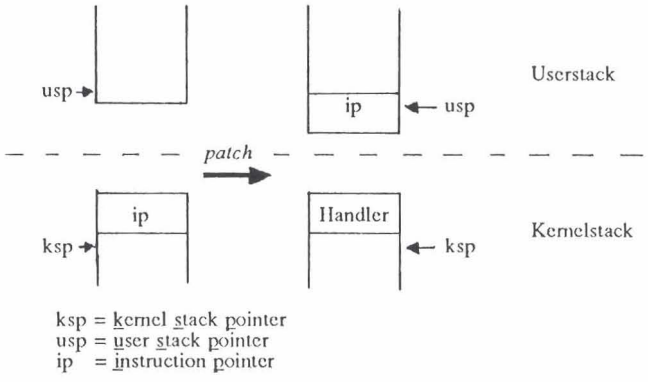

Bild 5.3: Manipulation des Laufzeitkontextes

Dieses Beispiel soll im wesentlichen zwei Tatsachen verdeutlichen:

a) der bei dem Übergang vom *Usermode* in den *Kernelmode* von dem realen Prozessor auf den *Kernelstack* gerettete Prozessorstatus des betreffenden Prozesses wird manipuliert. Hierzu wird im wesentlichen der *Instruction-Pointer* mit der Adresse des entsprechenden *Exception-Handlers* im *Usermode* definiert;

b) der *Userstack* des betreffenden Prozesses wird manipuliert. Diese Manipulation äußert sich darin, daß der *Userstack* mindestens um den *Instruction-Pointer* des vormals geretteten Prozessorstatus erweitert wird. Damit wird auf dem *Userstack* ein aus dem *Usermode* erfolgter Aufruf des *Exception-Handlers* modelliert.

Die genaue technische Realisierung dieses Modells wird sich mit jeder UNIX-Version durchaus unterscheiden können. Das generelle Prinzip, daß mindestens der *Instruction-Pointer* des geretteten Prozessorstatus manipuliert wird, um die Aktivierung des *Exception-Handlers* zu ermöglichen, ist allen UNIX-Versionen gemeinsam.

Das bedeutende Problem in diesem Zusammenhang ist weniger die Manipulation des Prozessorzustandes, als die Expandierung des *Userstacks*. Hierzu muß evtl. der Adreßraum des betreffenden Prozesses erweitert werden, um Platz für die zusätzlichen Argumente eines *Exception-Handlers*, zumindest jedoch um Platz für den alten *Instruction-Pointer* des vormals geretteten Prozessorstatus zu schaffen. Diese Manipulation des *Userstacks* ermöglicht erst, daß mit der Termination des *Exception-Handlers* der Prozeß seine Ausführung an der Stelle wieder aufnimmt, an der er zuletzt vom *User-* in den *Kernelmode* gewechselt ist.

Zusätzlich bedingt dieses Verfahren die Notwendigkeit einer physikalischen Trennung zwischen *Userstack* und *Kernelstack*, damit die Expansion des *Userstacks* direkt möglich ist. Wäre der *Kernelstack* physikalisch direkt hinter dem *Userstack* angeordnet, müßte der

Kernelstack vor der Expansion des *Userstacks* evtl. verschoben werden. Diese Problemtatik ist z.B. ein wesentlicher Grund, weshalb in QNX nur ein *Kernelstack* für alle Prozesse vorhanden ist, der exklusiv dem Kernel selbst zugeordnet ist und physikalisch von allen *Userstacks* getrennt ist[43]. Andererseits ermöglicht erst der Einsatz einer MMU, daß in UNIX dieses Verfahren zur Aktivierung eines *Exception-Handlers* möglich ist.

Die Notwendigkeit der Manipulation des Laufzeitkontextes eines Prozesses ergibt sich nicht dadurch, daß die Aktivierung des *Exception-Handlers* asynchron zu den anderen Aktivitäten eines Prozesses geschehen muß. Mit jeder Beendigung eines *System-Calls* z.B., könnte, aufgrund eines bestimmten Resultates, der reaktivierte Prozeß eigenständig die Ausnahmesituation erkennen und den entsprechenden *Exception-Handler* aktivieren. Notwendig ist dieses Verfahren einzig deshalb, damit die Ausführung eines Prozesses, auch ohne explizit einen *System-Call* abgesetzt zu haben, unterbrochen werden kann. Das typische Beispiel hierfür ist der *Clock-Interrupt*, der den Ablauf der Zeitscheibe für den Prozeß signalisiert. In dieser Situation ist keine ausgezeichnete Stelle für den Prozeß vorgegeben, an der explizit die Ausnahmesituation eigenständig durch den Prozeß nachgefragt werden kann.

5.2.3. Das Modell zur Aktivierung von Exception-Handlern unter MOOSE

Unter MOOSE sind die mit UNIX vergleichbaren Mechanismen zur Aktivierung von *Exception-Handlern* nicht notwendig. Vielmehr ermöglicht das *Team*-Modell hierzu einen vollkommen anderen Ansatz, der zudem noch eine hardware-unabhängige Realisierung ermöglicht.

Wird das skizzierte Modell zur Aktivierung von *Exception-Handler* unter UNIX genauer betrachtet, so läßt sich ein wesentlicher Aspekt festhalten. Dieses Modell ermöglicht es prinzipiell, daß zwei verschiedene Aktivitäten innerhalb eines Prozesses "gleichzeitig" stattfinden können. Zum einen sind dies die Aktivitäten, die der Prozeß unter normalen Bedingungen realisiert, zum anderen sind dies die Aktivitäten, die der *Exception-Handler* veranlaßt, d.h. Aktivitäten, die der Prozeß unter normalen Bedingungen nicht realisieren würde. Zur Modellierung dieser verschiedenen Aktivitäten wird quasi der Laufzeitkontext eines Prozesses gemultiplext.

5.2.3.1. Interprozeßkommunikation zwischen Exception-Client und -Server

Unter MOOSE wird den verschiedenen Aktivitäten, wie sie im Zusammenhang mit UNIX skizziert worden sind, jeweils ein eigener Laufzeitkontext und ein eigener kontrollierender Prozeß zugeordnet. Der *Exception-Handler* unter MOOSE wird somit immer durch einen eigenen Prozeß kontrolliert und kann damit systemglobal eindeutig identifiziert sowie vollständig abgekapselt werden.

Das Signalisieren der Ausnahmesituation kann aufgrund der Zuordnung eines Prozesses für den *Exception-Handler* unter MOOSE erheblich flexibler gestaltet werden als unter UNIX. Grundlage dafür stellen die Mechanismen zur Interprozeßkommunikation des MOOSE-Kernels

[43] QNX läuft nur auf Personal-Computer und basiert nicht auf Hardware zur Adreßraumabbildung.

dar.

Mit diesen beiden Voraussetzungen – einen kontrollierenden Prozeß für den *Exception-Handler* und Interprozeßkommunikation zum Signalisieren einer Ausnahmesituation – läßt sich das Modell der Aktivierung von *Exception-Handler* unter MOOSE vervollständigen. Bild 5.4 skizziert hierzu den grundsätzlichen Aufbau des Modells.

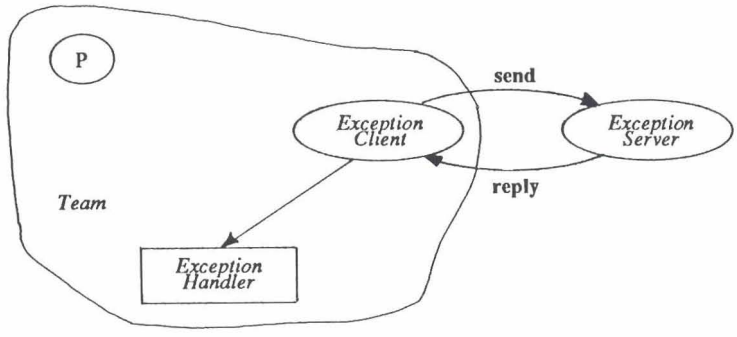

Bild 5.4: Modell zur Aktivierung eines *Exception-Handlers*

Der zusätzliche Prozeß zur Kontrolle des *Exception-Handlers* wird als *Exception-Client* bezeichnet. Die jeweilige Ausnahmesituation wird von einem bestimmten Systemprozeß signalisiert. Dieser Systemprozeß ist der sogenannte *Exception-Server*. Das Signalisieren einer Ausnahmesituation wird in elementarer Weise durch die Beendigung des *Rendezvous* zwischen *Exception-Client* und *Exception-Server* erreicht. Dem *Exception-Client* wird hierzu eine entsprechende *Reply-Message* zurückgegeben, die die aufgetretene Ausnahmesituation eindeutig beschreibt. Der *Exception-Server* operiert damit als *Administrator* im Sinne von [Gentleman 1981].

Mit diesem (zu UNIX innovativen) Modell ergeben sich wesentliche Schlußfolgerungen. Zum einen ist die Parameterisierung des *Exception-Handlers* sehr einfach zu vollziehen. Der *Exception-Client* entnimmt der *Reply-Message* bestimmte Informationen und übergibt diese dem *Exception-Handler* bei seiner prozeduralen Aktivierung. Zum anderen ist der *Exception-Client* selbst nicht reentrant. Das bedeutet, daß er solange die Annahme weiterer Ausnahmesituationen verzögert, wie der durch ihn aktivierte *Exception-Handler* nicht terminiert ist. Dadurch jedoch, daß aufgrund des *Team*-Konzeptes mehrere *Exception-Clients* in einem *Team* angesiedelt sein können, wäre die nebenläufige Behandlung von (nicht unbedingt unterschiedlichen) Ausnahmesituationen dennoch gegeben. Das bedeutet, daß geschachtelt aufgetretene Ausnahmesituationen – etwa ausgelöst durch den *Exception-Client* selbst – behandelt werden können.

Die in dem Modell skizzierte Kommunikationsbeziehung zwischen *Exception-Client* und *Exception-Server* ist ebenfalls von Bedeutung. Der *Exception-Server* ist unabhängig von den

Aktivitäten des *Exception-Clients*, was wesentlich ist für die Funktionalität des *Exception-Servers*. Diese Unabhängigkeit wird dadurch erreicht, indem der *Exception-Server*, im Zuge der Behandlung einer Ausnahmesituation, als *Rendezvous-Server* in Erscheinung tritt. Das Signalisieren der jeweiligen Ausnahmesituation, d.h., die Übermittlung der *Reply-Message* zum *Exception-Client*, wirkt, aufgrund des *Rendezvous*-Konzeptes, für den *Exception-Server* nicht blockierend. Der *Exception-Server* ist somit jederzeit bereit, zu anderen *Exception-Clients* insbesondere auch dieselbe Ausnahmesituation zu signalisieren.

5.2.3.2. Analogie zwischen Domain-Administrator und Exception-Server

Die Propagation von Ausnahmesituationen zur Benutzerebene stellt ein wesentliches Beispiel der Applikation von Prozeβdomänen dar. Die Ausnahmesituationen selbst sind einer bestimmten Domäne zugeordnet und demnach auch nur innerhalb derselben Domäne sinnvoll zu behandeln. Das bedeutet, daβ der *Exception-Client* der Domäne der jeweiligen Ausnahmesituation zugeordnet sein muβ, damit er mit der domänenspezifischen Behandlung der Ausnahmesituation beauftragt werden kann.

Die wesentliche Funktionalität des *Exception-Servers* entspricht damit der Funktionalität des *Domain-Administrators*, nämlich ein bestimmtes Systemereignis einer bestimmten Gruppe von Prozessen zuzustellen. Dennoch ergeben sich zwischen den beiden Systemprozessen bedeutende Unterschiede. Diese Unterschiede resultieren daher, daβ jeweils verschiedene Phasen eines Dienstes, als Ergebnis der Zuordnung eines Systemereignisses zu einer Gruppe von Prozessen, betrachtet werden müssen. Der *Exception-Server* durchläuft mit der Reaktivierung des *Exception-Clients* die terminale Phase eines Dienstes, der die domänenspezifische Behandlung von Ausnahmesituationen gestattet. Das *Rendezvous* zu dem *Exception-Client* wird beendet und der *Exception-Server* bleibt aktiv. Der *Domain-Administrator* dagegen startet die initiale Phase eines bestimmten Dienstes, der domänenglobal erbracht werden soll. Damit ist der Beginn eines *Rendezvous* verbunden, mit der Konsequenz der Blockierung des *Domain-Administrators*.

Trotz dieser Unterschiede zwischen *Domain-Administrator* und *Exception-Server* ist es zweckmäβig, die Funktionalitäten der beiden Systemprozesse von einem *Administrator* (und zwar vom *Domain-Administrator*) erbringen zu lassen. Damit wird die Grundlage geschaffen, Ausnahmesituationen als eine bestimmte Form von Systemdiensten zu betrachten, die einer domänenspezifischen Behandlung unterzogen werden können. Mit diesem Modell läβt sich somit z.B. vollständig die Signalbehandlung unter UNIX realisieren.

Die Problematik der Blockierung des *Domain-Administrators* bei der domänenglobalen Initiierung bestimmter Systemdienste, kann durch Mechanismen zur nichtblockierenden Interprozeβkommunikation umgangen werden. Hierzu stellt das *Team*-Konzept von MOOSE wiederum die dazu notwendigen Voraussetzungen zur Verfügung. Auf die Möglichkeiten zur nichtblockierenden Interprozeβkommunikation unter MOOSE wird in noch folgenden Kapiteln dieser Arbeit genauer eingegangen.

5.2.3.3. Propagation von Hardware-Traps zur Benutzerebene

Ausnahmesituationen werden unter MOOSE auf drei verschiedenen Ebenen erkannt. Zum einen ist dies die physikalische Ebene, repräsentiert durch den jeweiligen realen Prozessor. Zum anderen sind dies die Systemebene, repräsentiert durch die Systemprozesse, und die Benutzerebene, repräsentiert durch die Benutzerprozesse. Ein zusammenhängender Komplex von Benutzerprozessen bzw. Systemprozessen stellt jeweils einen abstrakten Prozessor [Schindler 1983] dar.

Der reale Prozessor erkennt bestimmte Ausnahmesituationen, die durch entsprechende Mechanismen des MOOSE-Kernels zur Systemebene propagiert werden können. Diese Ausnahmesituationen werden im folgenden als *Hardware-Traps* bezeichnet. Systemprozesse (*Trap-Server*) sind somit für die Behandlung der jeweiligen Ausnahmesituation verantwortlich.

Die von der physikalischen Ebene signalisierten *Hardware-Traps* lassen sich grundsätzlich in zwei Kategorien unterteilen:

a) transparente *Hardware-Traps*: Die bei der Ausführung eines bestimmten Prozesses erkannte und signalisierte Ausnahmesituation, wird vollständig und ohne Auswirkung auf den jeweiligen Prozeβ auf der Systemebene behandelt. Ein typisches Beispiel hierfür ist der sogenannte *Page-Fault* im Zusammenhang mit *Virtual Memory* Architekturen.

b) nicht transparente *Hardware-Traps*: Die bei der Ausführung eines bestimmten Prozesses erkannte und signalisierte Ausnahmesituation wird, mit Auswirkung auf den jeweiligen Prozeβ, auf der Systemebene behandelt. Diese Behandlung kann sich darin äuβern, den entsprechenden Prozeβ zu terminieren oder die Ausnahmesituation zur Benutzerebene zu propagieren. Ein typisches Beispiel hierfür sind *Hardware-Traps*, die im Zusammenhang mit der Interpretation bestimmter arithmetischer Operationen durch den realen Prozessor festgestellt worden sind.

Die domänenspezifische Behandlung eines *Hardware-Traps* bedingt somit, daβ sich ein *Trap-Server* zur Annahme des durch den MOOSE-Kernel propagierten Trap bereitstellt. Bild 5.5 skizziert das entsprechend erweiterte Modell zur Aktivierung eines *Exception-Handlers* unter MOOSE.

Zur Veranschaulichung, wie ein *Hardware-Trap* zur Benutzerebene propagiert wird, soll beispielhaft davon ausgegangen werden, daβ P für das Signalisieren eines Traps verantwortlich ist. Sobald der Trap von der physikalischen Ebene erkannt worden ist, wird dieser von dem realen Prozessor zur Kernelebene propagiert. Der *Gate-Handler* des entsprechenden *Trap-Gates* identifiziert den *Trap-Server* als behandelnden Systemprozeβ und propagiert den Trap weiter zur Systemebene. Der *Trap-Server* bekommt damit eine *Trap-Message* zugestellt. Der Absender dieser Nachricht ist hierbei der den Trap auslösende Prozeβ P.

Die Aktivierung des entsprechenden *Exception-Handlers* für den angezeigten *Hardware-Trap* wird erreicht, indem der *Trap-Server* den *Exception-Server* davon benachrichtigt. Das dazu in Anwendung gebrachte Protokoll zwischen dem *Trap-Server* und dem *Exception-Server* ist sehr elementar und ermöglicht dem *Exception-Server* die Identifikation des jeweils zu aktivierenden *Exception-Clients*. Hierzu dient die *process ID* von P und die Information, welcher *Exception-Client* im *Team* von P für die Behandlung des signalisierten Traps

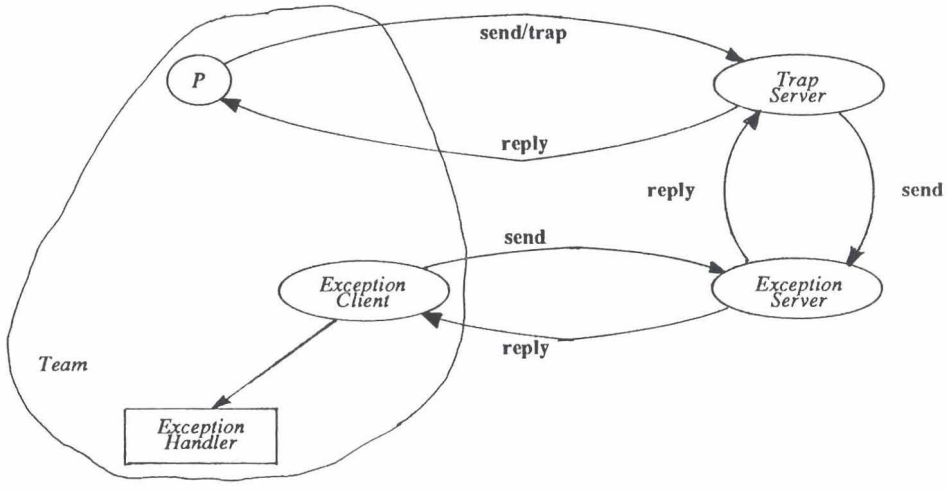

Bild 5.5: Propagation systemspezifischer Ausnahmesituationen

verantwortlich ist.

Der *Exception-Server* wird den *Trap-Server* nach Annahme der zu propagierenden Ausnahmesituation reaktivieren und damit dem *Trap-Server* die Möglichkeit geben, sich für die Annahme weiterer Traps wieder bereitzustellen. Die mit dem jeweiligen *Hardware-Trap* assoziierte Behandlungsstrategie auf der Systemebene wird durch den *Exception-Server* definiert. Sollte insbesondere die Wiederaufnahme der Ausführung von P möglich und erwartet sein, so würde dies der *Exception-Server* in der *Reply-Message* zum *Trap-Server* entsprechend kodieren. Der *Trap-Server* selbst wird in diesem Fall eine *Reply-Message* an P zurückgeben und damit für P das noch immer bestehende *Rendezvous* beenden. Diese *Reply-Message* enthält die durch den jeweiligen *Gate-Handler* aufgebaute *Trap-Message* und damit den Prozessorzustand von P.

5.2.3.4. Propagation von Software-Traps zur Benutzerebene

Mit dem bisher dargestellten Modell zur Propagation von *Hardware-Traps* zur Benutzerebene ist ebenso die Propagation allgemeinerer Ausnahmesituationen zu dieser Ebene möglich. Diese Ausnahmesituationen, als *Software-Traps* bezeichnet, sind durch Systemprozesse erkannt oder durch Benutzerprozesse signalisiert worden.

Für *Software-Traps*, die auf der Systemebene erkannt werden, ergeben sich die verschiedensten Ursachen. Von besonderer Bedeutung unter MOOSE sind jene *Software-Traps*, die die Termination eines Prozesses einem bestimmten Applikationskontext mitteilt. Die Propagation dieser Ausnahmesituationen zur Benutzerebene ermöglicht es, Benutzerprozesse davon zu benachrichtigen, daß die Abarbeitung der durch sie repräsentierten Applikation nicht weiterhin sichergestellt ist. Die entsprechende domänenspezifische Behandlung solcher Ausnahmesituationen könnte bedeuten, einen Prozeß zu erzeugen, der die applikationsabhängige Aufgabe des terminierten Prozesses übernimmt.

Ein weiterer *Software-Trap* von besonderer Bedeutung stellt das Signalisieren der Migration eines Dienstes dar. In diesem Fall könnte die domänenspezifische Behandlung dieser Ausnahmesituation vorsehen, die Identifikation des neuen diensterbringenden Prozesses des migrierten Dienstes domänenlokal bekanntzugeben. Dies würde in einfacher Weise durch die erneute Importierung des migrierten Dienstes bewerkstelligt werden können.

Allgemein wird es auch Benutzerprozessen ermöglicht, *Software-Traps*, sogenannte *user defined exceptions*, abzusetzen. Für den *Exception-Server* ergibt sich generell kein Unterschied, ob der *Software-Trap* durch Systemprozesse oder Benutzerprozesse angezeigt wird. Die Selektion und Aktivierung des jeweiligen *Exception-Clients* wird somit immer nach denselben Strategien erfolgen. Diese Strategien orientieren sich nach dem jeweils eingesetzten *Exception-Server* und können insbesondere zur Laufzeit eines MOOSE-Betriebssystems rekonfiguriert werden. Hierzu braucht lediglich der entsprechende *Exception-Server* ausgetauscht zu werden. Ein für MOOSE typischer *Exception-Server* repräsentiert die Funktionalitäten, wie sie mit der Signalbehandlung von UNIX 4.2 BSD vergleichbar sind.

5.2.3.5. Bedeutung der Exception-Clients für Applikationen

Wenn eine Behandlung von Ausnahmesituationen von einem bestimmten Benutzerprozeß bekannt gegeben worden ist, muß die Aktivierung des *Exception-Handlers* jederzeit möglich sein. Das bedeutet, daß die Aktivierung des entsprechenden *Exception-Clients* mit erhöhter Priorität für das entsprechende *Team* betrachtet werden muß.

Die besondere Problematik bei der Aktivierung eines *Excpeption-Clients* ergibt sich dadurch, daß für den Prozeß P die Zeitscheibe abgelaufen ist und damit nicht blockierend die Kontrolle abgegeben hat. Der MOOSE-Kernel wird Prozeß P dann wieder reaktivieren, wenn das *Team* des Prozesses P erneut zur Ausführung selektiert worden ist. Das bedeutet, daß unter normalen Umständen der aktive Prozeß eines *Teams* zwischen zwei aktiven Phasen desselben *Teams* nicht gewechselt wird. Die Ausführungsreihenfolge von Prozessen innerhalb desselben *Teams* wird damit durch den Ablauf der Zeitscheibe für das betreffende *Team* nicht beeinflußt. Die Deblockierung des *Exception-Clients* würde mit diesem Hintergrund somit nicht den gewünschten Erfolg haben, mit der Reaktivierung des *Teams* von P, sofort den *Exception-Handler* zu starten. Dazu müßte gerade der aktive Prozeß des betreffenden *Teams* ausgewechselt werden können, d.h. der *Exception-Client* und nicht P ist als aktiver Prozeß des *Teams* von P zu selektieren.

Diese Problematik wird unter MOOSE, anders als unter UNIX, in sehr einfacher Weise gelöst. Der *Exception-Client* wird als *Interrupt-Task* repräsentiert. Gerade die Eigenschaft dieser Prozesse ist es, daß deren Deblockierung immer dazu führen wird, bei der nächsten Aktivierung ihres *Teams* sofort die Kontrolle über das *Team* zu erhalten. Der *Exception-Handler* kann mit diesem Ansatz sofort gestartet werden, sofern das entsprechende *Team* erneut "*gescheduled*" worden ist. Die Blockierung des *Exception-Clients* auf den *Exception-Server*, in der Erwartung der Propagation einer weiteren Ausnahmesituation, führt zur Reaktivierung von P.

Ein anderer Aspekt, der die Bedeutung der *Exception-Clients* für Applikationen (und allgemein für das Gesamtsystem) hervorhebt, liegt in der Reaktion auf Ausnahmesituationen, die durch *Broadcasts* im System verbreitet werden. Hierbei ist insbesondere die Erzeugung

und Termination von Prozessen gemeint. Nur Systemprozessen ist es gestattet, sich an der Verarbeitung von *Broadcasts* beteiligen zu dürfen. Von diesen Prozessen wird allgemein ein höheres Maß an Sicherheit erwartet, als es von Benutzerprozessen der Fall ist. Würde nur ein Prozeß in der Kette von *Broadcast-Servern* sich nicht konform zu dem jeweiligen *Broadcast-Protocol* verhalten, wäre die systemglobale Verbreitung bestimmter Informationen nicht mehr gewährleistet. Privilegierte Systemprozesse würden abhängig werden von nicht privilegierten Benutzerprozessen, wenn die Benutzerprozesse sich an *Broadcasts* beteiligen dürften.

Dennoch sind bestimmte systemglobale Informationen über einzelne Prozesse auch für Benutzerprozesse bzw. Applikationssysteme sinnvoll auszunutzen. Von besonderer Bedeutung ist hierbei die Benachrichtigung über die Termination eines Prozesses. Zur Annahme solcher systemspezifischen Ausnahmesituationen sind die *Exception-Clients* geeignet. Sie erlauben es, Systemprozesse unabhängig von der Behandlung solcher Ausnahmesituationen durch Benutzerprozesse betrachten und realisieren zu können.

5.3. Emulation von Systemdiensten

Systemdienste werden immer im Zusammenhang zu einer bestimmten Familiengruppe stehen. Damit wird jeweils ein bestimmtes Mitglied der MOOSE-Betriebssystemfamilie zur Erbringung des angeforderten Systemdienstes verantwortlich sein.

5.3.1. Die Bedeutung des Abstraktionsniveaus von Systemdiensten

Das Konzept der MOOSE-Betriebssystemfamilie geht auf [Parnas 1975] zurück. Eine Betriebssystemfamilie wird demnach dadurch formuliert, indem Systemkomponenten derart entworfen werden, daß sie jeweils als Bausteine zur Realisierung verschiedener Betriebssysteme angewendet werden können. Diese Betriebssysteme repräsentieren dann jeweils Mitglieder einer Betriebssystemfamilie. Die Mitglieder zeichnen sich dadurch aus, daß sie auf gemeinsamen Entwurfsentscheidungen, die durch die Funktionalität der jeweiligen Systemkomponenten festgelegt worden sind, basieren [Habermann et al. 1976].

Die Modellierung einer Betriebssystemfamilie stellt eine wesentliche Anforderung des Systementwurfs in den Vordergrund. Diese Anforderung bedeutet, daß Entwurfsentscheidungen, die den flexiblen Einsatz von Systemkomponenten unterbinden könnten, soweit als möglich zurückzustellen sind, damit eine große Anzahl von Systemkomponenten zum Aufbau der Mitglieder der Betriebssystemfamilie zur Verfügung steht. Ein nach diesen Prinzipien entworfenes System wird ein wesentliches Merkmal aufweisen: das jeweilige Abstraktionsniveau [Schindler 1983] zwischen der benutzenden und der benutzten Systemkomponente wird sich nur minimal unterscheiden.

Unter MOOSE werden die zum Aufbau einer Betriebssystemfamilie signifikanten Systemkomponenten durch entsprechende Systemprozesse repräsentiert. Die einzelnen Mitglieder der MOOSE-Betriebssystemfamilie verstehen sich demnach als untereinander kooperierende und miteinander kommunizierende Prozesse. Die bisher dargestellten Systemprozesse – insbesondere der *Team-* und *Context-Administrator* – entsprechen den oben genannten Maximen zum Entwurf einer Betriebssystemfamilie. Diese beiden Prozesse

ermöglichen einerseits die Modellierung von Prozessen, gestattet jedoch andererseits den Aufbau spezieller Prozeβmodelle sowie Beziehungen zwischen den einzelnen Prozessen. Die durch diese Systemprozesse definierte Abbildung eines Prozesses auf einen bestimmten Adreβraum, läβt z.B. die Realisierung der Prozeβmodelle von UNIX, QNX und des *Team*-Modells von MOOSE direkt zu.

5.3.2. Abbildung von Systemdiensten

Die prinzipielle Aufgabe eines Emulators unter MOOSE besteht darin, das Bindeglied zwischen unterschiedlichen Betriebssystemdomänen darzustellen. Das bedeutet insbesondere, daβ Dienste einer Betriebssystemdomäne X durch Anwendung von Diensten des jeweiligen Emulators bzw. einer Betriebssystemdomäne Y erbracht werden. Für alle Dienste, die der Emulator nicht eigenständig erbringt, muβ durch den Emulator somit eine Abbildung von Systemdiensten realisiert werden.

Die durch den Emulator zu realisierende Abbildung wird umso leichter möglich sein, wenn die Dienste der Betriebssystemdomäne X hinreichend ähnlich sind zu Diensten der Betriebssystemdomäne Y. Jedem Dienst wird ein bestimmtes Abstraktionsniveau zugeordnet werden können. Damit bedeutet die hinreichende Ähnlichkeit von Diensten der Betriebssystemdomäne X und Y, daβ sich die Abstraktionsniveaus der einzelnen Dienste nur minimal unterscheiden. Die Aufgabe des Emulators ist es in diesem Zusammenhang, die unterschiedlichen Abstraktionsniveaus gleichartiger Dienste durch eine geeignete Abbildungsfunktion zu überbrücken. Die Komplexität dieser Funktion wird im wesentlichen von dem jeweiligen Abstand zwischen den bei der Abbildung zu betrachtenden Abstraktionsebenen abhängen.

5.3.3. Dienstabbildungen auf unterschiedlichen Ebenen eines MOOSE-Systems

Zur Abbildung eines Dienstes auf einen anderen Dienst oder einer Menge von anderen Diensten ist es notwendig, den jeweils abzubildenden Dienst erfassen zu können. Hierzu wird unter MOOSE davon ausgegangen, daβ Systemdienste über *System-Calls*, realisiert durch entsprechende Trap-Instruktionen, an der Schnittstelle zwischen Benutzer- und Systemebene in Anspruch genommen werden.

Für die Reaktion auf solche *System-Calls* stellt der MOOSE-Kernel geeignete Mechanismen zur Verfügung. Diese Mechanismen basieren darauf, daβ Traps von der Kernel- zur Systemebene propagiert werden, d.h. daβ sich ein *Trap-Server* an den Trap-Vektor des entsprechenden *System-Calls* anbinden kann. Je nach der Art des *Gates*, das zur Anbindung des *Trap-Servers* an den entsprechenden Trap-Vektors eingesetzt wird, erfolgt die Dienstabbildung auf unterschiedlichen Ebenen.

5.3.3.1. Dienstabbildung auf der Systemebene

Wird zur Anbindung des Emulators an den seiner Domäne zugeordneten Trap-Vektor ein *Trap-Gate* eingesetzt, muß der Emulator als Systemprozeß (der *Trap-Server*) eigenständig die Abbildung der zu emulierenden Systemdienste auf die jeweiligen Systemdienste des zugrunde liegenden Referenzsystems bewerkstelligen. Das bedeutet, daß die zu emulierenden Prozesse indirekt über den Emulator Dienste anderer Systemprozesse in Anspruch nehmen müssen. Bild 5.6 skizziert dieses Modell.

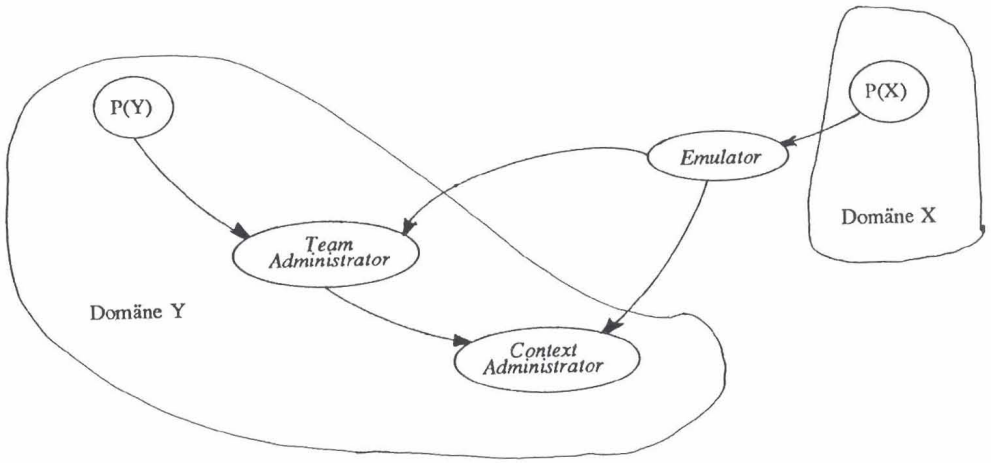

Bild 5.6: Der Emulator als Bindeglied zwischen Betriebssystemdomänen

Für die Emulation von Systemdiensten ergeben sich auf Grundlage dieses Modells zwei Konsequenzen. Diese Konsequenzen bedeuten im einzelnen:

1. der Emulator blockiert, wenn er zur Emulation Dienste anderer Systemprozesse in Anspruch nehmen muß. Der Emulator kann damit zeitweilig keine weiteren *System-Calls* annehmen;
2. der Emulator muß die im Zusammenhang mit der Dienstabbildung zu betrachtenden Dienste der anderen Systemprozesse mit den Zugriffsrechten der jeweils zu emulierenden Prozesse in Anspruch nehmen. Der Emulator muß somit jeweils in die Prozeßdomäne des zu emulierenden Prozesses wechseln.

Punkt 2 wird im wesentlichen durch das Konzept der Prozeßdomänen unter MOOSE ermöglicht. Die eigentliche technische Problematik bei der Abbildung von Systemdiensten, ergibt sich für den Emulator durch Punkt 1. Seine Blockierung würde eine streng sequenzielle Abarbeitung zu emulierender Systemdienste bedeuten. Dies ist insbesondere in den Fällen problematisch in denen die Emulation noch anstehender *System-Calls* direkt, d.h. ohne Inanspruchnahme von Diensten anderer Systemprozesse, erfolgen könnte. Zu emulierende *System-Calls* die diese Problematik besonders herausstellen, sind z.B. im Zusammenhang mit der Dateneingabe von zeichenorientierten Geräten zu betrachten. Die Blockierung des Emulators bei der Emulation solcher Dienste würde unbestimmt lange anhalten können.

Zur Lösung dieser Problematik auf der Systemebene von MOOSE bietet sich das *Team*-Konzept an. Der Emulator wird als *Team* bestehend aus mindestens zwei Prozessen betrachtet. Der zusätzliche Prozeβ in diesem *Team* übernimmt die Emulation von Ein-/Ausgabediensten, womit der ursprüngliche Prozeβ (der *Trap-Server*) des *Teams* alle anderen *System-Calls* emuliert. Diese Strukturierung läβt sich je nach gewünschter Funtkionalität eines Emulationssystems noch weiter fortführen, so daβ bestimmte logisch zusammenhängende *System-Calls* jeweils ihren eigenen Emulatorprozeβ zugeordnet bekommen.

5.3.3.2. Dienstabbildung auf der Kernelebene

Die durch bestimmte Systemprozesse (Emulatoren) ermöglichte Emulation bestimmter *System-Calls* besitzt einen wesentlichen Vorteil. Dieser Vorteil äuβert sich darin, daβ die Emulationssysteme auf Prozeβebene dynamisch rekonfigurierbar sind und damit ein hohes Maβ an Flexibilität erreicht wird. Der Nachteil bei diesem Ansatz ergibt sich im wesentlichen durch den Emulatorprozeβ selbst (bzw. durch eine Menge von Emulatorprozessen). Die Emulation durch einen Prozeβ bedingt immer einen zusätzlichen Prozeβwechsel. Dieser Prozeβwechsel wird dann keinen signifikanten Laufzeitverlust bedeuten, wenn der zu emulierende Dienst selbst sehr komplex ist (z.B. die Erzeugung eines neuen Prozesses). Für sehr elementare Dienste kann die Dauer des Prozeβwechsels jedoch in einem sehr ungünstigen Verhältnis zu der Laufzeit der Emulation stehen. Dies gilt insbesondere dann, wenn die Distanz zwischen den bei der Emulation zu betrachtenden Abstraktionsebenen sehr gering ist.

Die Abbildung von Diensten, die hinreichend ähnlich sind zu Diensten eines bestimmten Referenzsystems, kann in den Kernel migriert werden. Das wird dadurch erreicht, indem zur Anbindung eines Emulators an den seiner Domäne zugeordneten Trap-Vektor ein *Call-Gate* eingesetzt wird. In diesem Fall wird der Emulatorprozeβ nur für bestimmte Systemdienste in Anspruch genommen. Prinzipiell ergibt sich hierbei das in Bild 5.7 dargestellte Modell eines Emulationssystems.

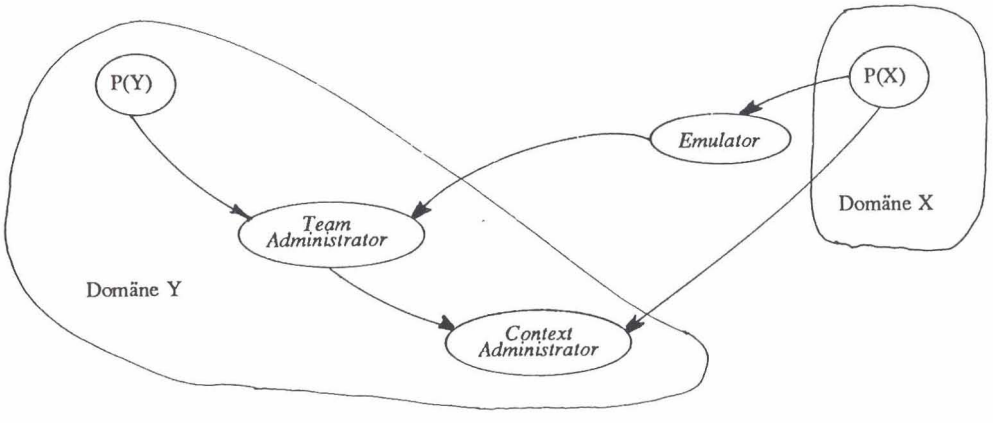

Bild 5.7: Sichtweise des Emulators bei der Dienstabbildung auf der Kernelebene

Prozeβ P(X) kann durch die über ein *Call-Gate* gesteuerte Emulation von Systemdiensten direkt Dienste anderer Betriebssystemdomänen in Anspruch nehmen. Der Emulator übernimmt nur die Abbildung komplexer Systemdienste. Dazu wird zwischen dem *Gate-Handler* des jeweiligen *Call-Gates* und dem Emulator ein bestimmtes Kommunikationsprotokoll seine Anwendung finden, damit P(X) über den *Gate-Handler* mit dem Emulator in Verbindung treten kann. Dieses Protokoll kann für jedes Paar (*Gate-Handler*, Emulator) unterschiedlich ausgelegt werden; es ist jeweils domänenspezifisch.

Die Problematik im Zusammenhang mit der Emulation von Diensten, bei denen der Emulator unbestimmt lange blockiert, kann mit einer auf der Kernelebene stattfindenden Dienstabbildung vermieden werden. Ist, ohne Zuhilfenahme des Emulators, auf der Kernelebene eine direkte Abbildung des zu emulierenden Dienstes möglich, so wird der zu emulierende Prozeβ selbst auf den diensterbringenden Systemprozeβ blockieren. Ist der Eingriff des Emulators notwendig, so kann, durch entsprechender Auslegung des Protokolls zwischen dem Emulator und dem *Gate-Handler*, der zu emulierende Prozeβ indirekt dazu veranlaβt werden, auf den diensterbringenden Prozeβ zu blockieren. In jedem Fall ist zu beachten, daβ alle Aktivitäten des *Gate-Handlers* unter der Identifikation des zu emulierenden Prozesses erfolgen. Damit wird die Blockierung zu emulierender Prozesse, nach erfolgter Abbildung des jeweiligen Dienstes, in "natürlicher" Weise durch das *Rendezvous* dieses Prozesses mit dem entsprechenden diensterbringenden Systemprozeβ ermöglicht.

5.3.3.3. Der Domain-Switch

Das Erfassen eines *System-Calls* erfolgt grundsätzlich dadurch, daβ sich ein *Trap-Server* als Emulator an den entsprechenden Trap-Vektor des realen Prozessors anbindet. Damit ist der *Trap-Server* oder nur der *Gate-Handler* in der Lage, über diesen Trap-Vektor signalisierte *System-Calls* zu emulieren. In diesem Zusammenhang kann jedoch dann ein Konflikt auftreten, wenn verschiedene Betriebssystemdomänen ihre *System-Calls* über denselben Trap-Vektor signalisiert erwarten. Bevor die Emulation des jeweiligen *System-Calls* erfolgen kann, muβ somit bekannt sein, welcher Betriebssystemdomäne dieser *System-Call* zuzuordnen ist.

Die Identifikation von Betriebssystemdomänen bei der Emulation von *System-Calls* erfolgt durch den sogenannten *Domain-Switch*. Erfolgt die Emulation ausschlieβlich auf der Systemebene, so wird der *Domain-Switch* als eigener Systemprozeβ betrachtet. Bild 5.8 zeigt ein einfaches Emulationssystem mit dem *Domain-Switch* als Systemprozeβ. Erfolgt die Emulation auf der Kernelebene, so wird der *Domain-Switch* in den jeweiligen *Gate-Handler* migriert. Die Funktionalität des *Domain-Switch* wird jedoch immer gleich bleiben.

Anhand der *family group ID* des zu emulierenden Prozesses, wird seine Betriebssystemdomäne bestimmt und daraufhin die entsprechenden domänenspezifischen Maβnahmen zur Emulation eines *System-Calls* ergriffen. Für die Kernelebene bedeutet dies die Aktivierung eines entsprechenden *Gate-Handlers*, für die Systemebene bedeutet dies das Weiterleiten der empfangenden *Trap-Message* zu dem jeweiligen Emulatorprozeβ.

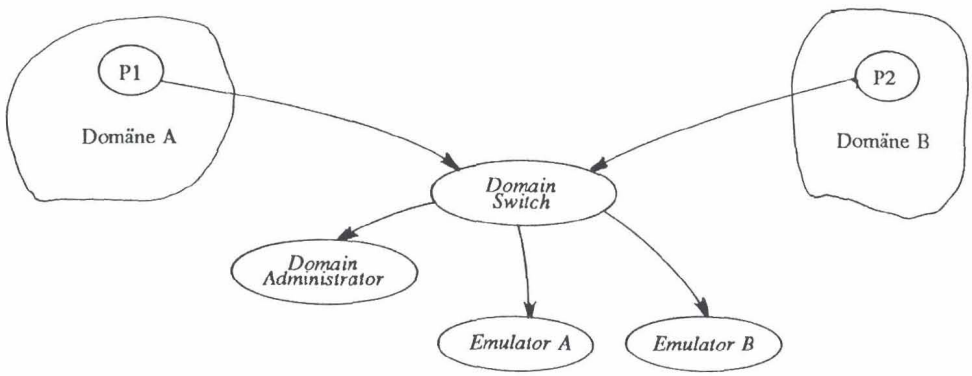

Bild 5.8: Die Einordnung des *Domain-Switch*

5.3.4. Ein Dienstprotokoll zur Kommunikation mit Systemprozessen

Die Abbildung zu emulierender Dienste auf Dienste anderer Systemprozesse, wird sich, auch für den günstigsten Fall – die Distanz zwischen beiden zu betrachtenden Abstraktionsebenen ist minimal –, nicht nur auf das einfache Durchreichen von Argumenten durch den Emulatorprozeß bechränken. Der logische Bezug der Argumente zu dem jeweils zu emulierenden Prozeß muß bestehen bleiben. Bild 5.9 soll diese grundlegende Problematik bei der Emulation von Systemdiensten allgemein verdeutlichen.

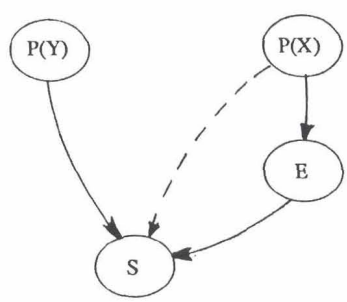

Bild 5.9: Grundlegende Sichtweise eines Emulators

Der Systemprozeß S erbringt Systemdienste, die von Prozessen unterschiedlicher Betriebssystemdomänen angefordert werden können. Prozeß P(X) kann die Dienste jedoch nur

über den Emulator E indirekt anfordern, wohingegen P(Y) auf die Dienste von S direkt zugreifen kann. Die Erbringung eines Dienstes durch Prozeβ S bedeutet, daβ P(Y) und E jeweils ein *Rendezvous* mit S eingehen müssen. Das wiederum bedeutet, daβ S aufgrund des *Rendezvous*-Konzeptes jeweils die Prozeβidentifikation von P(Y) und E, als dienstanfordernde Prozesse, mitgeteilt bekommt. Der angeforderte Dienst ist jedoch in jedem Fall immer im Zusammenhang mit P(X) und P(Y) zu sehen. Dies ist insbesondere für den Emulator E von Bedeutung, da S die mit dem Dienst verbundenen Integritätsüberprüfungen nicht mit den Zugriffsrechten von E, sondern mit denen von P(X) durchführen muβ.

Nicht nur die Überprüfung der Integrität einer Dienstanforderung ist von Bedeutung. Sollte der durch S erbrachte Dienst z.B. auf Informationen zurückgreifen müssen, die nicht direkt durch eine Nachricht übermittelt, sondern indirekt durch eine Referenz in den Adreβraum des betreffenden Prozesses (P(X) oder P(Y)) beschrieben werden, ergibt sich ein besonderer Konflikt. Hat E solch einen Dienst für P(X) in Anspruch genommen, so muβ jedoch nicht im Adreβraum von E der mit dem Dienst assoziierte Datenbestand angesiedelt sein. Die im Zusammenhang mit dem Dienst zu betrachtende Adresse wäre dem Adreβraum von P(X) zuzuordnen.

Zur Lösung dieses Konfliktes bieten sich verschiedene Verfahren an. Verfahren jedoch, die auf bestimmte hardware-technische Voraussetzungen des realen Prozessors basieren – z.B. *Shared-Memory* oder *capabilities* – scheiden für MOOSE aus Gründen der Portabilität aus. Vielmehr wird zur hardware-unabhängigen Realisierung dieses Konfliktes auf ein bestimmtes Dienstprotokoll zwischen allen Systemprozessen unter MOOSE zurückgegriffen.

Das Dienstprotokoll zur Kommunikation mit den MOOSE-Systemprozessen stellt bestimmte Anforderungen an das Format der jeweils auszutauschenden Nachrichten. Dieses Format strukturiert die Nachrichten in einen protokollspezifischen und einen dienstspezifischen Bereich. Tabelle 5.2 zeigt die Grobstruktur einer *System-Call Message* unter MOOSE.

system-call code	*process ID*	*team ID*	*process attributes*	*system-call arguments*
1	2	3	4	5

Tabelle 5.2: Format einer *System-Call Message*

Die Bedeutung der einzelnen Bereiche der *System-Call Message* sind im einzelnen:

1 Die Kodierung des *System-Calls* zur Anforderung des benannten Dienstes bzw. die Kodierung des Resultates der Diensterbringung. Dieses Resultat bezieht sich jedoch nur auf das Dienstprotokoll und gibt Aufschluβ über die erfolgreiche oder nicht erfolgreiche Erbringung eines Dienstes.

2 Die Identifikation des Prozesses, für den der Dienst erbracht werden soll. Die *process ID* ermöglicht die Identifikation des Adreβraumes, mit dem bestimmte Argumente des Dienstes assoziiert werden müssen.

3 Die Identifikation des *Teams*, das zur Integritätsüberprüfung des Dienstes betrachtet werden soll.

4 Die Prozeβattribute des Prozesses, für den der Dienst erbracht werden soll.

5 Die Argumentenliste des Dienstes.

Der dienstspezifische Anteil der Nachricht ist im Bereich 5 enthalten und wird vom Dienstprotokoll vollständig transparent betrachtet. Der protokollspezifische Anteil der Nachricht ist in den Bereichen 1 bis 4 enthalten.

Auf Grundlage des Dienstprotokolls von MOOSE ist es dem Emulatorprozeβ E möglich, beliebige Dienste im Namen von P(X) durch S erbringen lassen zu können. Desweiteren bedeutet das *Rendezvous*-Konzept von MOOSE, daβ der Emulatorprozeβ E als *Rendezvous-Client* dem *Rendezvous-Server* S gegenübersteht. Damit wird die Beendigung der Diensterbringung von S an E dadurch mitgeteilt, daβ das *Rendezvous* beendet wird. Der Emulatorprozeβ besitzt damit die Möglichkeit, entsprechend der Konventionen der jeweiligen Betriebssystemdomäne, den zu emulierenden Prozeβ P(X) über Erfolg oder Miβerfolg des *System-Calls* zu benachrichtigen.

5.4. Dynamische Rekonfiguration

Zur Realisierung des Konzeptes einer Familie von Betriebssystemen ist ein Systementwurf Voraussetzung, der auf bestimmten funktionalen Einheiten basiert. Die funktionalen Einheiten werden dabei durch einzelene Systemkomponenten repräsentiert. Die Schwierigkeit auf der konzeptionellen Ebene besteht bei dem Systementwurf darin, die Systemkomponenten so zu gestalten, daβ sie ausschlieβlich als Bausteine eines gröβeren Systemkomplexes eingesetzt werden können – nur mit dieser Forderung kann der oben genannten Maxime entsprochen werden. Die Funktionalitäten dieser Systemkomponenten müssen so ausgelegt sein, daβ ein flexibler Einsatz möglich ist. Insbesondere sind die Systemkomponenten so zu bestimmen, daβ die durch sie zur Verfügung gestellten Funktionalitäten auch vollständig zum Tragen kommen. Das bedeutet, eine Systemkomponente wäre dann aufzusplitten (in zwei oder mehrere Einheiten), wenn für ein bestimmtes Einsatzgebiet ersichtlich ist, daβ nicht alle zur Verfügung gestellten Funktionalitäten genutzt werden.

Wie klein die Systemkomponenten gefaβt werden können wird demzufolge durch die sie benutzenden Komponenten eines Systems bestimmt. Dies jedoch bedeutet gleichzeitig, daβ von vornherein kein "optimaler" Systementwurf vorgegeben werden kann, da nicht alle möglichen Applikationsprofile des Systems von vornherein bekannt sind. Die konsequente Modularisierung eines Systems ist demnach nur als die minimale Voraussetzung anzusehen, eine Familie von Betriebssystemen modellieren zu können. Von zentraler Bedeutung wird es sein, ein gewisses Maβ an Dynamik mit dem System zu verbinden, damit eine Anpassung an bestimmte Applikationsprofile möglich ist. Das bedeutet, daβ es gewährleistet sein muβ, Systemkomponenten in ein bestehendes System nachträglich zu integrieren bzw. aus dem System herauszunehmen. Mehr noch, die Repräsentation bestimmter Systemkomponenten müβte sich dynamisch ändern können, so daβ eine Systemkomponente in zwei oder mehrere Einheiten nachträglich aufgesplittet werden kann.

Die Integration, das Entfernen und die Änderung der Repräsentation von Systemkomponenten zur Laufzeit des durch sie aufgebauten Systems, bedeutet unter MOOSE die dynamische Rekonfiguration dieses Systems. Die Kernproblematik besteht hierbei darin, daß die dynamische Rekonfiguration transparent für alle anderen Aktivitäten des bestehenden Systems geschehen muß.

5.4.1. Prozesse als rekonfigurierbare Komponenten eines Systems

Die dynamische Rekonfiguration eines Systems wird unter MOOSE auf der Ebene von Prozessen betrachtet. Dieser Ansatzpunkt entspricht generell dem Entwurfsprinzip von RC4000 [Hansen 1970] und wird für MOOSE durch zwei grundlegende Aspekte motiviert.

Der erste Aspekt betrachtet die Portabilität eines auf Prozeßebene dynamisch rekonfigurierbaren Systems. Das Prozeßmodell sowie die verschiedenen Varianten zur Erzeugung von Prozessen unter MOOSE lassen sich auf jedem realen Prozessor verwirklichen. Im Gegensatz zu FAMOS [Habermann et al. 1976] und DAS [Isle et al. 1977] wird keine spezielle Hardware benötigt, um die Repräsentation von Systemkomponenten, die unter MOOSE durch Prozesse abgekapselt sind, zur Laufzeit ändern zu können.

Der zweite Aspekt betrachtet die These, daß Prozesse das adäquate Hilfsmittel darstellen, eine Familie von Betriebssystemen dynamisch zu modellieren. Die einzelnen Systemkomponenten der MOOSE-Betriebssystemfamilie sind komplex genug, um durch Prozesse abgekapselt zu werden. Die Komplexität einer Systemkomponente ist jedoch nur ein maßgeblicher Faktor dafür, die jeweiligen Komponenten durch einen Prozeß abzukapseln. Für bestimmte Hardware-Architekturen, bei denen der durch den realen Prozessor definierte logische Adreßraum eines Prozesses stark eingeschränkt ist, wird es in diesem Zusammenhang notwendig sein, einen bestimmten Systemkomplex in mehrere Prozesse aufzusplitten. Dies wird um so mehr an Bedeutung gewinnen, wenn umfangreiche Applikationen zur Ausführung gebracht werden sollen.

5.4.2. Typische Funktionalitäten austauschbarer Systemkomponenten

Die Systemkomponenten der MOOSE-Betriebsystemfamilie stellen komplexe abstrakte Datentypen dar, die wesentliche Teilfunktionalitäten eines typischen Betriebsysstems erbringen. Objekte dieser abstrakten Datentypen werden jeweils durch eigene Prozesse kontrolliert. Mit der Ausführung dieser Prozesse werden bestimmte Funktionalitäten eines Mitglieds der Betriebssystemfamilie erbracht.

Die Anwendung von Prozessen zur dynamischen Rekonfiguration eines Systems und zum Aufbau einer Bertriebssystemfamilie unterscheidet MOOSE konkret von den entsprechenden Systemen wie DAS [Isle et al. 1977] und FAMOS [Habermann et al. 1976]. Die mit diesen Systemen auf prozeduraler- bzw. Modulebene ermöglichte dynamische Rekonfiguration ist zu detailliert, als daß sie den adäquaten Ansatz zum Aufbau einer Betriebssystemfamilie darstellt. Systemkomponenten einer Betriebssystemfamilie werden sich dadurch auszeichnen, daß sie komplexe Funktionalitäten repräsentieren um z.B.

- verschiedene Prozeβmodelle zu verwirklichen;
- verschiedene Filesysteme auf demselben blockorientierten Ein-/Ausgabesystem zu realisieren;
- unterschiedliche Strategien zur blockorientierten Ein-/Ausgabe zu ermöglichen, als da sind *caches*, bestimmte (geräteoptimierte) Sortierverfahren, asynchrone Ein-/Ausgabeschnittstellen, Hochvolumenverkehr (*multi sector I/O*) usw.;
- unterschiedliche Strategien zur zeichenorientierten Ein-/Ausgabe zu ermöglichen, als da sind *Line-Editing*, *Character-Mapping*, *Window-Management*, asynchrone Ein-/Ausgabeschnittstellen usw.;
- erweiterte Mechanismen zur (UNIX-*like*) Interprozeβkommunikation wie *pipes* und *sockets* zur Verfügung zu stellen;
- netzwerktransparente Kommunikationsdienste zu ermöglichen;
- Bindeglieder zwischen verschiedenen Betriebssystemdomänen in Form von Emulatoren darzustellen.

Diese Funktionalitäten können aufgrund ihrer Komplexität software-technisch jedoch nur durch eine Menge von Moduln zur Verfügung gestellt werden. Die einzelnen Moduln definieren jedoch nicht für sich allein die von den Systemkomponenten zur Verfügung gestellten Funktionalitäten. Jedes Modul erbringt jeweils bestimmte Teilfunktionalitäten dieser Systemkomponenten. Die auf Modulebene durchgeführte dynamische Rekonfiguration eines Systems würde demnach nur die Repräsentation eines bestimmten Teils einer Systemkomponente ändern und nicht notwendigerweise zum vollständigen Austausch einer Systemkomponente führen. Die Systemkomponente als Gesamtheit betrachtet, und nicht die einzelnen in ihr enthaltenen Moduln, stehen demnach unter MOOSE zum Aufbau einer Betriebssystemfamilie im Vordergrund.

Die dynamische Rekonfiguration eines Systems kann sicherlich auch durch Anwendung der in FAMOS und DAS zugrunde liegenden Techniken erfolgen. Nur stellen Prozesse in dieser Hinsicht allgemein die flexiblere Lösung dar, wenn es darum geht, die Funktionalitäten und nicht Teile (Moduln) der zur Erbringung der Funktinonalitäten notwendigen Systemkomponenten zur Laufzeit auszutauschen. Insbesondere bedeutet die Anwendung von Prozessen als Hilfsmittel zur dynamischen Rekonfiguration eines Systems, daβ keine spezielle Hardware-Unterstützung, vergleichbar mit der von FAMOS, als Voraussetzung gegeben sein muβ.

5.4.3. Mechanismen zur Modellierung einer Betriebssystemfamilie

Die Mechanismen zur Modellierung einer Betriebssystemfamilie unter MOOSE sind vereinzelt bereits vorgestellt worden. An dieser Stelle sollen diese Mechanismen im Zusammenhang betrachtet werden.

Den Ausgangspunkt für die Modellierung einer Betriebssystemfamilie stellen Prozesse dar, bei deren Ausführung bestimmte Funktionalitäten erbracht werden. Die mit einem Prozeβ zusammenhängende "natürliche" Dynamik – Prozesse können erzeugt und zerstört werden –

stellt die Grundlage zur dynamischen Rekonfiguration einer Betriebssystemfamilie dar.

Die Prozesse kooperieren untereinander und kommunizieren miteinander, so daß ein Mitglied der Betriebssystemfamilie durch einen Komplex von solchen Prozessen realisiert werden kann. Die geforderte Dynamik eines solchen Komplexes bringt es jedoch mit sich, daß die Prozesse, obgleich sie demselben Prozeßkomplex zugeordnet sind, voneinander abstrahieren müssen. Nur so kann die Transparenz der dynamischen Rekonfiguration des Systems gewährleistet bleiben. Die Abstraktion von den Prozessen erfolgt durch die Benennung der durch sie zur Verfügung gestellten Funktionalitäten. Das bedeutet, alle Prozesse zum Aufbau der Betriebssystemfamilie stellen bestimmte Dienste zur Verfügung, womit eine Abstraktion von dem jeweiligen diensterbringenden Prozeß erreicht wird. Ein Prozeß kann hierzu mehrere verschiedene Dienste zur Verfügung stellen. Ebenso können sich mehrere Prozesse die Erbringung dieser Dienste teilen. Hierbei wird in jedem Fall die eindeutige Zuordnung von einem Dienst zu genau einem Prozeß stattfinden. Dem jeweiligen dienstanfordernden Prozeß ist diese Zuordnung von Diensten zu einem Prozeß oder mehreren Prozessen verborgen. Anhand des Dienstnamens wird der jeweilige diensterbringende Prozeß von einem dienstanfordernden Prozeß logisch adressiert.

Die unterste Ebene schließlich ermöglicht die Kommunikation zwischen den einzelnen Prozessen. Die dazu vom MOOSE-Kernel zur Verfügung gestellten Mechanismen zeichnen sich hierbei mehr durch ihre Laufzeiteffizienz als durch ihre Funktionalität aus. Die Mechanismen zur Realisierung von *Message-Passing* unter MOOSE entsprechen denen von THOTH und insbesondere QNX. Der Entwurf des MOOSE-Kernels ist jedoch davon geprägt, daß in allen Fällen laufzeiteffizientere Interprozeßkommunikationsmechanismen als bei THOTH und QNX zur Verfügung gestellt werden. Diese Tatsache läßt den prozeßorientierten und auf *Message-Passing* basierenden Systementwurf von MOOSE, und damit die Modellierung von Mitgliedern der MOOSE-Betriebssystemfamilie, sehr attraktiv werden.

Eine andere wesentliche Funktionalität des MOOSE-Kernels, zur Unterstützung der dynamischen Rekonfiguration eines Systems, stellen die Mechanismen zur Propagation von Traps/Interrupts an entsprechende *Server*-Prozesse dar. Diese Mechanismen ermöglichen es erst, den prozeßorientierten Entwurf bis zur letzten Konsequenz, der Behandlung von Interrupts durch eigens dafür eingerichtete Prozesse, durchführen zu können. Damit können alle hardware-nahen Aktivitäten eines Betriebssystems in entsprechende Systemkomponenten zusammengefaßt und durch Prozesse abgekapselt werden.

Auf Grundlage dieser kurz skizzierten und grundlegenden Mechanismen von MOOSE – Prozesse, Dienste, *Message-Passing* und die Propagation von Traps/Interrupts – kann die Modellierung einer Betriebssystemfamilie in einfacher Weise durch einen entsprechenden Systemprozeß kontrolliert werden. Dieser Systemprozeß, der *Loader*, erkennt Dienstabhängigkeiten zwischen bestimmten Prozessen und sorgt dafür, daß vor Ausführung einer bestimmten Applikation alle notwendigen diensterbringenden Prozesse zur Verfügung stehen. Das bedeutet z.B. einen bestimmten Emulatorprozeß dann zur Ausführung zu bringen, wenn er zur Abarbeitung von Applikationsprogrammen notwendig ist.

5.4.4. Mechanismen zur Rekonfiguration einer Betriebssystemfamilie

Die Erweiterung einer Betriebssystemfamilie durch zusätzliche Systemkomponenten zur Laufzeit, entspricht unter MOOSE vollständig dem Aufbau von Applikationssystemen. Neue Systemkomponenten werden integriert, indem die jeweils zur Abkapselung der Systemkomponenten dienenden Systemprozesse erzeugt und untereinander in Verbindung gebracht werden. Ebenso erfolgt die Verkleinerung der Betriebssystemfamilie zur Laufzeit nach denselben Prinzipien, wie der Abbau eines Applikationssystems. Die entsprechenden Systemprozesse werden aus dem Familienkomplex herausgelöst, wenn die Dienste der ihnen zugeordneten Systemkomponenten nicht mehr in Anspruch genommen werden.

Eine verstärkt auftretende Problematik ergibt sich dann, wenn ein bestehendes System dahingehend rekonfiguriert werden soll, um die Repräsentation bestimmter Systemkomponenten zur Laufzeit zu ändern. In [Isle et al. 1977] ist die generelle Problematik dieses Verfahrens, mit *Replugging* bezeichnet, dargestellt. Diese Problematik ist unabhängig davon, mit welchen Mechanismen *Replugging* realisiert werden kann, d.h. ob auf Grundlage einer *capability based architecture*, wie in DAS, oder ob auf Grundlage eines bestimmten Prozeβmodells, wie in MOOSE, die Repräsentation von Systemkomponenten veränderbar ist.

Replugging muβ vollkommen transparent für alle anderen Aktivitäten des zu rekonfigurierenden Systems erfolgen. Unter MOOSE bedeutet dies insbesondere, daβ die von bestimmten Systemprozessen zur Verfügung gestellten Dienste nach erfolgter Rekonfiguration weiterhin angeboten werden müssen. Hierbei gilt jedoch nicht, daβ der neu integrierte ("*repluggte*") Systemprozeβ nur genau dieselben Dienste verfügbar macht, wie sie der ausgetauschte Prozeβ ursprünglich zur Verfügung gestellt hatte. Der neu integrierte Prozeβ kann durchaus mehr Funktionalität beeinhalten und diese z.B. durch eine größere Anzahl von Diensten zur Verfügung stellen. Die Transparenz im Zusammenhang mit der dynamischen Rekonfiguration eines Systems bedeutet damit, daβ die Funktionalität der von der Rekonfiguration betroffenen Systemprozesse nicht geringer werden darf. Die Funktionalität des betreffenden Systemprozesses muβ mindestens eine Obermenge der ursprünglich zur Verfügung gestellten Funktionalitäten darstellen.

5.4.4.1. Adoption von Diensten

Da die Identifikation der Systemprozesse anhand der von ihnen zur Verfügung gestellten Diensten erfolgt, ist es naheliegend, die dynamische Rekonfiguration auf der Dienstebene anzusetzen. Dienste können von dazu ausgezeichneten Prozessen adoptiert werden, womit die Migration des betreffenden Dienstes verbunden ist. Die Konsequenz der Adoption ist es, daβ dem bisherigen diensterbringenden Prozeβ der betreffende Dienst entzogen wird.

Die Notwendigkeit der Adoption eines Dienstes ergibt sich dann, wenn eine Systemkomponente aufgesplittet werden soll und der diese Systemkomponente abkapselnde Prozeβ mehr als einen Dienst zur Verfügung stellt. Die Aufsplittung der Systemkomponente wird damit gleichzeitig die Aufsplittung des diese Systemkomponente abkapselnden Prozesses zur Folge haben.

Die Adoption eines Dienstes bedeutet nicht, daβ eine bestimmte Funktionalität anders benannt wird. Vielmehr ändert sich die Zuordnung der benannten Funktionalität zu einem

Prozeß. Hierbei ist jedoch zu beachten, daß nach erfolgter Adoption von mindestens zwei Prozessen die gleiche Funktionalität erbracht wird: von dem ursprünglichen diensterbringenden Prozeß und von dem dienstadoptierenden Prozeß. Dennoch führt die Adoption eines Dienstes nicht zu einer Zweideutigkeit der Identifikation und Adressierung eines diensterbringenden Prozesses. Die Adoption eines Dienstes erfolgt unteilbar durch den *Service-Administrator* – dieser stellt gerade einen sequentiell ablaufenden Prozeß dar. Damit ist die Eindeutigkeit der Zuordnung eines Dienstes zu einem Systemprozeß nach erfolgter Adoption weiterhin gewährleistet.

Jede nach erfolgter Adoption ermöglichte Identifikation eines Systemprozesses anhand eines adoptierten Dienstes, adressiert einen neuen diensterbringenden Prozeß. Alle bereits vor der Adoption getroffenen Zuordnungen von Diensten zu Prozessen, behalten weiterhin ihre Gültigkeit. Nur der betreffende Dienst wird adoptiert, der ursprüngliche diensterbringende Prozeß kann weiterhin adressiert werden. Dies ist ein wesentlicher Aspekt, da mit der Adressierung dieses Prozesses im Zusammenhang mit der Erbringung eines bestimmten Dienstes, insbesondere auch nach erfolgter Adoption, weiterhin zu rechnen ist. Die Ursache liegt darin, daß die *processID* dieses Prozesses, aufgrund der Importierung des nachträglich adoptierten Dienstes, lokal in dem dienstanfordernden Prozeß gehalten wird.

Die Adoption eines Dienstes kann im System global bekannt gemacht werden. Hierfür wird die Migration dieses Dienstes als systemspezifische Ausnahmesituation domänenspezifisch verteilt. Damit wird jedem Prozeß der dem ursprünglichen diensterbringenden Prozeß zugeordneten Prozeßdomäne die Möglichkeit gegeben, auf diese systemspezifische Ausnahmesituation reagieren zu können. Die Behandlung dieser Ausnahmesituation würde bedeuten, die Zuordnung einer *processID* zu dem adoptierten Dienst erneut zu treffen und damit die Identifikation des aktuellen diensterbringenden Prozesses zu erhalten und lokal zu vermerken.

5.4.4.2. Adoption von Rendezvous-Server

Die bei der Adoption eines Dienstes auftretende Situation, daß die mit dem Dienst assoziierte Funktionalität durch zwei Prozesse erbracht wird, kann vermieden werden. Hierzu wird, anstatt eines einzelnen Dienstes, der gesamte diensterbringende Prozeß (der *Rendezvous-Server*) adoptiert. Für diese Maßnahme stellt der MOOSE-Kernel Mechanismen zur Verfügung, die es erlauben, daß ein Prozeß repräsentativ für einen anderen Prozeß alle Nachrichten empfangen kann. Mit diesem Mechanismus des Kernels wird die Adoption aller Dienste des betroffenen Prozesses erreicht. Damit kann der so adoptierte Prozeß aus dem bestehenden Systemkomplex herausgelöst und zerstört werden. Bild 5.10 skizziert die Adoption eines diensterbringenden Prozesses und die gleichzeitige Aufsplittung dieses Prozesses.

Die Ausgangssituation in diesem Beispiel stellt die Beziehung zwischen Prozeß U1, als *Service-User*, und Prozeß P, als *Service-Provider*, dar. Hierbei wird erstens ein Dienst von P adoptiert, woraufhin die Aufsplittung von P in P1 und P2 resultiert, und zweitens wird P selbst von dem *Service-Replugger* adoptiert, damit P aus dem System herausgenommen werden kann.

Die zentrale Aufgabe bei der dynamischen Rekonfiguration eines Systems übernimmt der *Service-Replugger*. Dieser Systemprozeß nimmt alle ursprünglichen an P gerichteten Nachrichten an. Diese Nachrichten kodieren die von P jeweils zur Verfügung gestellten

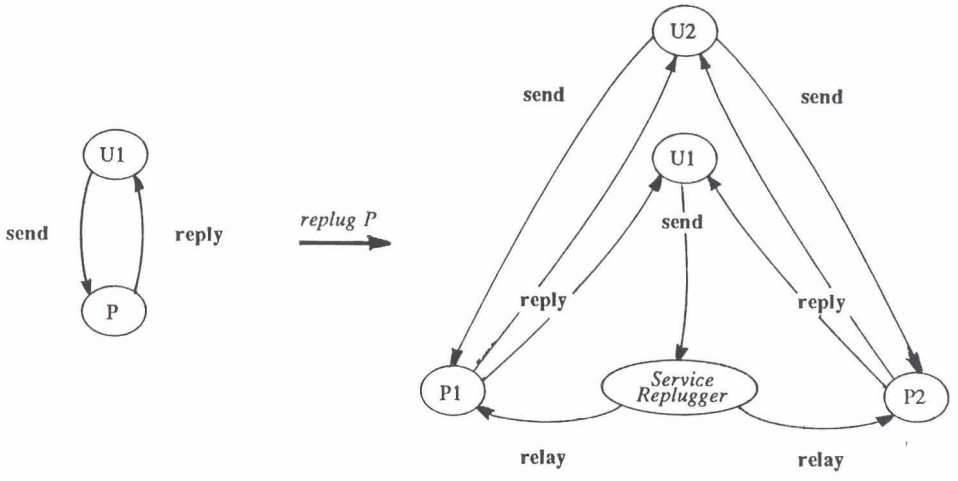

Bild 5.10: Adoption und Aufsplittung eines Prozesses

Dienste. Da P in zwei diensterbringende Prozesse P1 und P2 aufgesplittet worden ist, werden die ursprünglich von P zur Verfügung gestellten Dienste auf P1 und P2 verteilt. P1 und P2 beziehen sich jeweils auf dasselbe Dienstprotokoll wie es mit P seine Gültigkeit besaβ. Da sich damit insbesondere für U1 dieses Protokoll – und damit das Schnittstellenverhalten zu den jeweils diensterbringenden Prozessen– nicht verändert hat, erfolgte die Adoption der Dienste von P, sowie von P selbst, ohne Auswirkungen auf U1.

5.4.5. Die Transparenz der dynamischen Rekonfiguration eines Systems

Die dynamische Rekonfiguration auf Grundlage der Adoption von Diensten bzw. Prozessen ist auf technischer Ebene transparent für die jeweiligen dienstanfordernden Prozesse (in dem skizzierten Beispiel durch U1 repräsentiert). Dennoch kann sich, wie in [Parnas, Siewiorek 1972] beschrieben, durch den *Service-Replugger* ein Verlust an Transparenz der Repräsentation von Diensten ergeben. Hierbei wären jedoch einzig nur die durch den zusätzlichen Prozeβwechsel bedingten höheren Laufzeiten bei der Erbringung eines Dienstes die Ursache. Diese höheren Laufzeiten könnten den Ablauf zeitkritischer Applikationen, die u.a. durch U1 kontrolliert werden, stören.

Die vollkommene Transparenz der Adoption eines Prozesses wird erreicht, wenn die Adoption der von diesen Prozeβ zur Verfügung gestellten Dienste als Ausnahmesituation im System verteilt werden. In diesem Fall könnte U1 auf die dynamische Rekonfiguration diensterbringender Prozesse im Zuge einer domänenspezifischen Ausnahmebehandlung reagieren. Diese Behandlung würde üblicherweise dazu führen, daβ alle nachfolgenden Dienste direkt von P1 oder P2 und nicht indirekt über den *Service-Replugger* in Anspruch genommen werden. Der *Service-Replugger* ist nur für die Prozesse notwendig, die auf solche Ausnahmesituationen nicht reagieren, denen jedoch auch nicht die weitere Anforderung der Erbringung von Diensten verwehrt werden kann.

Das Signalisieren der Migration eines Prozesses bzw. Dienstes als eine systemspezifische Ausnahmesituation, würde, nach erfolgter Migration durch entsprechende Dienste des *Service-Administrators*, der *Service-Replugger* selbst übernehmen. Hierfür wird sich der *Service-Replugger* entsprechender Funktionalitäten des *Domain-Administrators* bedienen. Da ein Dienst an einen Prozeß (lose) gebunden und dieser Prozeß eindeutig einer bestimmten Prozeßdomäne zugeordnet ist, ist gleichzeit der jeweilige Dienst dieser Prozeßdomäne zuzuordnen. Damit kann mit Hilfe des *Domain-Administrators* jedem Prozeß dieser Domäne die Migration eines Dienstes als Ausnahmesituation zugestellt werden.

5.5. Nichtblockierende Kommunikationsmechanismen

Die Mechanismen des MOOSE-Kernels zur Interprozeßkommunikation wirken blockierend auf die jeweils anwendenden Prozesse. Dafür ist jedoch die Realisierung dieser Mechanismen sehr elementar und ebenso laufzeiteffizient. Mit Hilfe des *Team*-Konzeptes, der Ausführungsstrategien und der Interprozeßkommunikationsmechnismen des MOOSE-Kernels lassen sich jedoch in elementarer Weise auch nichtblockierende Mechanismen zur Interprozeßkommunikation realisieren. Der Kernansatz der nachfolgend vorgestellten Kommunikationsmodelle basiert darauf, daß die Verarbeitung einer Nachricht unabhängig von ihrem Empfang stattfindet und daß die sich daraus ergebenden funktional unterschiedlichen Komponenten jeweils durch eigene Prozesse, angesiedelt innerhalb desselben *Teams*, kontrolliert werden.

5.5.1. Trennung zwischen Annahme und Verarbeitung von Nachrichten

Die Blockierung eines Prozesses während der Verarbeitung von übermittelten Nachrichten durch einen anderen Prozeß, ist nicht primär durch die jeweiligen Kommunikationsmechanismen gegeben. Dies gilt allgemein für *Message-Passing* Systeme, die zum MOOSE-Kernel vergleichbare Kommunikationsmechanismen anbieten (hierbei insbesondere THOTH [Cheriton 1982] und QNX [Quantum 1984]). Blockierend wirken die Mechanismen erst dann, wenn der jeweilige Kommunikationspartner nicht bereit ist, eine Nachricht zu übernehmen. Das bedeutet, die Blockierung eines Prozesses ist vom jeweiligen Laufzeitverhalten seines Kommunikationspartners abhängig.

Wäre z.B. ein *Rendezvous-Server* immer bereit, Nachrichten anzunehmen, so bräuchte ein *Rendezvous-Client* nur solange blockiert zu sein, wie der eigentliche Transfer der Nachricht an Zeit benötigt. Das Kommunikationsprotokoll zwischen *Rendezvous-Client* und *Rendezvous-Server* ist dafür verantwortlich, wann das jeweilige *Rendezvous* abgeschlossen werden kann. Würde direkt nach dem Empfang einer Nachricht der *Rendezvous-Server* den *Rendezvous-Client* reaktivieren, d.h. "*replyen*", wäre die Interprozeßkommunikation für den *Rendezvous-Client* nichtblockierend hinsichtlich der Verarbeitung der Nachricht durch den empfangenden Prozeß.

Auch wenn nichtblockierende Kommunikationsmechanismen auf Grundlage von *Message-Passing* in einem Kernel realisiert sind, bleiben die anwendenden Prozesse während des Transfers der jeweiligen Daten blockiert. Die zu übermittelnden Nachrichten werden hierbei

üblicherweise in entsprechenden Datenbereichen des Kerneladreβraumes zwischengespeichert. Desweiteren bedeutet diese Zwischenspeicherung der Nachrichten oftmals einen doppelten Kopiervorgang, um die Übermittlung der jeweiligen Nachrichten vom Sender- zum Empfängerprozeβ nichtblockierend bewerkstelligen zu können. Mit diesem Hintergrund ergibt sich kein signifikanter Unterschied zwischen den blockierenden und den nichtblockierenden Kommunikationsmechanismen. Wesentlich für die Blockierung eines Prozesses ist hierbei immer das Protokoll eines empfangenden Prozesses zur Verarbeitung einer Nachricht.

Die Blockierung eines Prozesses, bei der Anwendung blockierender Kommunikationsmechanismen des MOOSE-Kernels, läβt sich allgemein wie folgt klassifizieren:

- ein empfangender Prozeβ blockiert nur dann, wenn keine Nachrichten zum Empfang bereitstehen;
- die Dauer der Blockierung eines sendenden Prozesses ist linear abhängig von der Dauer der Verarbeitung einer Nachricht durch den empfangenden Prozeβ;
- die Dauer der Blockierung eines sendenden Prozesses ist abhängig von der Anzahl der Prozesse, die in der Empfangsreihenfolge vor dem betrachteten Senderprozeβ stehen.

Entsprechend dieser Klassifikation lassen sich, insbesondere mit Hilfe des *Team*-Konzeptes von MOOSE, Modelle angeben, die nichtblockierendes Senden und Empfangen von Nachrichten für Prozesse ermöglichen. Zur Realisierung bestimmter Kommunikationsprotokolle sind diese Modelle den durch einen Kernel realisierten Modellen nichtblockierender Kommunikationsmechanismen überlegen.

In [Liskov 1979] sind grundlegende Semantiken von Kommunikationsmechanismen beschrieben und [Bauerfeld 1981] wie auch [Shatz 1984] diskutieren diese Mechanismen vor dem Hintergrund netzwerkorientierter Kommunikationssysteme. Die in den nachfolgenden Abschnitten vorgestellten Modelle funktional angereicherter Mechanismen zur Interprozeβkommunikation, zeigen, wie insbesondere die in [Liskov 1979] skizzierten Mechanismen unter MOOSE in einfacher Weise zu realisieren sind.

5.5.2. Nichtblockierender Empfang von Nachrichten

Das einfachste Modell zur Realisierung nichtblockierender Kommunikationsmechanismen ergibt sich im Zusammenhang des nichtblockierenden Empfangs einer Nachricht, dem *nonblocking receive*. Die Realisierung des *nonblocking receives* von Nachrichten basiert im wesentlichen auf Informationen die Aufschluβ darüber geben, wieviel Nachrichten zum Empfang bereitstehen. Bild 5.11 skizziert das Modell eines *nonblocking receives* auf Grundlage des *Team*-Konzeptes von MOOSE.

In diesem Beispiel repräsentiert R1 den Prozeβ (*custodian*), der nichtblockierend Nachrichten annehmen will. S1 ist hierbei der Prozeβ, der potentiell eine Nachricht übermitteln kann. Das Modell des *nonblocking receives* ist hierbei durch zwei Entwurfsentscheidungen geprägt:

a) in dem *Team* von R1 ist ein zusätzlicher Prozeβ (*captive*), R0, angesiedelt, der repräsentativ für R1 die Nachrichten blockierend empfängt. Hierzu wird R0 das *nonspecific receive* des MOOSE-Kernels anwenden;

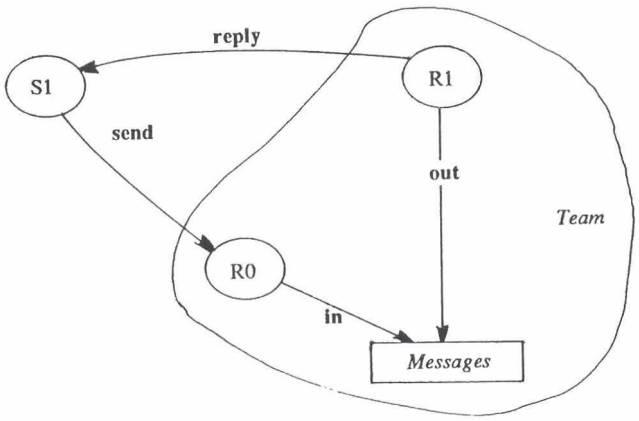

Bild 5.11: *Nonblocking Receive*

b) die Annahme einer Nachricht durch R1 wird z.B. auf elementare Operationen abgebildet, die auf *team*-lokalen Datenbestände operieren, die jeweils die durch R0 empfangenden Nachrichten repräsentieren. Diese Operationen ermöglichen es, direkt in Erfahrung zu bringen, ob und wieviel Nachrichten zur Verarbeitung bereitstehen.

Das *nonspecific receive* durch R0 ermöglicht, daß Nachrichten von beliebigen Prozessen angenommen werden können. Die jeweils empfangende Nachricht wird durch R0 in dem *team*-lokalen Datenbestand mit aufgenommen und damit R1 zur Verfügung gestellt. Das *nonblocking receive* durch R1 operiert direkt auf diesem Datenbestand, entnimmt, wenn vorhanden, die nächste Nachricht und liefert ein entsprechendes Ergebnis, das den erfolgreichen oder nicht erfolgreichen Empfang einer Nachricht anzeigt. Liefert das *nonblocking receive* die *process ID* des Prozesses, der für die Übermittlung der Nachricht verantwortlich ist, so kann R1 eigenständig das *Rendezvous* zwischen S1 und R0 abschließen, nachdem die Nachricht verarbeitet worden ist. Das *Rendezvous*-Konzept des MOOSE-Kernels ermöglicht es gerade, daß alle Prozesse eines *Teams* (in diesem Beispiel R0 und R1) das *reply*-Zugriffsrecht auf einen *Rendezvous-Client* (in diesem Beispiel S1) besitzen.

Für die Effizienz dieses Modells eines *nonblocking receives* ergibt sich aufgrund des *Team*-Modells ein bedeutender Sachverhalt. Der Empfang einer Nachricht durch R1 bedingt zwar die zusätzliche Aktivierung eines Prozesses (R0), die Nachricht selbst wird jedoch nur einmal transferiert. Es ist in diesem Fall eine Frage der Schnittstelle des *nonblocking receives*, ob das abermalige Kopieren von Nachrichten erforderlich wird, damit R1 den Zugriff darauf erhält. Mit einem *Call-by-Reference* Argument, zur Übergabe der Nachricht an R1, wird in jedem Fall der zweite Kopiervorgang vermieden.

Ein *nonblocking receive*, daß direkt in einem *Message-Passing* Kernel integriert ist, wird nicht notwendigerweise eine laufzeiteffizientere Interprozeßkommunikation bedingen, als mit dem hier skizzierten Modell möglich ist. Der Nachrichtentransfer selbst wird nicht schneller vollzogen werden können. Nur die Aktivierung des zusätzlichen Prozesses könnte die

Laufzeiten bei der Interprozeβkommunikation auf Grundlage des hier skizzierten Modells eines *nonblocking receives* heraufsetzen. Der zusätzliche Prozeβ R0 bedeutet jedoch gleichzeitig, daβ während der inaktiven Phase von R1, Nachrichten bereits in den Adreβraum von R1 transferiert werden können. Mit dieser Technik könnten weitestgehend die auftretenden Laufzeiten bei der zusätzlichen Prozeβumschaltung kompensiert werden. Ein vergleichbares Verfahren auf Grundlage eines im Kernel realisierten *nonblocking receives*, bedingt spezielle Mechanismen zum Datentransfer im Kernel (z.B. *Shared-Memory* zwischen S1 und R1) und konfrontiert den Kernel, wie auch den empfangenden Prozeβ, mit den damit verbundenen Nebenläufigkeitsproblemen.

5.5.3. Nichtblockierendes Senden von Nachrichten

Für ein Modell zum nichtblockierenden Senden von Nachrichten, dem *nonblocking send*, stehen prinzipiell zwei Ansatzpunkte zur Verfügung. Beiden Ansatzpunkten ist gemeinsam, daβ, ähnlich wie bei der Skizzierung des *nonblocking receives*, ein zusätzlicher Prozeβ (*captive*) in einem *Team* angesiedelt wird, der jeweils das *nonblocking send* für einen Prozeβ (*custodian*) ermöglicht. Der Unterschied zwischen beiden Ansätzen richtet sich danach, ob dieser zusätzliche Prozeβ im *Team* des sendenden oder des empfangenden Prozesses angesiedelt ist. Je nach seiner Zugehörigkeit zu einem der beiden betrachteten *Teams*, ergeben sich bestimmte Schluβfolgerungen für Kommunikationsprotokolle auf Grundlage des hier skizzierten *nonblocking sends*.

5.5.3.1. Senderseitige Kontrolle

Der eine Ansatz zur Verwirklichung des *nonblocking sends* ordnet dem *Team* des jeweils sendenden Prozesses den zusätzlichen Prozeβ zu. Dieser zusätzliche Prozeβ ist somit lokal dem sendenden Prozeβ zugeordnet und übernimmt die senderseitige Kontrolle der Nachrichtenübermittlung. Bild 5.12 skizziert dieses Modell.

Das *nonblocking send* von S1 zu R1 wird dadurch eingeleitet, daβ S1 die zu übermittelnde Nachricht in einen bestimmten Datenbereich innerhalb seines *Teams* ablegt. Die Übermittlung eines Signals zeigt S0 an, daβ eine Nachricht zur Übermittlung bereitsteht. S0 übernimmt für S1 die Übermittlung der jeweiligen Nachrichten zu den entsprechenden Empfängerprozessen (in diesem Beispiel R1). S0 verwendet hierzu die vom MOOSE-Kernel zur Verfügung gestellten Mechanismen zur Interprozeβkommunikation (in diesem Fall das *send*) und blockiert damit für S1.

Je nach der Schnittstelle des *nonblocking sends* kann ein doppelter Kopiervorgang der zu übermittelnden Nachricht vermieden werden. Hierzu wäre eine ähnliche Strategie in Anwendung zu bringen, wie sie im Zusammenhang mit dem *nonblocking receive* bereits skizziert worden ist.

Die senderseitige Kontrolle über nichtblockierend übermittelte Nachrichten unterstützt die elementare Realisierung komplexerer Kommunikationsprotokolle. So kann insbesondere der zur Verfügung stehende Datenbereich, zur Aufnahme der zu übermittelnden Nachrichten, nach den jeweiligen Empfängerprozessen ausgerichtet sein. Jedem Empfängerprozeβ können lokale (senderseitige) Datenbereiche unterschiedlicher Kapazitäten zugeordnet werden. Diese

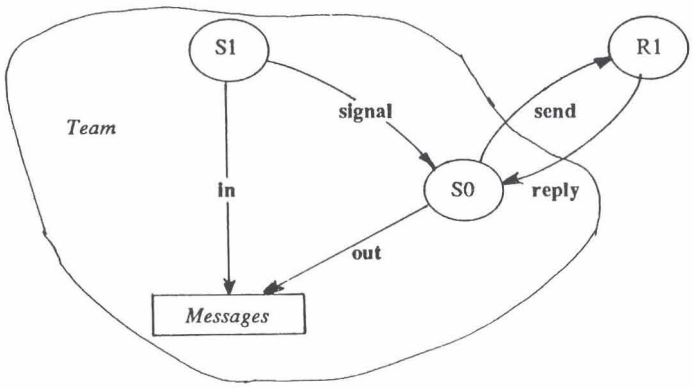

Bild 5.12: *Nonblocking send* mit senderseitiger Kontrolle

Datenbereiche sind jeweils mit einem bestimmten *Team* in Verbindung zu sehen, so daß in unterschiedlichen *Teams* verschiedene Kapazitäten dieser Datenbereiche vorhanden sein können. Für jede Applikation könnte somit eine optimale Realisierung des *nonblocking sends* erreicht werden.

Ein weiterer Aspekt der senderseitigen Kontrolle zu übermittelnder Nachrichten ergibt sich im Zusammenhang mit der Verfügbarkeit dieser Nachrichten für den nichtblockierenden Prozeß S1. Alle noch nicht von S0 übermittelten Nachrichten sind allen Prozessen innerhalb desselben Teams von S0 zugänglich. Damit kann insbesondere durch S1 die Verwerfung aller noch zu übermittelnden bzw. bestimmten Prozessen zuzustellenden Nachrichten erreicht werden.

Das *nonblocking send* mit senderseitiger Kontrolle entspricht vollständig dem in [Liskov 1979] beschriebenen *no-wait send*. S1 wird selbst mit der Übermittlung einer Nachricht zu R1 nicht direkt betroffen sein und kann, durch Abgabe der Nachricht zu S0, mit seiner weiteren Ausführung fortfahren.

5.5.3.2. Empfängerseitige Kontrolle

Der andere Ansatz für ein *nonblocking send* sieht vor, die nichtblockierende Übermittlung von Nachrichten durch einen empfangenden Prozeß kontrollieren zu lassen. Bei diesem Ansatz wird dem "logisch" empfangenden Prozeß in seinem *Team* ein zusätzlicher Prozeß zugeordnet. Bild 5.13 skizziert diese Variante des Modells eines *nonblocking sends*.

Der wesentliche Unterschied zu dem vorher skizzierten Ansatz besteht darin, daß die durch S1 übermittelten Nachrichten in jedem Fall in ein anderes *Team* (das von R1) plaziert werden. Die Strategien, bereits übermittelte Nachrichten vor ihrer Verarbeitung zu verwerfen, werden hierbei durch R0 bestimmt und sind im allgemeinen für alle sendenden Prozesse einheitlich. Mit dem Ansatz der senderseitigen Kontrolle des *nonblocking sends* dagegen, könnte jeder sendende Prozeß seine eigene Strategie zur Verwaltung zu übermittelnder Nachrichten angeben.

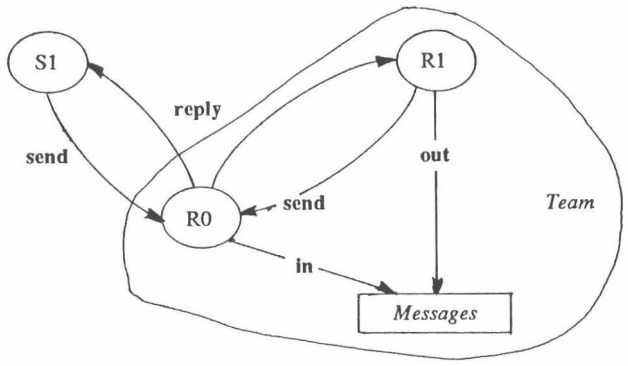

Bild 5.13: *Nonblocking send* mit empfängerseitiger Kontrolle

Das *nonblocking send* wird durch R0 realisiert. Dieser Prozeβ wendet das *Rendezvous*-Konzept des MOOSE-Kernels an, um die von S1 übermittelten Nachrichten anzunehmen; S1 wird, nach Annahme einer Nachricht durch R0, sofort wieder reaktiviert. R0 wird die empfangenden Nachrichten lediglich in einen bestimmten (lokal in seinem *Team* zur Verfügung stehenden) Datenbereich ablegen und sonst keine weiteren Aktivitäten im Zusammenhang mit den jeweiligen Nachrichten durchführen. Damit kann dieser Prozeβ immer als empfangsbereit bezeichnet werden, mit der Konsequenz, daβ die Blockierungsdauer sendender Prozesses minimal gehalten wird.

Die eigentliche Verarbeitung der jeweiligen Nachrichten wird durch R1 vorgenommen. Die Bereitschaft dazu zeigt R1 durch das *Rendezvous* mit R0 an. Diese Beziehung von R1 zu R0 dient vornehmlich der Synchronisation von R1 auf den Empfang einer Nachricht aus dem durch R0 gefüllten Datenbereich des beiden Prozessen gemeinsamen *Teams*. Die Beendingung des *Rendezvous* zwischen R1 und R0 (erreicht durch R0), signalisiert R1, daβ eine weitere Nachricht zur Verarbeitung bereitsteht.

Zwischen S1 und R0 wird mit diesem Modell ein *synchronization send* nach [Liskov 1979] realisiert. R0 reaktiviert S1, sobald die Nachricht von S1 übernommen worden ist. Zwischen S1 und R1 besteht jedoch die Kommunikationssemantik eines *no-wait send*, da S1 nicht auf die Annahme der Nachricht von R1, dem die Nachricht verarbeitenden Prozeβ, warten muβ.

5.5.4. Nichtblockierendes Senden und Empfangen von Nachrichten

Die bisher dargestellten Modelle nichtblockierender Kommunikationsmechanismen lassen sich in elementarer Weise miteinander verknüpfen. Daraus ergibt sich ein Kommunikationsmodell, das jeweils das nichtblockierende Senden und Empfangen von Nachrichten unterstützt und damit die asynchrone Interprozeβkommunikation ermöglicht.

5.5.4.1. Einseitig asynchrone Interprozeβkommunikation

Je nach betrachtetem Modell für das *nonblocking send* zeichnen sich zur Realisierung einseitig asynchroner Kommunikationsmechanismen ebenfalls zwei Varianten ab. Bild 5.14 skizziert hierbei die Variante mit empfängerlokaler Kontrolle der nichtblockierend übermittelten Nachrichten.

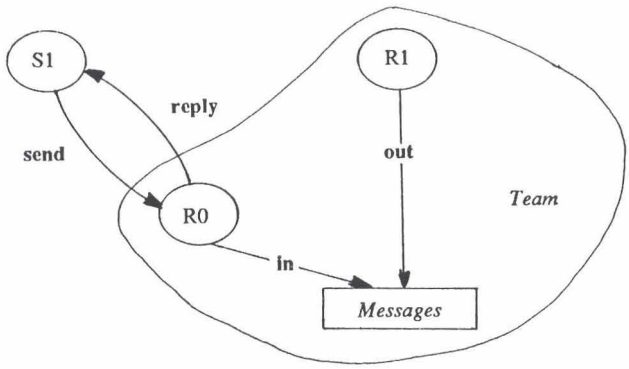

Bild 5.14: Einseitig asynchrone Kommunikation mit empfängerseitiger Kontrolle

Die mit diesem Modell dargestellte Variante unterscheidet sich nur minimal von den skizzierten Modellen des *nonblocking receives* und *nonblocking sends* mit empfängerseitiger Kontrolle der Nachrichtenübermittlung. Der signifikante Unterschied zum zuletzt genannten Modell ergibt sich hierbei dadurch, daβ zwischen R1 und R0 kein *Rendezvous* vorgesehen ist und damit auch keine Blockierung von R1 auftreten kann. Der signifikante Unterschied zu dem *nonblocking receive* besteht darin, daβ R0 bereits das *Rendezvous* zu S1 abschlieβt. S1 bleibt somit nicht solange blockiert, bis R1 die Verarbeitung der von S1 übermittelten Nachrichten vollständig abgeschlossen hat.

Die andere Variante zum nichtblockierenden Senden und Empfangen von Nachrichten basiert auf dem *nonblocking send* mit senderlokaler Kontrolle der Nachrichtenübermittlung. Bild 5.15 skizziert das dazu notwendige Modell.

In dieser Variante ist S1 unabhängig davon, ob R0 oder R1 das *Rendezvous* im Zusammenhang mit der Kommunikation zwischen S1 und R1 abschlieβt. Dadurch, daβ R0 selbst das *Rendezvous* zu S0 abschlieβt, ist S0 in der Lage, die zu übermittelnden Nachrichten, unabhängig von dem Laufzeitverhalten von R1, nach eigener Strategie, den entsprechenden Bestimmungsprozessen zustellen zu können. R1 schlieβlich ist unabhängig von dem jeweiligen Sendeprofil von S1.

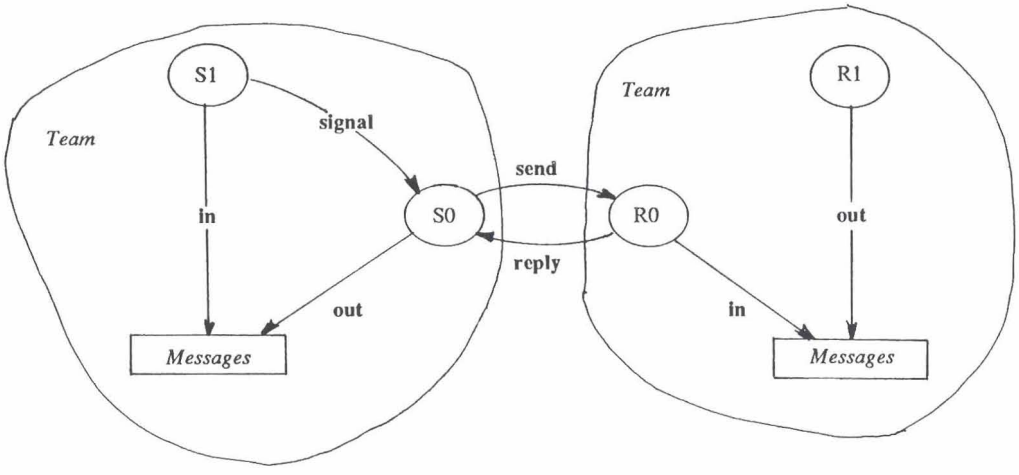

Bild 5.15: Einseitig asynchrone Kommunikation mit senderseitiger Kontrolle

5.5.4.2. Vollständig asynchrone Interprozeβkommunikation

Die Realisierung eines vollständig asynchronen Kommunikationsmodells ist durch die elementare Erweiterung des Modells zur einseitig asynchronen Interprozeβkommunikation möglich. Die *Teams* des jeweils sendenden und empfangenden Prozesses, resp. S1 und R1, brauchen lediglich den gleichen Satz an unterstützenden Prozessen aufzuweisen. Bild 5.16 stellt das entsprechende Modell auf Grundlage der empfängerseitigen Kontrolle der Nachrichtenübermittlung vor.

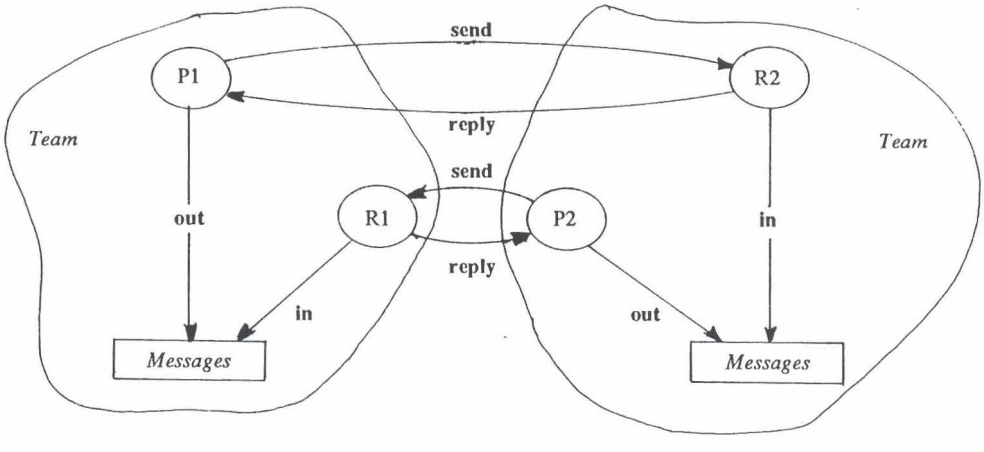

Bild 5.16: Asynchrone Kommunikation mit empfängerseitiger Kontrolle

Die Prozesse R1 und R2 erbringen in diesem Modell jeweils zwei Funktionalitäten. Zum einen ermöglichen sie das *nonblocking send*, und zwar R2 für P1 und R1 für P2. Zum

anderen gestatten sie das *nonblocking receive* den Prozessen, die ihrem *Team* zugeordnet sind. In dieser Hinsicht wirken hierbei jeweils R2 unterstützend für P2 und R1 unterstützend für P1.

Die andere Variante, skizziert durch Bild 5.17, basiert auf dem Modell der senderseitigen Kontrolle des *nonblocking sends*. Hierbei ergeben sich wiederum die grundlegenden Vorteile des in Anwendung gebrachten Modells des *nonblocking sends*.

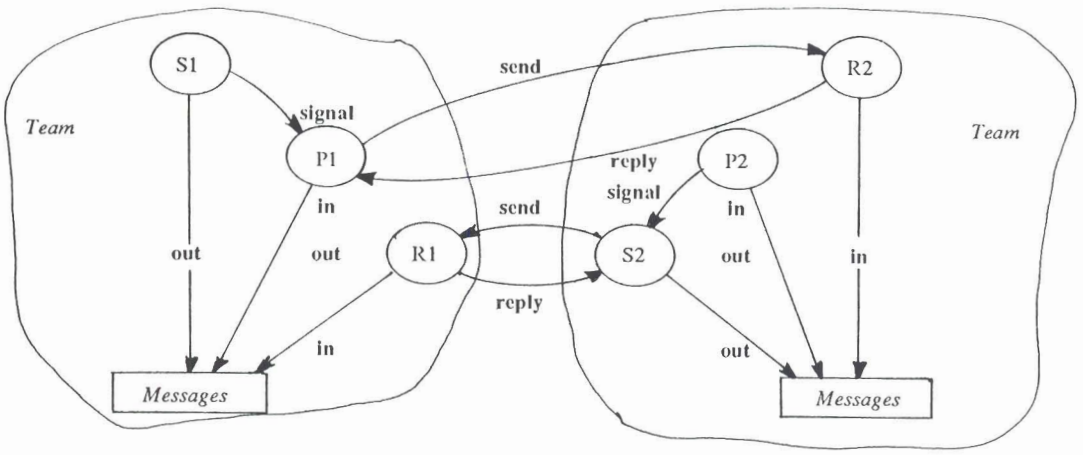

Bild 5.17: Asynchrone Kommunikation mit beidseitiger Kontrolle

Diese Variante repräsentiert bereits eine sehr konsequente Anwendung des *Team*-Konzeptes zur Modellierung komplexerer Kommunikationsarchitekturen. Auch in diesem durchaus komplexen Modell muß es nicht bedeuten, daß die Vielzahl der notwendigen Prozeßwechsel einen bedeutenden Leistungsverlust des jeweiligen *nonblocking sends* bzw. *nonblocking receives* mit sich bringen. Die jeweiligen Protokolle zur Realisierung des *nonblocking sends* und *nonblocking receives* könnten so ausgelegt werden, daß die Aktivitäten von S1 und R1 bzw. S2 und R2 jeweils während der inaktiven Phase von P1 bzw. P2 stattfinden. Die durch den MOOSE-Kernel festgelegten Ausführungsstrategien von Prozessen innerhalb desselben *Teams* ermöglichen hierbei die für P1 bzw. P2 transparente Abarbeitung von S1 und R1 bzw. S2 und R2.

5.5.5. Analogie zu netzwerkorientierten Kommunikationssystemen

Mit den hier vorgestellten Modellen zur nichtblockierenden Interprozeßkommunikation lassen sich vergleichbare Ansatzpunkte herausarbeiten, wie sie im Zusammenhang mit Kommunikationsarchitekturen für netzwerkorientierte Systeme gelten. Das bedeutet, daß die Prinzipien zur Kommunikation in dezentral organisierten Systemen ebenfalls für Monoprozessorsysteme ihre Gültigkeit besitzen.

Die mit Hilfe des *Team*-Konzeptes realisierten nichtblockierenden Mechanismen zur Interprozeßkommunikation basieren auf Konzepten, wie sie grundlegend in [Schindler 1980] und [ISO 1985] niedergelegt sind. Dies gilt ebenfalls für die vom MOOSE-Kernel zur

Verfügung gestellten Mechanismen zur Interprozeβkommunikation. Der Zusammenhang mit [Schindler 1980] bzw. [ISO 1985] ist hierbei jedoch nicht direkt offensichtlich. Die Ursache hierfür liegt darin, daβ die Kommunikationsmechanismen des MOOSE-Kernels selbst sehr elementar und einfach zu handhaben sind und ihre Anwendung die Notwendigkeit konkreter Kommunikationsprotokolle nicht explizit hervorhebt. Dennoch, auch in diesem Fall müssen sich die miteinander kommunizierenden Prozesse über ein bestimmtes Format der auszutauschenden Nachrichten einigen. Eine Kommunikation im Sinne des gemeinsamen Verständnisses über die auszutauschenden Informationen wäre sonst nicht möglich. Damit besteht auf dieser Ebene bereits die Notwendigkeit eines, wenn auch sehr elementaren, Kommunikationsprotokolls.

Zur Realisierung der skizzierten Modelle zur nichtblockierenden Kommunikation zwischen Prozessen sind bereits kompliziertere Kommunikationsprotokolle notwendig. Damit ergibt sich bereits in "natürlicher" Weise eine Hierarchie von verschiedenen Kommunikationsebenen. Wird noch in Betracht gezogen, daβ, auf Grundlage dieser Kommunikationsebenen, die applizierenden Prozesse selbst ein bestimmtes Kommunikationsprotokoll benötigen, um untereinander kommunizieren zu können, so ergibt sich für die skizzierten Kommunikationsmodelle eine drei-schichtige Kommunikationsarchitektur. Bild 5.18 stellt die Hierarchie der einzelnen Kommunikationsebenen dar[44].

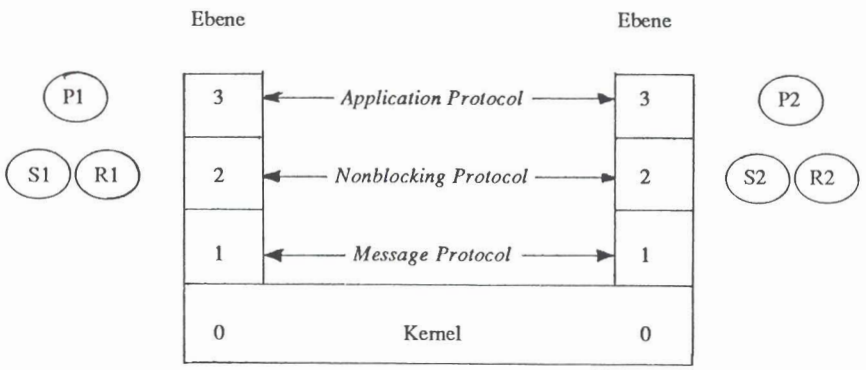

Bild 5.18: Hierarchie von Kommunikationsebenen

Der MOOSE-Kernel würde in diesem Zusammenhang das Transportmedium darstellen, das insbesondere keine Anforderungen an die zu übermittelnden Nachrichten stellt. Auf der Ebene 1 würde ein einfaches *Message Protocol* angesiedelt sein, wie es auch seine Anwendung zwischen Benutzer- und Systemprozessen unter MOOSE findet. Dienstanforderungen von

[44] Die in dieser Abbildung dargestellten Kommunikationsebenen besitzen lediglich einen beispielhaften Charakter für die in diesem Kapitel vorgestellten Mechanismen zur nichtblockierenden Interprozeβkommunikation. Diese Ebenen besitzen nicht die in [ISO 1985] beschriebenen Funktionalitäten.

Prozessen werden durch entsprechende Nachrichtenformate der Ebene 1 repräsentiert. Auf der Ebene 2 ist im wesentlichen ein Protokoll, als *Nonblocking Protocol* bezeichnet, angesiedelt, das jeweils zur Realisierung der nichtblockierenden Kommunikationsdienste beiträgt. Ab der Ebene 2 stehen somit nichtblockierende Kommunikationsdienste zur Verfügung. Die Ebene 3 schließlich repräsentiert das applikationsorientierte Kommunikationsprotokoll, mit *Application Protocol* bezeichnet. Es ist in wesentlichen Zügen mit dem *Message Protocol* der Ebene 1 vergleichbar.

Die Realisierung dieser Kommunikationsarchitekturen muß keinesfalls bedeuten, daß die Kommunikation zwischen P1 und P2 mit einem großen Verwaltungsaufwand verbunden ist. Der Verwaltungsaufwand wird sich im wesentlichen nach der Komplexität der jeweiligen Kommunkationsprotokolle der dargestellten Ebenen richten. In jedem Fall sind Abbildungen an den Schnittstellen der einzelnen Ebenen notwendig, die die zu übermittelnden Nachrichten der Ebene *n* über Nachrichten der Ebene *n-1* repräsentieren. Das bedeutet, daß eine der Ebene *n* zur Verfügung gestellte *service data unit* mindestens auf eine *protocol data unit* der Ebene *n-1* abgebildet wird. Ob diese Abbildung technisch realisiert werden muß oder nur konzeptionell zu beachten ist, bestimmt das jeweilige Kommunikationsprotokoll der Ebene *n-1*.

Neben den reinen datenstrukturellen Betrachtungen lassen sich ebenfalls bestimmte operationelle Beziehungen zwischen den miteinander kommunizierenden Prozessen, insbesondere im Sinne von [Schindler 1980], vergleichend herausarbeiten. Die Modelle des *nonblocking sends* repräsentieren jeweils einen *nonconfirmed service* im Zusammenhang mit der Kommunikation zwischen Prozessen der Ebene 3. Hierbei sind insbesondere Strategien realisierbar, die das Absetzen eines *nonconfirmed service* sender- oder empfangsseitig anzeigen können. Dies richtet sich nach dem in Anwendung gebrachten Modells des *nonblocking sends*, d.h. ob die sender- oder empfangsseitige Kontrolle der Nachrichtenübermittlung auf der Ebene 2 betrachtet wird.

Ist auf der Ebene 2 ein Protokoll realisiert, das nur das *nonblocking receive* unterstützt, so würde hiermit das Prinzip des *confirmed service* seine Anwendung finden können. Die Kommunikationsmechanismen des MOOSE-Kernels repräsentieren in diesem Zusammenhang immer einen *confirmed service*. Dies ist auf das *Rendezvous*-Konzept zurückzuführen. Auf der Ebene 1 jedoch, kann bereits ein Kommunikationsprotokoll seine Anwendung finden, das einen *nonconfirmed service* ansatzweise unterstützt. Hierzu braucht sich das *Rendezvous* zwischen dem jeweils sendenden- und empfangenden Prozeß nur auf den Transfer und nicht auf die Verarbeitung einer Nachricht zu beziehen, d.h. nach [Liskov 1979], ein *synchronization send* anstatt eines *remote invocation send* zugrunde legen. Dies gerade stellt die wesentliche Voraussetzung dar, das *Nonblocking Protocol* der Ebene 2 realisieren zu können.

Kapitel 6

Advanced UNIX

In den vorangegangenen Kapiteln sind die Mechanismen von MOOSE im einzelnen vorgestellt worden. Die Funktionalität des MOOSE-Kernels und der zentralen MOOSE-Administratoren (*Service-*, *Team-* und *Context-Administrator*) sind erläutert worden. Die Bedeutung des *Team*-Konzeptes unter MOOSE (komplexe Kommunikationsmechanismen und benutzerspezifische Behandlung von Ausnahmesituationen) wurde dargestellt, ebenso wie die Mechanismen zur dynamischen Rekonfiguration eines Systems bzw. zur Modellierung einer Betriebssystemfamilie. In diesem Kapitel wird anhand eines konkreten Mitglieds der MOOSE-Betriebssystemfamilie aufgezeigt, wie die bisher vorgestellten Funktionalitäten von MOOSE genutzt werden können, um ein funktionstüchtiges Betriebssystem aufzubauen.

Das nachfolgend von seiner Struktur her vorgestellte Betriebssystem wird AX, abkürzend für *Advanced UNIX*, genannt. Von diesem System liegen gegenwärtig zwei Versionen vor. Zum einen eine vollständig einsatzfähige Version für den Personal-Computer-Bereich, PAX, und zum anderen eine noch in der Portierung auf einen Motorola mc68000 befindliche Version, MAX. Im Rahmen der nachfolgenden Diskussion wird die Systemstruktur und die Funktionalität von PAX im Vordergrund stehen. Ein Vergleich mit MAX findet soweit statt, um einerseits den Portierungsaufwand von AX bestimmen und andererseits evtl. Leistungsunterschiede zwischen den beiden Systemen darlegen zu können.

Neben dieser Diskussion von PAX als Mitglied der MOOSE-Betriebssystemfamilie, erfolgt ein genereller Vergleich von PAX mit anderen modernen Betriebssystemen. Hierzu wird jeweils ein Vertreter des prozedur-, prozeß- und objektorientierten Systementwurfs herangezogen. Diese Vertreter sind, in derselben Reihenfolge, UNIX, V [Cheriton 1984] und SOS [Shapiro et al. 1985]. Der Vergleich mit diesen Systemen kann jedoch nur auf konzeptioneller bzw. funktionaler Ebene erfolgen, da zur Realisierung von PAX und MAX gegenwärtig keine vergleichbaren Hardware-Architekturen zugrunde gelegt worden sind. Im Zusammenhang mit der Darstellung der Funktionalitäten von PAX wird jedoch ein Vergleich mit QNX [Quantum 1984] stattfinden. Dies ist zwangsläufig der Fall, da PAX sich gerade dadurch auszeichnet, eine vollständige Emulation von QNX darzustellen.

6.1. AX

Das Einsatzgebiet von AX entspricht dem der gängigen UNIX-Versionen – daher auch das Anghängsel "*UNIX*" bei *Advanced UNIX*. Ein wesentlicher Aspekt bei dem Entwurf und der Realisierung von AX bestand jedoch darin, insbesondere die Lücke bei UNIX zwischen Rechnersystemen der Ein- und Mehr-Benutzerkategorie zu schließen. Demzufolge erstreckt sich das Einsatzgebiet von AX von den einfachen Personal-Computer-Systemen aufwärts, bis

hin zu komplexeren Rechnersystemen mit einem typischen Anwenderprofil von 4 bis 16 Benutzern.

Das Attribut "*Advanced*" soll verdeutlichen, mit AX nicht nur eine Re-Implementierung von UNIX, auf Grundlage eines prozeßorientierten und auf *Message-Passing* basierenden Systementwurfs, durchgeführt zu haben. Vielmehr sind mit AX Konzepte von MOOSE verwirklicht, die es erlauben, die Funktionalität eines spezifischen (Betriebs- oder Applikations-) Systems zur Laufzeit dynamisch ändern zu können. Hierunter fallen insbesondere

- die Integration, der Austausch und das Entfernen von *Schedule-*, *Swap-* und *Account-*Strategien;
- die Erkennung und Auflösung von *Deadlocks*, die durch Anwendung der blockierend wirkenden Kommunikationsprimitiven des MOOSE-Kernels auftreten können;
- die Adoption terminierter Prozesse, um die Kommunikation mit diesen Prozessen als domänenspezifische Ausnahmesituation behandeln zu können;
- die Überwachung der Kommunikationsaktivitäten des Systems.

Diese kurz skizzierten Funktionalitäten sind sämtlichst jeweils durch einen Systemprozeß oder mehrere Systemprozesse in AX realisiert, womit die Grundlage für die dynamische Rekonfiguration eines Systems unter MOOSE gegeben ist.

Der markante Unterschied zwischen MAX und PAX besteht in den verschiedenen Emulationssystemen, die mit beiden AX-Versionen verbunden sind. MAX stellt ein UNIX *look-alike* dar, das eine Emulation UNIX-spezifischer Systemdienste ermöglichen soll. PAX dagegen stellt in erster Linie ein UNIX *work-alike* dar, das durch die QNX-Emulation geprägt ist. Ein UNIX *look-alike* zeichnet sich dadurch aus, daß die von einem entsprechenden System zur Verfügung gestellten *System-Calls* kompatibel zu denen von UNIX sind. Ein UNIX *work-alike* dagegen weist diese Kompatibilität nicht auf, stellt jedoch an der obersten Benutzerschnittstelle typische Dienstprogramme von UNIX zur Verfügung, und die Realisierung des Filesystems entspricht oftmals der von UNIX.

In den nachfolgenden Abschnitten werden die wesentlichen Systemkomponenten von PAX im Überblick vorgestellt[45]. Dabei wird so verfahren, daß die funktionale Anreicherung von PAX, vom erfolgten *Bootstrap* eines Basissystems bis hin zum Mehr-Benutzersystem betrachtet wird.

6.1.1. Systemgenerierung und Bootstrap

Anders als in prozedurorientierten Systemen, stellt sich der *Bootstrap*-Vorgang eines prozeßorientierten Systems, wie AX, komplizierter dar. Die Begründung liegt darin, daß bereits Prozesse, die gerade die administrativen Aufgaben eines prozeßorientierten Betriebssystems wahrnehmen, initial mitgeladen werden müssen. Diese Aufgaben bedeuten z.B. für AX, daß

[45] Im Literaturverzeichnis sind alle Arbeiten des MOOSE-Projektes explizit angegeben, die, über mehrere Entwicklungsphasen hinweg, zur Verwirklichung von AX beigetragen haben. Diese Arbeiten beschreiben detailliert die Struktur und Aufgaben einzelner Systemkomponenten von AX.

Prozesse erzeugt und zerstört, sowie Programme vom Filesystem geladen und zur Ausführung gebracht werden müssen. In UNIX sind die vergleichbaren Funktionalitäten im Kernel integriert. Damit können, nachdem der Kernel geladen worden ist, direkt Prozesse erzeugt und Programme zur Ausführung gebracht werden.

In QNX, das ebenfalls ein prozeßorientiertes Betriebssystem darstellt, werden alle Systemprozesse und der Kernel als ein gesamtes *Load-Image* betrachtet, das während des *Bootstrap*-Vorgangs in den Hauptspeicher transferiert wird. Die Systemgenerierung von QNX bedeutet in diesem Sinne, daß ein Speicherabzug von den mitzuladenen Prozessen erstellt wird. Dieser Speicherabzug wird auf reservierten Sektoren des *Boot-Devices* abgelegt und kann direkt in den Hauptspeicher geladen werden. Die Initialisierung des geladenen Systems bedeutet praktisch die Erzeugung von Systemprozessen, nur daß dazu effektiv kein Prozeß verantwortlich gezeichnet werden kann.

Der *Bootstrap*-Vorgang unter AX verschmelzt die entsprechenden in UNIX und QNX zu betrachtenden Phasen. Einerseits werden die initial zu ladenen Systemkomponenten unter AX über die üblichen *Load-Files* des Systems beschrieben. Das bedeutet, daß diese Systemkomponenten keinen Speicherabzug darstellen. In dieser Hinsicht liefert der Vergleich zu UNIX keinen Unterschied. Andererseits bedeutet die Initialisierung der geladenen Systemkomponenten, wie unter QNX, die Simulation der Erzeugung von Systemprozessen.

Mit diesem Verfahren unter AX ist es möglich, beliebige Programme im Zuge des *Bootstraps* zu laden, ohne daß spezielle Vorkehrungen, wie z.B. die Erzeugung eines Speicherabzuges, getroffen werden müssen. Welche Programme dabei zu betrachten sind, wird durch eine Beschreibung zur Systemgenerierung vorgegeben. Diese Beschreibung definiert in abstrakter (programmiersprachlicher) Notation

- welche Programme als *Teams* geladen werden sollen;
- welche Prozeduren in diesen Programmen als Prozesse zu betrachten sind;
- wie groß der Stackbereich für die einzelnen Prozesse sein soll;
- welche Attribute mit einem *Team* verknüpft sind;
- welche Attribute mit den einzelnen Prozessen eines *Teams* verknüpft sind.

Die in dieser Beschreibung niedergelegten Informationen entsprechen weitestgehend den Argumenten der *System-Calls* von AX, die zur Erzeugung von *Teams* (*custodians*) und Prozessen (*captives*) zur Verfügung stehen. Das nachfolgende Beispiel stellt einen Auszug aus einer solchen Systemspezifikation des initial zu ladenen MOOSE-Systems dar. Dieser Auszug entspricht der für AX gültigen Spezifikation des *Team-Administrators*.

```
team team_admin is
        tsystem tbackground tinform tprivileged
        task _main is
                pindivisible pserver pteam
                stacksize 1024
        end
        task _broadcast is
                pindivisible ptask
                stacksize 512
        end
end
```

Der *Team-Administrator* setzt sich demnach aus zwei Prozessen zusammen. Diese Prozesse werden durch "*_main*" und "*_broadcast*" definiert. Beide Angaben entsprechen den Symbolen von Prozeduren des durch "*team_admin*" definierten *Load-Files*. Der *custodian* ("*_main*") des betreffenden *Teams* ist mit "*pteam*" attributiert, wohingegen "*ptask*" einen *captive* ("*_broadcast*") auszeichnet. Die *team*-globalen Attribute – die Angaben vor der Spezifikation von "*_main*" – sowie die prozeßlokalen Attribute sind jeweils nach dem Schlüsselwort "**is**" angegeben.

Die zu ladenen *Teams*, spezifiziert durch das nach dem Schlüsselwort "**team**" benannte *Load-File*, sind in der *Directory* "*/moose*" auf dem jeweils aktuellen *Boot-Device* abgelegt. Der Ur-Lader für AX kann unterschiedliche Systemspezifikationen entgegen nehmen. Diese Spezifikationen sind ebenfalls unter "*/moose*" abgelegt. Welche der Spezifikation und damit welches System geladen werden soll, gibt der Operateur explizit über die Konsole ein.

Entsprechend der bisher skizzierten Maßnahmen zum initialen Laden eines AX-Systems läßt sich der *Bootstrap*-Vorgang für PAX zusammenfassend wie folgt darstellen:

1. *Primary-Bootstrap*
   Ein einfacher Ur-Lader wird vom *Boot-Device* geladen. Dieser Ur-Lader startet den konfigurationsabhängigen Ladevorgang;
2. *Secondary-Bootstrap*
   Der PAX-*Loader* wird über den Ur-Lader geladen. Entsprechend des *Bootstrap*-Vorgangs von QNX stellt der PAX-*Loader* einen Speicherabzug dar und liegt als *Load-Image* auf reservierten Sektoren des *Boot-Devices* vor;
3. *Main-Bootstrap*
   Das PAX-System wird durch den PAX-*Loader* geladen. Hierzu interpretiert der PAX-*Loader* die ihm übergebene Systemspezifikation. Die in der Systemspezifikation angegebenen *Load-Files* aus "*/moose*" werden als *Teams* geladen, und die innerhalb der jeweiligen *Teams* angesiedelten Prozesse werden initialisiert.

Die Punkte 1) und 2) entsprechen den unter QNX üblichen *Bootstrap*-Vorgängen. Punkt 3) führt dazu, daß alle initial mitgeladenen Prozesse so aufgesetzt werden, daß sie für den

MOOSE-Kernel zur Ausführung auf einen realen Prozessor zur Verfügung stehen. Diese Phase entspricht prinzipiell der "Erzeugung" dieser Prozesse.

Die Systemgenerierung von AX bedeutet:

a) eine Beschreibung der zu ladenen *Teams* sowie die Spezifikation dieser *Teams* und die durch sie zusammengefaßten Prozesse zu erstellen. Diese Beschreibung stellt die Systemspezifikation der zu ladenen AX-Version dar;

b) die betreffenden *Teams* als *Load-Files* unter "*/moose*" zu deponieren;

c) anhand der erstellten Systemspezifikation eine interne Darstellung für den AX-*Loader* zu erzeugen, die die einzelnen Ladevorgänge steuert.

Punkt c) wird durch ein entsprechendes Dienstprogramm ermöglicht. Dieses Dienstprogramm parsiert die Systemspezifikation und kontrolliert das Vorhandensein der enstprechenden *Load-Files*. Desweiteren wird die Symboltabelle der einzelnen *Load-Files* nach den Namen der Prozeduren abgesucht, die repräsentativ für die mitzuladenen Prozesse stehen. Das Resultat der Systemgenerierung (insbesondere von Punkt c)) ist die von dem AX-*Loader* benötigte Beschreibung des jeweils zu ladenen Systems.

6.1.2. Zentrale Systemkomponenten

Der wesentliche Unterschied zwischen PAX und MAX ist struktureller Natur. Zum einen ist unter PAX der *Context-Administrator* mit dem *Team-Administrator* verschmolzen. MAX stellt in dieser Hinsicht ein Redesign von PAX dar, da es sich aus Gründen der Portabilität als sinnvoll erwiesen hat, die hardware-abhängige Systemkomponente, zur Zuordnung eines Adreβraumes für die einzelnen Prozesse, durch einen eigenen Systemprozeβ abzukapseln. Zum anderen findet der *Domain-Administrator* unter PAX als zentrale Systemkomponente nicht seinen Einsatz. Die Begründung hierfür ergibt sich durch das mit PAX verbundene Emulationssystem für QNX und dem sich daraus ergebenden Einsatz in einer QNX-Umgebung. Die Funktionalitäten des *Domain-Administrators* im Sinne von QNX, werden durch die Emulation QNX-spezifischer Systemdienste eigenständig vom QNX-Emulator erbracht. Unter MAX wird jedoch der *Domain-Administrator* als zentrale Systemkomponente integriert sein. Die im Zusammenhang mit einer UNIX-Emulation benötigten Systemdienste zur Signalbehandlung sowie zur Definition von Benutzer- und Prozeβgruppen, werden durch den *Domain-Administrator* zur Verfügung gestellt.

Nicht alle Systemprozesse von PAX sind nach erfolgtem *Bootstrap* des Systems bereits präsent. Es sind nur jeweils jene Systemprozesse vorhanden, die minimal benötigt werden, um den Rest des Systems zu konfigurieren. Bild 6.1 skizziert die Beziehung dieser initialen Systemprozesse nach erfolgtem *Bootstrap* von PAX.

Die Funktionalitäten des *Service-* und *Team-Administrators* sind bereits in vorangegangenen Abschnitten beschrieben worden. Unter PAX nimmt der *Team-Administrator* zusätzlich die Funktionalität des *Context-Administrators* wahr. Die Beziehung zu dem *Service-Administrator* ist der Übersichtlichkeit wegen weggelassen. Von jedem Prozeβ in PAX wird eine Beziehung zum *Service-Administrator* bestehen, um Dienste der anderen Prozesse des Systems in Anspruch nehmen zu können bzw. selbst Dienste systemglobal zur Verfügung

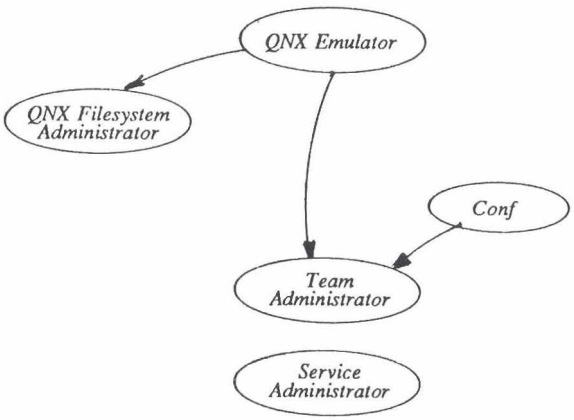

Bild 6.1: PAX nach dem *Bootstrap*

zu stellen.

Es ist bereits erwähnt worden, daß PAX eine vollständige Emulation des QNX-Betriebssystems ermöglicht. In diesem Sinne stellen der QNX-Emulator und QNX-*Filesystem-Administrator* die zentralen Systemprozesse der QNX-Betriebssystemdomäne dar.

Der QNX-Emulator nimmt im wesentlichen die folgenden Funktionalitäten wahr (siehe auch [Quantum 1984]):

- er stellt alle Dienste des *Task-Administrators* von QNX zur Verfügung;
- er emuliert alle Dienste des Kernels von QNX, die nicht hinreichend ähnlich sind zu Diensten des MOOSE-Kernels. Das bedeutet, daß unter PAX die Abbildung von QNX-Diensten zuerst auf der Kernelebene erfolgt. Komplexere Abbildungen werden zum Emulatorprozeß weitergereicht. Insbesondere können auf der Kernelebene alle Dienste des *Device-Administrators* von QNX in einfacher Weise auf entsprechende Dienste der PAX-Systemprozesse effizient abgebildet werden.

Der QNX-*Filesystem-Administrator* von PAX entspricht vollständig dem originalen Systemprozeß von QNX. Dieser Systemprozeß ist in der Weise für PAX konfiguriert worden, daß an seiner Schnittstelle zu dem Ein-/Ausgabesystem eine Abbildung auf entsprechende Dienste des Ein-/Ausgabesystems von PAX stattfindet. Hierfür ist das *Port-Package* des QNX-*Filesystem-Administrators* entsprechend für PAX adaptiert worden.

6.1.2.1. Ein-Benutzerbetrieb

Auf Grundlage der vier skizzierten Systemkomponenten, wird durch *Conf* die weitere Konfiguration von PAX, bis hin zum vollständigen Mehr-Benutzersystem, kontrolliert. Der erste Schritt besteht hierbei darin, Systemprozesse zur Gerätekontrolle zur Verfügung zu stellen. Hierunter fällt die Erzeugung des *Block-I/O-Systems* und *Character-I/O-Systems*. Zu einer für die Erzeugung dieser Systemprozesse notwendigen Maßnahme gehört u.a. das Laden

bestimmter Programme. Um dies in dieser Phase der Systeminitialisierung von PAX bewerkstelligen zu können, ist der QNX-*Filesystem-Administrator* mit dem Gerätetreiber für das *Boot-Device* zusammengebunden. Dieser Gerätetreiber stellt jedoch nur sehr geringe Funktionalitäten zur Verfügung, die darin bestehen, Blöcke vom *Boot-Device* nur lesen und in den Arbeitsspeicher transferieren zu können. Das *Block-I/O-System* übernimmt und erweitert die Funktionalität dieses Gerätetreibers, nachdem es selbst geladen ("*gebootstrapped*") worden ist. Bild 6.2 stellt die Struktur von PAX vor, wie sie sich nach der zusätzlichen Erzeugung des *Single-User Shells* ergibt.

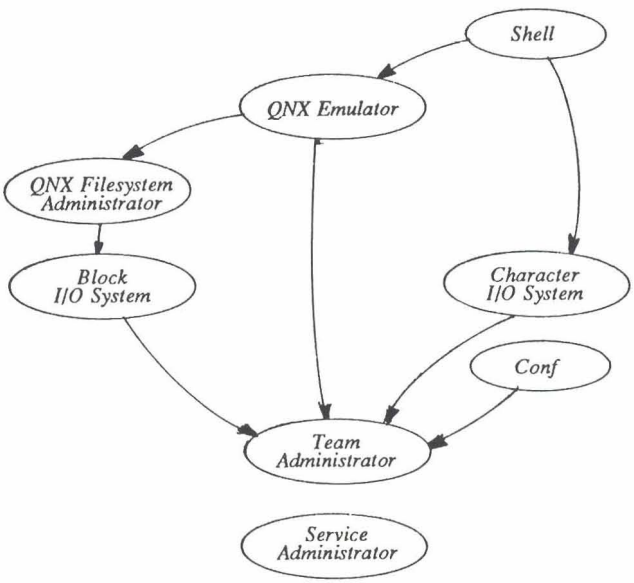

Bild 6.2: PAX im Ein-Benutzerbetrieb

Das *Block-I/O-System* stellt dem QNX-*Filesystem-Administrator* funktional angereicherte Ein-/Ausgabedienste zur Verfügung. Dies bedeutet Hochvolumenverkehr beim Lesen/Schreiben von Blöcken, auch als *multi sector I/O* bezeichnet. Desweiteren ist im *Block-I/O-System* ein *Buffer-Cache* angesiedelt, um bei langsamen Ein-/Ausgabegeräten (z.B. *Floppy-Disk-Drives*) die Wartezeiten beim Lesen/Schreiben der Blöcke für die das *Block-I/O-System* benutzenden Prozesse (in diesem Fall der QNX-*Filesystem-Administrator*) zu minimieren.

Das *Character-I/O-System* kontrolliert die zeichenorientierten Geräte. Die Funktionalitäten dieses Systems beinhalten einfaches *Line-Editing*, *Character-Mapping* sowie ein Start-/Stop-Protokoll zur Flußkontrolle und die Erkennung von Sonderzeichen zur Kontrolle der Ausführung von Prozeßgruppen.

Der *Shell* ermöglicht dem Operateur die Interaktion mit dem System. Dieser Prozeß repräsentiert den originalen QNX-Shell und wird demzufolge vollständig emuliert. Auf dieser

Ebene sind bereits sämtliche QNX-Dienstprogramme ausführbar.

6.1.2.2. Mehr-Benutzerbetrieb

Der Übergang vom Ein- zum Mehr-Benutzerbetrieb von PAX wird durch die Termination von *Shell* eingeleitet. Der *Team-Administrator* teilt *Conf* die Termination dieses Prozesses mit (genauer: *Conf* wartet auf die Termination von *Shell*). *Conf* wird dann seinerseits alle notwendigen Maßnahmen zur Konfiguration des Mehr-Benutzerbetriebs veranlassen.

Die erste der zu treffenden Maßnahmen von *Conf* besteht darin, nochmals einen *Shell* zu erzeugen, dessen einzige Aufgabe darin besteht, ein Kommando-File zu interpretieren. In diesem File sind konfigurationsspezifische Anweisungen enthalten. Diese Anweisungen, die durch übliche Dienstprogramme repräsentiert sind, führen u.a. zur Konfiguration des *Block-I/O-Systems*, indem die verschiedenen realen/logischen Plattenbereiche bekanntgegeben ("*gemountet*") werden.

Mit der Termination dieses *Shells* erzeugt *Conf* den Systemprozeß *Init*, der seinerseits das Programm */etc/init* zur Ausführung bringt. *Init* repräsentiert dieselbe Funktionalität, wie sie für */etc/init* unter UNIX bekannt ist (siehe auch [Kernighan, McIlroy 1979]). Das bedeutet in PAX, für jedes angeschlossene Terminal einen *Login*-Prozeß zu erzeugen. Welches Terminal in dieser Konfigurationsphase betrachtet werden soll, entnimmt *Init* dem File */etc/ttys*. Bild 6.3 skizziert die zu diesem Zeitpunkt gültige Mehr-Benutzerkonfiguration von PAX.

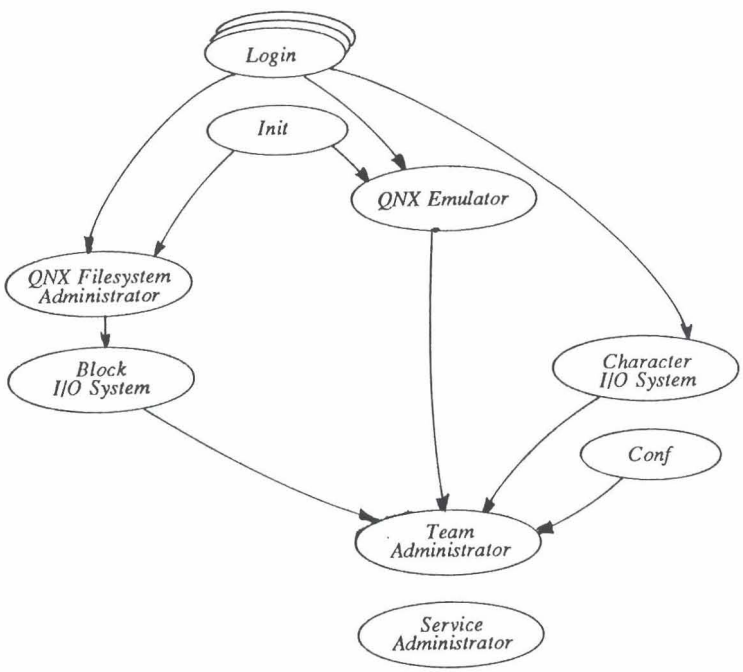

Bild 6.3: PAX im Mehr-Benutzerbetrieb

Login wartet in dieser Phase der Konfiguration des Systems auf Eingabe vom *Character-I/O-System*, d.h. daß ein Benutzer von PAX mit der Login-Sequenz beginnt. Mit Beginn der

Login-Phase – ein Benutzer hat Zeichen an einem bestimmten Terminal eingegeben – bedient sich *Login* der in dem File */etc/passwd* enthaltenen Informationen, um zu entscheiden, ob der Benutzer autorisiert ist, mit PAX zu arbeiten. Sollte der Benutzer autorisiert sein, ändert *Login* seine Repräsentation, indem das in */etc/passwd* ausgewiesene Dienstprogramm zur Ausführung gebracht wird. Üblicherweise wird dieses Dienstprogramm der für den Benutzer zur Verfügung gestellte *Shell* sein.

Die Aufgabe von *Init* besteht weiterhin darin, nach jeder Termination von *Login*, diesen Prozeß erneut zur Ausführung für das ihm zugeordnete Terminal zu erzeugen und damit eine neue Login-Phase zu ermöglichen. Die Termination von *Init* bedeutet den Übergang vom Mehr- in den Ein-Benutzerbetrieb. Im Ein-Benutzerbetrieb wird *Init*, ähnlich wie bei *Login*, erneut von *Conf* zur Ausführung gebracht, woraufhin *Init* wiederum mit der Erzeugung von *Login* für jedes in */etc/ttys* verzeichnete Terminal beginnt. Die zwischenzeitige Änderung von */etc/ttys*, während des Ein- oder Mehr-Benutzerbetriebs, ermöglicht es, verschiedene Terminal-Konfigurationen zum Einsatz kommen zu lassen.

6.1.3. Zusätzliche Systemkomponenten

Mit den bisher vorgestellten Systemkomponenten von PAX ist ein vollständiger Mehr-Benutzerbetrieb möglich. Die den Benutzern von PAX zur Verfügung gestellten Dienstprogramme entsprechen vollständig denen von QNX[46].

Zusätzlich zu den durch QNX ermöglichten Funktionalitäten, können unter PAX weitere funktional bedeutsame Systemprozesse zum Einsatz gebracht werden. Diese Systemprozesse könnten bereits im Zuge der Interpretation des Kommando-Files, beim Übergang vom Ein- zum Mehr-Benutzerbetrieb, oder erst im Mehr-Benutzerbetrieb erzeugt werden. Unabhängig davon, wann diese Systemprozesse erzeugt worden sind, ist deren Zerstörung bzw. dynamische Rekonfiguration jederzeit möglich. Diese Systemprozesse sind im einzelnen der

- *Timer*
  Der *Timer* stellt Funktionalitäten zur Verfügung, die eine zeitkontrollierte Ausführung von Prozessen ermöglichen. Auf Grundlage dieser Funktionalitäten können *Timeouts* für Systemdienste definiert und *Alarm-Clocks* für Prozesse eingerichtet werden. Hierzu nimmt der *Timer* die vom MOOSE-Kernel zur Systemebene propagierten *Clock-Interrupts* an. Die kleinste Auflösung der Abstände zwischen zwei Zeitsignalen des *Timers* liegt für PAX bei 100 Millisekunden. Die Differenz zwischen zwei direkt aufeinanderfolgende *Clock-Interrupts* liegt jedoch bei 10 Millisekunden. In diesem Abstand zählt der MOOSE-Kernel seinen internen *Clock-Tick* weiter, dessen Wert von den Prozessen abgefragt werden kann.
- *Scheduler*
  Der MOOSE-Kernel realisiert nur eine sehr einfache *round-robin* Strategie zur zyklischen Vergabe des realen Prozessors an *Teams*. Erst der *Scheduler* realisiert

[46] Die Ausnahme stellen Dienstprogramme dar, die auf QNX-interne Datenstrukturen zurückgreifen. Von solchen Dienstprogrammen sind jedoch sehr wenige unter QNX zu finden. Als typischer Vertreter ist *task* zu nennen, der durch ein PAX-spezifisches Dienstprogramm (*ps*) ersetzt worden ist.

komplexere Strategien zur Selektion des nächsten ausführbaren *Teams*. Damit ist der MOOSE-Kernel von der Problematik eines applikationsspezifischen *Schedulings* weitestgehend entlastet. Durch die Abkapselung der *Schedule*-Strategien durch einen eigenen Systemprozeß, ist es unter PAX möglich, diese Stratgien zur Laufzeit zu ändern und damit eine optimale Anpassung an das jeweilige Benutzerprofil von PAX zu erzielen.

- *Swapper*
  Der *Swapper* stellt eine weitere zentrale Systemkomponente unter PAX dar, die dynamisch das Laufzeitverhalten eines Systems zugunsten der Ausführung komplexer Applikationen verändert. Erst wenn der frei zur Disposition stehende Speicherplatz auf ein bestimmtes Minimum abgesunken ist, braucht der *Swapper* unter PAX geladen und gestartet zu werden. Dieses Minimum bestimmt sich durch die Komplexität des *Swappers* und durch den von ihm benötigten Speicherplatz zur Erfüllung seiner Aufgabe. Die von der Aus-/Einlagerung betroffenen Prozesse sind durch das sie jeweils umschließende *Team* vorgegeben. Das bedeutet, die kleinste vom *Swapper* betrachetete Einheit stellt ein *Team* dar. Die Formulierung von Prozeßgruppen – d.h. eine bestimmte Anzahl von *Teams*, die einen bestimmten Applikationskomplex zugeordnet sind – ermöglicht es dem *Swapper*, größere, applikationsorientierte Einheiten zur Aus-/Einlagerung zu betrachten. In diesem Sinne können somit vollständige Prozeßgruppen vom *Swapper* aus- bzw. eingelagert werden.
- *Accounter*
  Der Nachweis des Laufzeitverhaltens bestimmter Prozesse unterstützt im besonderen Maße die Bewertung der Leistungsfähigkeit dieser Prozesse und damit des durch diese Prozesse konstituierten Systems. Für diesen Nachweis werden für PAX verschiedene Strategien zugrunde gelegt, die jeweils durch einen entsprechenden *Accounter*-Prozeß repräsentiert werden. Diese Strategien betrachten jeweils verschiedene Ansatzpunkte zur Aufzeichnung des Schnittstellenverhaltens eines Prozesses. Auf der untersten Ebene können die vom MOOSE-Kernel zur Verfügung gestellten Mechanismen zur Interprozeßkommunikation betrachtet werden. Hiermit wird es ermöglicht, die Dauer eines *Rendezvous* zwischen dem *Rendezvous-Client* und dem *Rendezvous-Server* festzustellen. Weitere Ansatzpunkte ergeben sich an der Empfangsschnittstelle der Systemprozesse unter PAX. An dieser Schnittstelle wird aufgezeichnet, welcher Prozeß welche Dienste in Anspruch nimmt. Das gleiche Verfahren kann jedoch auch auf beliebige Prozesse eines bestimmten Systemkomplexes angewendet und somit applikationsspezifisch ausgenutzt werden. Die oberste Ebene von *Accounting* unter PAX verzeichnet, wann und wie lange Benutzer mit dem System gearbeitet haben. Die Kombination aller Strategien ist möglich, so daß für einen bestimmten Benutzer alle wesentlichen Systemaktivitäten aufgezeichnet und im nachhinein analysiert werden können.
- *Deblocker*
  Der *Deblocker* unterstützt den MOOSE-Kernel darin, die Kommunikationsaktivitäten zu nicht mehr existenten Prozessen zu unterbinden. Dieser Prozeß wird als Repräsentant aller terminierten Prozesse von PAX in Erscheinung treten und damit generell als

Kommunikationspartner zur Verfügung stehen. Die Kommunikation mit dem *Deblocker* wird jedoch sofort eine Ausnahmesituation signalisieren, die domänenspezifisch behandelt werden kann.

- *Deadlock-Server*
  Der *Deadlock-Server* untersucht sämtliche Kommunikationsbeziehungen zwischen den gegenwärtigen Prozessen von PAX. Er versucht hierbei *Deadlocks* festzustellen, diese geeignet aufzulösen und ermöglicht die Aufzeichnung dieser Ausnahmesituation zwecks späterer Analyse.
- *Filesystem-Administrator*
  Neben dem durch den QNX-*Filesystem-Administrator* repräsentierten Filesystem, steht unter PAX ein zu UNIX kompatibles Filesystem zur Verfügung. Der dazu entsprechende *Filesystem-Administrator* stellt nicht nur die UNIX-typische Struktur eines Filesystems zur Verfügung – wenn 4.2BSD einmal ausgenommen wird –, seine Dienstschnittstelle umfaßt sämtliche *System-Calls* die in UNIX zur File-Verwaltung angewendet werden können. Ein Dienstprogramm (*Filer*) ermöglicht den Transfer von Dateien zwischen dem QNX- und UNIX-orientierten Filesystem.
- *Remote Filesystem-Administrator*
  PAX stellt Mechanismen zum *remote file access* zur Verfügung. Diese Mechanismen werden über einen bestimmten Prozeßkomplex realisiert, der die Dienste des UNIX-spezifischen *Filesystem-Administrators* als *remote procedure calls* auffaßt. Das zugrunde gelegte Modell führt dazu, daß für die lokale Inanspruchnahme der Dienste des *Filesystem-Administrators* kein zusätzlicher Laufzeitverlust für die dienstanfordernden Prozesse entsteht. Die für den *Remote Filesystem-Administrator* in Anwendung gebrachte Netzwerkschnittstelle basiert auf dem in [Bormann 1985] skizzierten System. Die Integration des *Remote Filesystem-Administrators* unter PAX erfolgt ohne Auswirkungen auf den jeweiligen lokalen *Filesystem-Administrator*. In dieser Hinsicht werden die Mechanismen zum *remote file access* nach ähnlichen Prinzipien in PAX integriert, wie es mit der *Newcastle Connection* [Brownbridge et al. 1982] erfolgt ist. *Remote file access* wird unter PAX durch Systemprozesse zur Verfügung gestellt, die hierarchisch eindeutig über dem *Filesystem-Administrator* angeordnet sind.
- *Pipe-Administrator*
  Pipes sind die unter UNIX üblichen Mechanismen zur Interprozeßkommunikation. Ihr Einsatz ist sinnvoll in Applikationen, bei denen eine bestimmte Gruppe von Prozessen als Einheit an der Verarbeitung eines "geradlinigen" Datenstroms beteiligt ist. Typische Beispiele ergeben sich hierbei durch den *Shell*. Die Mechanismen unter PAX zur Interprozeßkommunikation über *pipes* sind durch einen eigenen Systemprozeß abgekapselt. Der *Pipe-Administrator* ist dieser Systemprozeß und stellt dieselben, wie unter UNIX, mit *pipes* verbundenen Funktionalitäten zur Verfügung.
- *Window-Manager*
  PAX ist ein Mehr-Benutzer- und Mehr-Prozeß-System. Die zuletzt genannte Klassifikation ermöglicht es, daß mehrere Prozesse die Abarbeitung einer oder mehrerer Applikationen kontrollieren können. Für den Zustandsüberblick über einzelne Phasen

der Abarbeitung von Applikationen, ist eine geeignete, applikationsorientierte Strukturierung der Meldungen auf ein Ausgabegerät (Terminal) sehr hilfreich. Die gleiche Sichtweise gilt für die Eingabe von Daten. Mit jedem Prozeß kann ein eigenes Ein-/Ausgabefenster (*Window*) des Terminals assoziiert werden. Der *Window-Manager* unter PAX stellt hierzu grundlegende Funktionalitäten zur Verfügung, die die applikationsorientierte Behandlung von Ein-/Ausgabeströmen unterstützen. Bestimmte Systemprozesse von PAX – der *Accounter*, *Deblocker* und *Deadlock-Server* – machen von den Funktionalitäten des *Window-Managers* Gebrauch, um jederzeit eine geordnete Zustandsübersicht des Systems zu ermöglichen.

Mit diesen kurz skizzierten und zusätzlichen Administratoren von PAX wird die Menge der durch die QNX-Emulation zur Verfügung gestellten Dienste erheblich erweitert. Tabelle 6.1 zeigt die grundlegenden Dienste des MOOSE-Kernels, die sämtlichst für PAX ihre Gültigkeit besitzen.

Message Passing		*Port Management*		*Process Management*		*Trap-/Interrupt-Propagation*	
send		**attach**	(+)	**dispatch**	(+)	**serve**	(-)
receive		**detach**	(+)	**force**	(+)	**sever**	(-)
reply		**signal**		**hold**		**entry**	(-)
relay		**await**		**unhold**		**beam**	(-)
descriptor	(-)	**flush**		**value**	(+)	**accept**	(-)
transfer	(-)			**attribute**	(+)		
				map	(-)		
				setup	(-)		
				getpid	(-)		
				gettid	(-)		

Tabelle 6.1: *Kernel-Calls* von MOOSE

Die in dieser Tabelle mit "-" bezeichneten *Kernel-Calls* finden keine vergleichbaren Vertreter unter QNX. Die mit "+" bezeichneten *Kernel-Calls* sind ebenfalls unter QNX zu finden, jedoch dort mit geringerer Funktionalität verbunden.

Die durch die Systemprozesse von PAX zur Verfügung gestellten Dienste sind in Tabelle 6.2 genannt. Die in dieser Tabelle aufgezeichneten Dienste die unter UNIX keinen vergleichbaren Vertreter finden, sind mit "-" gekennzeichnet. Dienste, die unter MAX durch den *Domain-Administrator* erbracht werden, sind mit "*" gekennzeichnet.

Zur genauen Beschreibung der genannten *Kernel-* und *System-Calls* sei auf [Schroeder et al. 1986] verwiesen.

Administratoren					
Service		*Team*		*Filesystem*	*Pipe*
service	(-)	**fork**		**creat**	**pipe**
provider	(-)	**task**	(-)	**open**	
adopt	(-)	**exit**		**close**	
inherit	(-)	**kill**		**read**	
		getppid		**write**	
		expand		**lseek**	
		overlay	(-)	**link**	
		wait	(*)	**unlink**	
		setgid	(*)	**mkdir**	
		getgid	(*)	**rmdir**	
		seteuid	(*)	**chdir**	
		geteuid	(*)	**mknod**	
		setpgrp	(*)	**fstat**	
		getpgrp	(*)	**sync**	

Tabelle 6.2: *System-Calls* von PAX

6.1.4. Fallstudie spezieller Funktionalitäten des MOOSE-Kernels

In diesem Abschnitt sollen spezielle Funktionalitäten des MOOSE-Kernels anhand von Beispielen skizziert werden. Diese Beispiele finden in der hier vorgestellten Form ihren Einsatz unter PAX. Dabei werden zwar die betreffenden Systemprozesse zum Teil erheblich mehr an Funktionalität aufweisen, als mit den betrachteten Beispielen aufgezeigt wird. Dennoch sind die vorgestellten Prinzipien allgemeingültig.

Der Übersichtlichkeit wegen, wird in den skizzierten Beispielen das Gelingen der jeweiligen *Kernel-/System-Calls* vorausgesetzt. Die verwendete Programmiersprache ist C [Kernighan, Ritchie 1978]. Eine genaue Beschreibung der verwendeten *Kernel-/System-Calls* ist [Schroeder et al. 1986] zu entnehmen.

6.1.4.1. Monitoren von Kommunikationsaktivitäten

Das Monitoren von Kommunikationsaktivitäten wird durch den MOOSE-Kernel dadurch unterstützt, indem sich ein entsprechender Monitor-Prozeß zwischen dem *Rendezvous-Client* und dem *Rendezvous-Server* schaltet. Dieser Monitor-Prozeß wird repräsentativ für den *Rendezvous-Server* alle Nachrichten annehmen, diese vorverarbeiten und anschließend zu dem eigentlichen Empfänger weiterreichen. Die in Abbildung 6.4 dargestellte Programmsequenz skizziert die prinzipielle Arbeitsweise eines solchen Monitor-Prozesses – genauer gesagt, des *Accounters* unter PAX.

Die Argumente für *monitor* geben jeweils den zu monitorenden *Rendezvous-Server* (*proc*) und die Adresse (*strat*) einer Prozedur an, die die jeweilige Strategie zum Monitoren

```
 1    #include <proctypes.h>
 2
 3    monitor (proc, strat)
 4    int proc, (*strat)();
 5    {
 6          register int moni, old, pid, attr;
 7
 8          moni = getpid();
 9          attr = attribute(0, PTASK, 0);
10
11          attribute(0, attr | PTASK | PSERVER, 1);
12
13          old = map(proc, moni);
14
15          for (;;) {
16                pid = receive(0, 0, 0);
17
18                attribute(0, attr | PTASK | PINIDIVISIBLE, 1);
19
20                map(proc, old);
21                relay(pid, proc);
22                map(proc, moni);
23
24                attribute(0, attr | PTASK, 1);
25
26                (*strat)(pid);
27          }
28    }
```

Bild 6.4: Prinzipielle Struktur eines Monitor-Prozesses

implementiert. Das Adoptieren des *Rendezvous-Servers* erfolgt durch *map* in Zeile 13. Die Anweisungen von Zeile 20 bis 22 ermöglichen das Weiterreichen der empfangenden Nachricht an den adoptierten *Rendezvous-Server*. Für diese Maßnahme muß der Monitor-Prozeß die originale Beziehung zwischen *Rendezvous-Client* und *Rendezvous-Server* wieder herstellen, da er sonst durch das *relay* in Zeile 21 dieselbe Nachricht nocheinmal empfangen würde.

In der dargestellten Programmskizze wird eine wesentliche Eigenschaft des MOOSE-Kernels verdeutlicht, nämlich die problemorientierte Unteilbarkeit von Prozessen zu ermöglichen. Das Weiterreichen der empfangenden Nachricht in dem skizzierten Beispiel, stellt einen kritischen Abschnitt dar. Würde der Monitor-Prozeß während dieser Phase nicht unteilbar laufen, so bestände die Möglichkeit, daß er nicht alle an *proc* gerichtete Nachrichten mitgeteilt bekommt. Dies wäre dann der Fall, wenn, zwischen Zeile 20 und 22, die Zeitscheibe des Monitor-Prozesses abläuft und vor seiner erneuten Reaktivierung ein Prozeß aktiv wird, der eine Nachricht zu *proc* übermittelt. Zur Lösung dieser Problematik erfolgt die Markierung des kritischen Bereiches durch *attribute* in Zeile 18 und 24. Während dieses Abschnittes wird der Monitor-Prozeß die Zeitscheibe seines *Teams* nicht abgeben.

6.1.4.2. Garbage-Collector für terminierte Rendezvous-Server

Eine wesentliche Funktionalität unter PAX nimmt der *Deblocker* wahr. Er sorgt dafür, daß alle Kommunikationswünsche zu bereits terminierten Prozessen abgefangen werden. Die Vorgehensweise des *Deblockers* ist vergleichbar mit der des Monitor-Prozesses. Der *Deblocker* empfängt repräsentativ für einen terminierten *Rendezvous-Server* die übermittelte Nachricht. Hierbei braucht der *Deblocker* jedoch nur den Empfang der Nachricht als das

Signal zu betrachten, eine entsprechende Ausnahmesituation behandeln zu müssen bzw. die domänenspezifische Propagation dieser Ausnahmesituation einzuleiten. Die Interpretation der empfangenden Nachricht ist nicht notwendig. Die in Abbildung 6.5 gezeigten Instruktionssequenzen skizzieren die prinzipielle Struktur des *Deblockers*.

```
 1  #include <proctypes.h>
 2  #include <force.h>
 3
 4  static dead (ppid, strat)
 5  int ppid, (*strat)();
 6  {
 7       register int pid;
 8
 9       for (;;) {
10            pid = receive(0, 0, 0);
11            (*strat)(pid);
12       }
13  }
14
15  deblock (strat)
16  int (*strat)();
17  {
18       register int pid, deb;
19       register unsigned attr;
20
21       deb = task(PSERVER | PBACKGROUND, 128, dead, strat);
22
23       attribute(0, attribute(0, PTASK, 0) | PTASK | PEXIT | PFORK, 1);
24       attribute(0, attribute(0, PTEAM, 0) | PTEAM | PINFORM, 1);
25
26       for (;;) {
27            pid = receive(0, 0, 0);
28
29            attr = attribute(pid, PTASK, 0);
30
31            if (!(attr & PBROADCAST))
32                 force(pid, FREADY);
33            else {
34                 map(pid, attr & PEXIT ? deb : pid);
35                 relay(pid, 0);
36            }
37       }
38  }
```

Bild 6.5: Prinzipielle Struktur des *Deblockers*

Zur Unterstützung der Deblockierung von Prozessen bedient sich der *Deblocker* eines Hilfsprozesses, der in Zeile 21 durch *task* erzeugt wird. Die Primitive *task* erzeugt in diesem Beispiel einen *captive*, der als *Server*-Prozeβ operiert und dessen Stack 128 *Bytes* groβ ist. Der *captive* wird durch die Prozedur *dead* repräsentiert und durch die *process ID deb* eindeutig identifiziert. Der Prozedur *dead* wird mit Aktivierung des *captives* zwei Argumente übergeben. Das implizite Argument *ppid* gibt die *process ID* des erzeugenden Prozesses an, und das Argument *strat* wird explizit durch *task* übergeben.

Der *Deblocker* besteht demnach aus zwei Prozessen, die demselben *Team* zugeordnet sind. Der Prozeβ *deblock* sorgt dafür, daβ alle terminierten Prozesse durch den Prozeβ *dead* repräsentiert werden. Die Termination von Prozessen wird *deblock* über den entsprechenden *Broadcast* mitgeteilt. Hierzu werden in Zeile 23 und 24 die entsprechenden Vorkehrungen

getroffen, damit *deblock* als *Broadcast-Server* vom MOOSE-Kernel selektiert wird.

In Zeile 34 wird, je nach aktivem *Broadcast*, die *Receive-Map* für den jeweiligen *Broadcast-Client* (der terminierte/erzeugte Prozeβ) entsprechend definiert. Die Termination eines Prozesses führt hierbei dazu, *dead* als Repräsentant des terminierenden Prozesses *pid* zu definieren. Die Erzeugung eines Prozesses führt dazu, eine evtl. Abbildung auf *dead* rückgängig zu machen. Durch das *relay* in Zeile 35 wird der jeweilige *Broadcast* weiter fortgeführt.

Deblock kann als Prozeβ generell Nachrichten von allen anderen Prozessen des Systems zugestellt bekommen. Das bedeutet somit, daβ *deblock* nicht nur aufgrund eines *Broadcasts* reaktiviert zu werden braucht. Es ist zu beachten, daβ mit einem *Broadcast* für einen bestimmten Prozeβ (der *Broadcast-Client*) mehrere *Rendezvous* ermöglicht werden. Das *Rendezvous* an sich gibt somit keinen Aufschluβ über den vom *Rendezvous-Server* angeforderten Dienst. Demzufolge stellt *deblock* sicher, daβ die Manipulation der *receive map* (in Zeile 34) nur in dem Fall durchgeführt wird, wenn sich der *Rendezvous-Client* in einem *Broadcast* befindet und damit gleichzeitig einen *Broadcast-Client* repräsentiert. In Zeile 31 führt *deblock* hierzu die entsprechende Überprüfung durch. Sollte kein *Broadcast-Client* an dem *Rendezvous* beteiligt sein, so wird das *Rendezvous* durch *force* abgebrochen. Der Prozeβ *pid* ist in diesem Zusammenhang der *Rendezvous-Client* und wird daraufhin wieder reaktiviert. Das Resultat des entsprechenden *Kernel-Calls* von *pid* – aufgrund des *Rendezvous* wird dies ein *send* gewesen sein – zeigt den Abbruch des *Rendezvous* (durch -1) an.

6.1.4.3. Behandlung propagierter Traps

Die durch Systemprozesse ermöglichte Behandlung von *Hardware-Traps*, wie sie in dem nachfolgenden Beispiel vorgestellt wird, stellt die Grundlage zur Emulation von *System-Calls* dar. Der Emulator bindet sich als *Trap-Server* an die Trap-Vektoren an, über die *System-Calls* abgesetzt werden – Trap-Instruktionen werden oftmals eingesetzt, um den mit einem *System-Call* verbundenen Übergang vom *User-* zum *Kernelmode* zu erzielen. Als Beispiel eines *Trap-Servers* unter PAX, wird in Abbildung 6.6 der Rumpf eines Systemprozesses skizziert. Dieser Systemprozeβ gestattet den Laufzeit-*Trace* für einen bestimmten Prozeβ.

Die Anbindung an den betreffenden Trap-Vektor des realen Prozessors (in diesem Beispiel der Intel i8086/88) erfolgt in Zeile 14, und das Einschalten des *Trace-Modes* des betreffenden Prozesses *proc* wird durch die Anweisung von Zeile 16 erreicht. Der *Trace-Mode* wird direkt durch den betrachteten realen Prozessor unterstützt. Dieser Prozessor wird automatisch nach jeder durch einen Prozeβ ausgeführten Instruktion einen *Hardware-Trap* signalisieren. Die entsprechend aufgesetzte *Trap-Message* wird in Zeile 19 empfangen und evtl. in Zeile 31 an den den Trap auslösenden Prozeβ zurückgegeben. Zeile 25 veranschaulicht, wie der Prozessorstatus eines Prozesses manipuliert werden kann – in diesem Fall wird der *Trace-Mode* für den Prozeβ zurückgesetzt. Das *sever* in Zeile 34 dient dazu, daβ sich der Systemprozeβ von dem Trap-Vektor VBPTTRP abbindet und nicht mehr zur Behandlung des entsprechenden Traps bereitsteht.

```
 1  #include <types.h>
 2  #include <trap.h>
 3  #include <port.h>
 4  #include <force.h>
 5  #include <vector.h>
 6  #include <i8086.h>
 7
 8  trace (proc, strat)
 9  int proc, (*strat)();
10  {
11        register int gate, pid;
12        trap_msg msg;
13
14        gate = serve(VBPTTRP, MDOMAIN);
15
16        force(proc, FTRACE);
17
18        for (;;) {
19              pid = receive(0, &msg, sizeof(msg));
20
21              if (pid == -1)
22                    break;
23
24              if (pid != proc)
25                    msg.tm_ups &= ~TFLAG;
26              else {
27                    if ((*strat)(pid, &msg) == -1)
28                          continue;
29              }
30
31              reply(pid, &msg, sizeof(msg));
32        }
33
34        sever(gate);
35  }
```

Bild 6.6: Ein *Trap-Server* unter PAX

6.1.4.4. Behandlung propagierter Interrupts

Zur Veranschaulichung der Behandlung von Interrupts unter PAX, wird nachfolgend das *Gate-Server*-Modell zugrunde gelegt. Der *Gate-Server* wird explizit erzeugt und mit der Behandlung eines bestimmten Interrups beauftragt. Der *Interrupt-Server* wird üblicherweise die Schnittstelle zu den Benutzerprozessen repräsentieren und sich mit dem *Gate-Server* über einen *Port* synchronisieren. Das in Abbildung 6.7 dargestellte Programmstück zeigt, mit welchen elementaren Hilfsmitteln ein *Interrupt-Handler* durch einen *Gate-Server* abgekapselt und die Kommunikation zwischen einem *Gate-Server* und seinem *Interrupt-Server* ermöglicht werden kann.

Der Kommunikationspunkt, d.h. der entsprechende *Port*, wird in Zeile 19 belegt. Die Anbindung an einen Interrupt-Vektor erfolgt durch das *serve* in Zeile 22. Das Attribut MTASK zeigt hierbei an, daß das *Gate-Server*-Modell für die Behandlung eines propagierten Interrupts zugrunde gelegt wird. Die Erzeugung des *Gate-Servers* erfolgt in Zeile 21. Der *Gate-Server* wird demselben *Team* wie sein erzeugender *Interrupt-Server* zugeordnet und erhält von dem *Interrupt-Server* Argumente, die er zur Behandlung eines Interrupts benötigt. Die Anbindung des *Gate-Servers* an den entsprechenden Interrupt-Vektor erfolgt durch das *entry* in Zeile 24. Das Abbinden das *Gate-Servers* findet ebenfalls durch *entry* statt, und zwar

```
 1  #include <types.h>
 2  #include <proctypes.h>
 3  #include <port.h>
 4
 5  static gate_server (ppid, vect, handl)
 6  int ppid, vect, (*handl)();
 7  {
 8        accept(-1);
 9
10        for (;;)
11              accept((*handl)(vect));
12  }
13
14  interrupt_server (vect, strat, handl)
15  int vect, (*strat)(), (*handl)();
16  {
17        register int gate, proc, intr;
18
19        intr = attach(MSIGNAL);
20
21        proc = task(PBACKGROUND | PINDIVISIBLE, 256, gate_server, vect, handl);
22        gate = serve(vect, intr | MTASK);
23
24        entry(gate, proc, 1);
25
26        for (;;) {
27              await(intr);
28
29              if ((*strat)(vect) == -1)
30                    break;
31        }
32
33        entry(gate, 0, 0);
34
35        sever(gate);
36        kill(proc);
37  }
```

Bild 6.7: Ein *Interrupt-* und *Gate-Server* unter PAX

in Zeile 33.

Der *Interrupt-Server* erwartet ein Signal vom *Gate-Server*, das anzeigt, daß ein Interrupt aufgetreten ist. Hierzu synchronisiert sich der *Interrupt-Server* in Zeile 27 auf den mit dem *Gate-Server* assoziierten *Port*. Die Aktivierung des *Gate-Servers* direkt nach seiner Erzeugung, führt in diesem Beispiel zur Synchronisation auf einen eintretenden Interrupt. Hierzu dient das *accept* aus Zeile 8. Das Argument -1 zeigt an, daß mit der Synchronisation des *Gate-Servers* kein Signal zu seinem *Interrupt-Server* übermittelt werden soll.

Ein über *vect* signalisierter Interrupt reaktiviert den *Gate-Server*. Dieser wird daraufhin den entsprechenden *Interrupt-Handler handl* aktivieren (Zeile 11). Das Resultat des *Interrupt-Handlers* zeigt an, ob mit der Synchronisation des *Gate-Servers* auf einen weiteren Interrupt, durch das *accept* in Zeile 11, ebenfalls eine Kommunikation mit einem *Interrupt-Server* stattzufinden hat. Sollte die Kommunikation gewünscht und das Resultat des *Interrupt-Handlers* entweder 0 oder *intr* sein, so wird der *Interrupt-Server* vom Kernel reaktiviert und in Zeile 27 seine Ausführung wieder aufnehmen.

6.1.5. Die Leistungsfähigkeit und Komplexität von AX

Die Bewertung von AX gliedert sich in zwei Bereiche; zum einen in die Darstellung der Leistungsfähigkeit des MOOSE-Kernels und zum anderen in die Beschreibung der Komplexität, d.h. des Speicherplatzbedarfs der einzelnen Systemkomponenten von PAX. Die Leistungsfähigkeit des MOOSE-Kernels wird anhand von Laufzeiten der physikalisch geschützten kritischen Abschnitte (Interrupts sind unterbunden) und anhand von *Benchmarks* für die wesentlichen Kernelaktivitäten aufgezeigt.

6.1.5.1. Laufzeiten der kritischen Abschnitte

Der MOOSE-Kernel selbst stellt bezüglich aller Phasen seiner *Kernel-Calls* eine zur sequentiellen Ausführung auf einen realen Prozessor gebrachte Systemkomponente dar. Sobald ein *Kernel-Call* initiiert worden ist, wird dieser entweder vollständig beendet oder der initiierende Prozeß gibt während der Ausführung des *Kernel-Calls* explizt den Prozessor ab. In jedem Fall wird dafür Sorge getragen, daß *Kernel-Calls* nicht überlappt ausgeführt werden können.

Der Kernel stellt damit insgesamt einen Monitor im Sinne von [Hoare 1974] dar. Dennoch können während der Ausführung eines *Kernel-Calls* sehr wohl Interrupts vom realen Prozessor angenommen und zur Kernelebene propagiert werden. Über die *Event-List* wird hierbei für die entsprechende Synchronisation der Kernelaktivitäten erreicht. Der Schutz der *Event-List* selbst ist jedoch nur mit Hilfe von Synchronisationsmaßnahmen möglich, die vollständig das Auftreten weiterer Interrupts physikalisch unterbinden. Vergleichbares gilt im Zusammenhang mit der Prozeßumschaltung und mit bestimmten Aktivitäten der *Gate-Handler*. Die nachfolgende Tabelle 6.3 gibt hierzu einen Überblick, welchen Umfang und welche Laufzeiten die Phasen des vollständig (physikalisch) unteilbaren Ablaufs von Kernelaktivitäten haben.

Abschnitt		Laufzeit	Programmzeilen	
			C	Assembler
Event-List	Aufnahme	8	2	4
	Übertrag	8	3	4
	Austrag	16/20	3	7/8
Process-Switch		14	-	9
Gate-Handler		15	-	13

Tabelle 6.3: Laufzeiten physikalisch unteilbarer und kritischer Abschnitte

Die Laufzeiten in dieser Tabelle sind in Mikrosekunden angegeben. Wie zu ersehen ist, ergeben sich für die Dauer eines kritischen Abschnitts, der durch physikalische Unterbindung von Interrupts zu schützen ist, sehr geringe Werte. Die Reaktion auf Interrupts wird damit durch den Kernel auf ein Minimum verzögert. Den wesentlichen Anteil für die Dauer der physikalischen Unterbindung von Interrupts, wird die Organisation der zugrunde liegenden

Hardware bestimmen. Mit dem Einsatz von *Interrupt-Controllern* bzw. Prozessoren, die ein mehrschichtiges Interrupt-Modell zur Verfügung stellen, kann auf Grundlage des MOOSE-Kernels ein sehr effizientes und realzeitfähiges System zur Interrupt-Behandlung konstruiert werden.

6.1.5.2. Benchmarks

Zur Bewertung der Leistungsfähigkeit des MOOSE-Kernels sind verschiedene *Benchmarks* durchgeführt worden. Diese *Benchmarks* wurden auf drei Zielsysteme angesetzt: QNX, MOOSE und PAX (d.h. auf ein Emulationssystem für QNX). Allen gemeinsam ist derselbe zugrunde liegende Prozessor, ein Intel i8086 mit 8 MHz Taktfrequenz. Die Programme (Prozesse) zur Durchführung der *Benchmarks* stellten auf allen drei Systemen die einzigen Systemaktivitäten dar, so daß die gemessenen Werte ausschließlich die Leistungsfähigkeit des Kernels wiedergeben.

Zur Relativierung der Ergebnisse sei hinzugefügt, daß unter QNX ein *Clock-Interrupt* im Abstand von 50 Millisekunden erfolgt und unter PAX sowie beim MOOSE-Kernel im Abstand von 10 Millisekunden auftritt. Desweiteren ist zu bemerken, daß mit den *Benchmarks* für PAX gleichzeitig der durch die Emulation bedingte Laufzeitverlust eines *Kernel-Calls* bestimmt werden kann. Unter PAX werden die QNX-*Kernel-Calls* auf die des MOOSE-Kernels abgebildet. Die Differenz zwischen den für PAX und den für den MOOSE-Kernel bestimmten Werten repräsentiert den Emulations-*Overhead.*

Da wesentliche *Kernel-Calls* von QNX mit denen des MOOSE-Kernels identisch sind, können die gemessenen Werte für MOOSE auch auf QNX-Applikationen direkt übertragen werden. Dies ist jedoch nur dann möglich, wenn die QNX-Applikationen mit der entsprechenden MOOSE-*Library* gebunden worden sind. Der mit PAX verbundene Emulations-*Overhead* fällt in diesem Fall vollkommen heraus. Die *Benchmark*-Programme für den MOOSE-Kernel stellen gerade solche QNX-Applikationen dar.

Der *Benchmark* basiert darauf, eine bestimmte Anzahl (jeweils 10000) ausgewählter *Kernel-Calls* der betrachteten Zielsysteme auszuführen und zu bestimmen, wieviel Zeit für eine Testreihe verstreicht. Die Ergebnisse des ersten *Benchmarks* sind in Tabelle 6.4 niedergelegt. Dieser *Benchmark* gibt einen Überblick über die Laufzeiten der Mechanismen zur Interprozeßkommunikation von QNX, PAX und des MOOSE-Kernels. Um einen direkten Vergleich mit QNX zu ermöglichen, sind bei MOOSE beide in dem *Rendezvous* involvierten Prozesse einem eigenen *Team* zugeordnet.

Die Nachrichtengröße ist für jeden Durchlauf der Testreihe jeweils um 16 *Bytes* verringert worden. Es ist hierbei zu beachten, daß mit jedem *Rendezvous* zwei Transfervorgänge verbunden sind. Zum einen der Nachrichtentransfer bei *send* und zum anderen der Nachrichtentransfer bei *reply*. Die jeweilige Nachrichtengröße *bpt* ist somit zweimal während eines *Rendezvous* von Bedeutung.

Die Laufzeitverbesserung der Interprozeßkommunikation läßt sich für MOOSE im Vergleich zu QNX mit ca. 10 % bei 256 *bpt* und 11 % bei 0 *bpt* vermerken. Die Emulation der Mechanismen zur Interprozeßkommunikation von QNX bedeutet für PAX einen Laufzeitverlust zu QNX von ca. 12 % bei 256 *bpt* und 25 % bei 0 *bpt.*

bpt	QNX		MOOSE		PAX	
	adr	*dpr*	*adr*	*dpr*	*adr*	*dpr*
256	502	1990	539	1852	399	2501
240	512	1950	553	1808	399	2501
224	523	1910	565	1769	407	2455
208	534	1870	577	1731	412	2422
192	546	1830	590	1693	419	2381
176	558	1790	604	1653	426	2343
160	571	1750	619	1615	434	2299
144	584	1710	633	1578	441	2263
128	597	1675	649	1539	449	2223
112	613	1630	666	1500	458	2183
96	628	1590	684	1460	466	2143
80	645	1550	703	1421	474	2106
64	662	1510	724	1381	484	2066
48	680	1470	746	1340	499	2001
32	696	1435	766	1304	501	1994
16	716	1395	791	1264	513	1948
0	738	1355	816	1225	523	1909

bpt = Anzahl der *Bytes* pro Transfer (je einmal für *send* und *reply*)
adr = Anzahl der *Rendezvous* (*send-receive-reply*)
dpr = Dauer pro *Rendezvous* in Mikrosekunden

Tabelle 6.4: Laufzeiten von *Rendezvous*

In Tabelle 6.5 sind die Ergebnisse zweier *Benchmarks* verzeichnet. Einerseits wurde die minimale Laufzeit eines *Kernel-Calls* bestimmt. Hierzu sind für die einzelnen Systeme jeweils *Kernel-Calls* herangezogen worden, mit denen keine weiteren Verwaltungsmaßnahmen (z.B. Kontrolle des Zugriffsrechts, Blockierung/Deblockierung eines Prozesses, Datentransfer, usw.) verbunden sind, als das Liefern eines bestimmten Resultats. Andererseits sind die Laufzeiten expliziter, d.h. nicht durch *Clock-Interrupts* verursachter Prozeßwechsel verzeichnet.

Auch in diesem Beispiel wurde für MOOSE der Wechsel zwischen zwei *Teams* betrachtet, um einen Vergleich mit QNX durchführen zu können. Um die Dauer eines Prozeßwechsels zu bestimmen, ist ein Modell von zwei Prozessen zugrunde gelegt worden. Damit wird bei allen drei Systemen sichergestellt, daß ein Prozeßwechsel bei Anwendung von *time_slice* bzw. *dispatch* auch tatsächlich stattfindet[47]. Beide Prozesse geben explizit mit *time_slice* bzw.

[47] *Dispatch* führt nicht zur Blockierung des aufrufenden Prozesses, sondern es wird lediglich der nächste Prozeß zur Ausführung selektiert. Würde nur ein Prozeß für die Testreihe zur Verfügung stehen, hätte *dispatch* den Effekt, immer denselben (*dispatch* aufrufenden) Prozeß zu selektieren und damit keinen Prozeßwechsel zu bewirken. Mit zwei Prozessen für die Testreihe, die jeweils *dispatch* aufrufen, ist somit sichergestellt, daß ein Prozeßwechsel stattfinden muß.

Benchmark	QNX		MOOSE		PAX	
	Call	*dpc*	*Call*	*dpc*	*Call*	*dpc*
Call-Overhead	*get_ticks*	175	*getpid*	146	*get_ticks*	418
Process-Switch	*time_slice*	360	*dispatch*	190	*time_slice*	439

dpc = Dauer pro *Call* in Mikrosekunden

Tabelle 6.5: Laufzeiten von *Kernel-Calls* und Prozeβwechsel

dispatch die Kontrolle des Prozessors ab. Demzufolge wird für die gesamte Testreihe die zweifache Anzahl entsprechender *Kernel-Calls* ausgeführt. Die Laufzeiten der einzelnen Prozeβwechsel berechnen sich demnach dadurch, indem jeweils die Hälfte der von beiden Prozessen abgesetzten *Kernel-Calls* zugrunde gelegt wird.

Die Ergebnisse für die minimale Laufzeit eines *Kernel-Calls* zeigen im Vergleich zu QNX für MOOSE eine Laufzeitverbesserung von ca. 8.5 % und für PAX einen Laufzeitverlust von ca. 23 %. Eine entsprechende Analyse für die Dauer eines Prozeβwechsels ergibt für MOOSE eine Laufzeitverbesserung von ca. 51 % und für PAX einen Laufzeitverlust von ca. 12 %. Hierbei ist jedoch zu beachten, daβ unter QNX ein prioritätsgesteuertes *Scheduling* vor dem Prozeβwechsel stattgefunden hat. Da jedoch nur ein Prozeβ im System aktiv war (der *time_slice* aufrufende Prozeβ), gibt der gemessene Wert für QNX generell den *Schedule-Overhead* inkl. Prozeβwechsel an.

Ein interessanter Vergleich ergibt sich zwischen den Ergebnissen für die *Kernel-Calls* *getpid* und *dispatch* unter MOOSE. Die Differenz von 44 Mikrosekunden gibt einerseits die Laufzeit für den bei einem Prozeβwechsel notwendigen Austausch des Laufzeitkontextes wieder. Dieser Laufzeitkontext setzt sich aus dem Stack (beschrieben durch den *Context-Descriptor*) und dem globalen Datenbereich (beschrieben durch den *Image-Descriptor*) eines Prozesses zusammen. Andererseits ist in den 44 Mikrosekunden die Selektion des nächsten *Teams* aus dem *Dispatch-Set* mit enthalten. Diese Selektion kostet ca. 10 Mikrosekunden, so daβ der Wechsel des Laufzeitkontextes mit ca. 34 Mikrosekunden bestimmt werden kann.

Der nächste *Benchmark* ist spezifisch für den MOOSE-Kernel und PAX. Unter QNX können keine vergleichbaren *Benchmarks* gefahren werden, da dort nicht die entsprechenden Funktionalitäten des MOOSE-Kernels zur Verfügung stehen. Tabelle 6.6 skizziert die Ergebnisse im einzelnen. In dieser Tabelle sind die Laufzeiten (in Mikrosekunden) der durch *Gate-Handler* kontrollierten Aktivitäten des MOOSE-Kernels angegeben. Im Falle eines *Trap-Gates* gibt das gemessene Ergebnis die Laufzeiten des *Rendezvous* mit einem *Trap-Server* an. Die Kommunikation mit dem *Gate-Server* erfolgt immer implizit, d.h. die Identifikation des *Trap-Servers* richtet sich nach dem an einem *Trap-Gate* angebundenen Prozeβ. Im Falle eines *Modul-/Task-Gates* sind, je nach Ergebnis des *Interrupt-Handlers*, drei verschiedene Phasen bei der *Interrupt-Behandlung* zu beachten. Diese Phasen unterscheiden sich darin, ob keine, die implizite oder explizite Kommunikation mit dem *Interrupt-Server* stattfinden soll.

Server-Gate	**Propagation**		
	keine	implizit	explizit
Trap	-	691	-
Modul	112	247	330
Task	267	507	584

Tabelle 6.6: Laufzeiten der Trap-/Interrupt-Propagation

Der *Interrupt-Handler* kann mit seinem Resultat anzeigen, daß keine Kommunikation mit seinem *Interrupt-Server* stattfinden soll. In diesem Fall geben die gemessenen Werte den Laufzeit-*Overhead* des MOOSE-Kernels an (gesteuert durch den *Gate-Handler*), um

a) entweder nur den Adreβraum für den *Interrupt-Handler* ein- und auszublenden (*Modul-Gates*) oder

b) die Aktivierung und Blockierung des einen *Interrupt-Handler* abkapselnden *Gate-Servers* (*Task-Gates*) zu erreichen.

Das Resultat 0 des *Interrupt-Handlers* zeigt den Kommunikationswunsch des *Interrupt-Handlers* über den implizit mit dem *Modul-/Task-Gate* assoziierten *Port* an. Der MOOSE-Kernel braucht hierfür die Integrität des *Ports* nicht zu überprüfen.

Die Differenz von 135 Mikrosekunden zwischen impliziter und keiner Propagation des *Interrupt-Signals* bei *Modul-Gates*, gibt die effektive Laufzeit der direkten Propagation eines Interrupts an. Die direkte Propagation bedeutet, daβ keine Synchronisationsmaβnahmen stattfinden müssen und damit der Verwaltungs-*Overhead* der *Event-List* nicht in Erscheinung tritt. Auf der Grundlage von *Task-Gates* ergibt sich eine Differenz von 240 Mikrosekunden. Da auch in diesem Fall 135 Mikrosekunden zur Propagation eines *Interrupt-Signals* zugrunde gelegt werden müssen, ergibt sich ein Überhang von 105 Mikrosekunden. Diese Differenz entspricht dem Verwaltungs-*Overhead* der *Event-List*. Der *Benchmark* auf Grundlage der *Task-Gates* ist so konzipiert, daβ jede Propagation eines *Interrupt-Signals* synchronisiert werden muβ und damit die *Event-List* als zusätzlich zu verwaltender globaler Datenbestand mit einzubeziehen ist.

Ein Resultat ungleich -1 spezifiziert explizit den *Port*, über den die Kommunikation mit dem *Interrupt-Server* durchgeführt werden soll. Hierzu ist vom MOOSE-Kernel jedoch die Integrität des angegebenen *Ports* sicherzustellen, was sich durch den gemessenen Leistungsverlust niederschlägt. Der Laufzeitverlust ergibt sich hierbei jeweils aus der Differenz zwischen den Werten der expliziten und impliziten Propagation. Dabei läβt sich eine Differenz von ca. 77/78 Mikrosekunden feststellen. Diese Differenz wird, unabhängig des zugrunde gelegten Modells zur Propagation von Interrupts, für denselben Prozessor immer gleich sein, da sich alle Modelle auf dieselben Kommunikations- und Synchronisationsmechanismen des MOOSE-Kernels beziehen.

6.1.5.3. Komplexität der einzelnen Systemkomponenten

Neben der Leistungsfähigkeit stellt die Komplexität eines Systems einen weiteren wesentlichen Bewertungsansatz dar. Wenn software-technische Aspekte einmal in den Hintergrund gestellt werden, so ist die Anzahl der Programmzeilen oftmals ein wesentliches Indiz dafür, daß ein Programmsystem leicht bzw. schwer wartbar oder überschaubar ist. Der Speicherplatzbedarf gibt Aufschluß darüber, welche Anforderungen an einen realen Prozessor zu stellen sind, um eine Ausführung des Programmsystems zu ermöglichen. Auch Techniken wie *Paging* oder *Swapping* werden die Ausführung eines Programmsystems dann nicht ermöglichen, wenn der reale Prozessor einen zu kleinen logischen Adreßraum für den das Programmsystem kontrollierenden Prozeß zur Verfügung stellt (siehe hierzu u.a. [Mueller et al. 1980]).

Für die Bewertung der Komplexität bestimmter Systemkomponenten von PAX ist der Vergleich mit QNX angebracht, da beide Systeme im Personal-Computer-Bereich eingesetzt werden. Hierbei können jedoch nur wenige Systemkomponenten von PAX in den Vergleich mit eingebracht werden, da QNX z.T. erhebliche strukturelle Unterschiede zu PAX aufweist. Tabelle 6.7 zeigt hierzu die Größen (in K-*Bytes*) der betrachteten Systemkomponenten.

Komponente	**Komplexität**	
	PAX	QNX
Kernel *Process-Management*	30	21
Filesystem-Management	21	23
Character-I/O-System	27	25

Tabelle 6.7: Komplexitätsvergleich von PAX und QNX

Der Vergleich mit QNX zeigt demnach, daß PAX allgemein einen größeren Speicherplatzbedarf aufweist. Hierbei ist jedoch zu berücksichtigen, daß, mit Ausnahme des *Filesystem-Managements*, alle in diesem Vergleich betrachteten Systemkomponenten von PAX ein größeres Maß an Funktionalität aufweisen, als die betreffenden Systemkomponenten von QNX. Im einzelnen ist dabei hervorzuheben, daß

- der MOOSE-Kernel, im Gegensatz zu QNX, Mechanismen zur Propagation von Traps/Interrupts beinhaltet. Insbesondere sind zusätzliche Tabellen im MOOSE-Kernel angesiedelt, die u.a. das zu QNX verbesserte Laufzeitverhalten begründen;
- die *Process-Management* Komponente betrachtet unter PAX die Verwaltung von *Teams* (durch den *Team-Administrator*) und unter QNX nur die Verwaltung einzelner Prozesse (durch den *Task-Administrator*). PAX stellt allgemein ein flexibleres Prozeßmodell zur Verfügung als QNX;

- das *Character-I/O-System* – unter QNX durch den *Device-Administrator* repräsentiert – zeichnet sich unter PAX durch seine im Vergleich zu QNX sehr hohe Portabilität aus, insbesondere wenn verschiedene Prozessortypen (für AX der i8086 und mc68000) betrachtet werden. Die Adaption dieses Systems für MAX beschränkte sich lediglich auf den Austausch eines sehr kleinen Bereiches der Gerätesteuerung. Die damit verbundene Portabilität ist jedoch nur durch die konsequente Modularisierung des *Character-I/O-Systems* ermöglicht worden, womit sich zwangsläufig – zumindest auf Grundlage des angewendeten Software-Entwicklungssystems – eine höhere Komplexität der betreffenden Systemkomponenten ergibt.

Neben den im Vergleich mit QNX betrachteten Systemkomponenten sind mit PAX noch weitere Systemkomponenten verbunden, die zur funktionalen Anreicherung eines bestimmten Basissystems eingesetzt werden. Diese zusätzlichen Systemkomponenten erlauben es, ein Mehr-Benutzersystem, vergleichbar mit UNIX Version 7 [Kernighan, McIlroy 1979], aufzubauen. Tabelle 6.8 gibt die Komplexität aller Systemkomponenten von PAX an.

Komponente		**Komplexität**			
		Code	Daten	Total	Zeilen
Kernel		8988	5705	14693	4363
Service-Administrator		1798	5314	7112	333
Team-Administrator		10593	5542	16135	3290
QNX-Emulator		9376	4904	14280	2073
QNX-Filesystem-Administrator		15723	5444	21167	-
Conf		1843	184	2027	107
Block-I/O-System		13191	20172	33363	2907
Character-I/O-System		18128	6067	24195	3534
Init		3966	485	4451	165
Timer	(+)	4897	329	5226	852
Scheduler	(+)	4110	268	4378	1494
Swapper	(+)	9948	1673	11621	1429
Filesystem-Administrator	(+)	21115	33395	58510	4589
Remote Filesystem-Administrator		18714	6967	25681	878
Pipe-Administrator	(+)	7233	550	7783	1229
Window-Manager		25144	6433	31577	4538

Tabelle 6.8: Komplexität der Systemkomponenten von PAX

Der Einsatz der mit "+" gekennzeichneten Komponenten führt, über entsprechende *Compatibility-Libraries*, zu einem mit UNIX Version 7 kompatiblen System. Die Zahlen für Code, Data und Total geben die *Byte*-Größe des jeweiligen Segmentes an.

6.2. Bezug zu vergleichbaren Systemen

Der Vergleich mit QNX ermöglichte die Bewertung von PAX auf technischer Ebene. Laufzeiten bestimmter *Kernel-Calls* sowie die Komplexität verschiedener Systemkomponenten sind miteinander verglichen worden.

Verschiedene mit MOOSE identifizierte Konzepte konnten jedoch nicht vergleichend mit QNX dargestellt werden. Diese sind im wesentlichen

- die Identifikation von Prozessen anhand der durch sie zur Verfügung gestellten Dienste;
- die Zuordnung eines gemeinsamen Adreβraumes für einen bestimmten Prozeβkomplex;
- die Modellierung von Prozeβdomänen;
- Möglichkeiten zur dynamischen Rekonfiguration eines prozeβorientierten Betriebssystems auf der Grundlage der Adaption von Diensten und Prozessen;
- die domänenspezifische Behandlung von Ausnahmesituationen auf Grundlage der Mechanismen zur Interprozeβkommunikation und des *Team*-Konzeptes des MOOSE-Kernels;
- die Abkapselung von *Interrupt-Handler* durch eigene Prozesse;
- die Propagation von Traps/Interrupts zu *Server*-Prozessen.

Anhand dieser kurz skizzierten Konzepte von MOOSE und deren Verwirklichung in PAX, soll eine vergleichende Diskussion mit anderen modernen Betriebssystemen stattfinden. Die in der Diskussion betrachteten Betriebssysteme basieren jeweils auf dem prozedur-, prozeβ- und objektorientierten Systementwurf. Entsprechend dieser Entwurfsprinzipien werden nachfolgend in derselben Reihenfolge verschiedene Vertreter aus dem Betriebssystembereich diskutiert. Die gewählte Reihenfolge entspricht dem "Stand der Kunst" in der Entwicklung auf dem Betriebssystembereich. Die ausgewählten und zur Diskussion gestellten Repräsentanten ermöglichen in dieser Hinsicht sehr gut die Einordnung von AX in die gegenwärtigen Forschungs- und Entwicklungsaktivitäten auf dem Betriebsssystembereich.

6.2.1. UNIX

Die Namensgebung des ersten einsatzfähigen Prototyps aus der MOOSE-Betriebssystemfamilie verdeutlicht den konkreten Bezug zu UNIX. AX bezieht sich hierbei jedoch nicht nur auf ein bestimmtes UNIX-System, sondern steht vielmehr stellvertretend für eine Menge von UNIX-Systemen. In diesem Sinne definiert AX die Familie von UNIX-ähnlichen Betriebssystemen. MOOSE definiert damit den übergeordneten Bezugsrahmen, unterschiedliche Betriebssysteme und nicht nur UNIX-ähnliche Systeme zusammenfassen zu können.

Die Einordnung von AX in die Menge der bekannten UNIX-Systeme läβt sich durch die Begriffe UNIX *look-alike* bzw. UNIX *work-alike* passend artikulieren (siehe u.a. [Fiedler

1983]). Als Klassifikationsmerkmale ergeben sich hierbei für diese beiden Begriffe

- *look-alike*
  Die Programmierschnittstelle eines Systems – die *System-Calls* bzw. ihre über *Compatibility-Libraries* erreichte Abbildung auf *System-Calls* eines bestimmten Basissystems – ist zu der des betrachteten UNIX-Referenzsystems identisch.
- *work-alike*
  Die Programmierschnittstelle eines Systems ist nicht identisch mit der des betrachteten UNIX-Referenzsystems, jedoch stehen typische Dienstprogramme des jeweiligen UNIX-Referenzsystems zur Verfügung. Das bedeutet somit, daß oberhalb der Shell-Kommandoebene die UNIX-Kompatibilität erreicht ist.

Der Bezugspunkt für beide Begriffe, d.h. das UNIX-Referenzsystem, ist jeweils durch ein bestimmtes "innovatives" UNIX-System vorgegeben. Da mit UNIX verschiedene Entwicklungsschritte verbunden waren (und sein werden), konnte dieser Bezugspunkt über einen längeren Zeitraum keinem UNIX-System zugeordnet werden. Dies erschwert die eindeutige Identifikation eines UNIX *look-/work-alikes*. Die zunehmende Bestrebung zur Standardisierung von UNIX – der Standard wird vornehmlich durch UNIX System V [Bell 1983] geprägt sein –, wird jedoch eine Möglichkeit darstellen, die Definition eines maßgeblichen UNIX-Referenzsystem zu erreichen. In diesem Sinne wird System V zum Vergleich mit AX als maßgebliches UNIX-Referenzsystem betrachtet.

Der Vergleich mit System V läßt AX eindeutig als UNIX *look-alike* klassifizieren. PAX, als eine bestimmte AX-Variante, ist, bedingt durch die QNX-Emulation, als UNIX *work-alike* zu klassifizieren, da QNX als UNIX *work-alike* klassifiziert werden muß. Diese Klassifikation für PAX ist jedoch fließend, da durch die Integration bestimmter Systemkomponenten von AX die vollständige UNIX-Kompatibilität erreicht werden kann. Diese Kompatibilität ist jedoch *a priori* für PAX nicht gegeben, da mit dieser AX-Variante die QNX-Kompatibilität im Vordergrund steht.

Das Familienkonzept von MOOSE erlaubt es, AX initial mit minimaler Funktionalität zu generieren[48]. Die funktionale Anreicherung von AX richtet sich nach den Systemdiensten, die von zusätzlichen Prozessen in Anspruch genommen werden. Diese Prozesse können der Applikations- aber auch der Systemebene selbst zugeordnet sein. Der *Service-Replugger* von AX sorgt dafür, daß eine Dienstanforderung durch einen mit diesem Dienst assoziierten *Service-Provider* erfüllt werden kann. Die Erzeugung und Integration eines *Service-Providers* kann dann veranlaßt werden, wenn für den betrachteten Dienst noch kein zuständiger Prozeß identifiziert werden konnte. Ein vergleichbarer Ansatz wurde mit MULTICS [Organick 1972] verfolgt, nur ermöglicht dieses System die dynamische Integration zusätzlicher Systemkomponenten auf prozeduraler Ebene.

In UNIX sind viele der durch Systemprozesse in AX realisierten Funktionalitäten üblicherweise fest im Kernel angesiedelt. Die Folge davon ist, daß nur eine vollständige

[48] Die Beschreibung der verschiedenen Phasen der Systeminitialisierung von PAX hat exemplifiziert, welche minimale Funktionalität angesetzt worden ist, um von einem Ein- zu einem Mehr-Benutzersystem überzugehen.

Neugenerierung des UNIX-Kernels zur Integration neuer Systemdienste führen würde. Um den Aufwand der Neugenerierung des Kernels zu umgehen, definiert der UNIX-Kernel oftmals eine Obermenge an statisch zur Verfügung stehender Systemdienste. Einzelne Systemdienste (z.B. *Accounting*) werden dann entsprechend der Zustände bestimmter (prozeßzugeordneter) *Flags* den Prozessen zugängig gemacht. Diese Organisation ist ein wesentlicher Grund für die zum Teil sehr große Komplexität des UNIX-Kernels.

Die grundlegende Dynamik des Familienkonzeptes von MOOSE führt auf funktionaler Ebene zu einer Überlegenheit von AX gegenüber System V –und allgemein zu allen bekannten UNIX *look-/work-alikes*. Die auf Prozeßebene dynamische Rekonfiguration von MOOSE stellt insbesondere den adäquaten Ansatz dar, Funktionalitäten bestimmter UNIX-Varianten nachträglich in ein laufendes System zu integrieren. Typische UNIX-Funktionalitäten sind hierbei z.B.

- *Pipes*
- *Sockets* und *Communication-Server*
- *File-Locking*
- *Message-Passing*
- *Semaphore-Management*
- *Shared-Memory*
- *Naming*
- *Job-Control*
- *Accounting*

Für diese Funktionalitäten ist unter AX der Einsatz entsprechender Systemprozesse vorgesehen, die als UNIX-*Server* bezeichnet werden. Die Notwendigkeit dieser Systemprozesse ergibt sich nur im Zusammenhang mit bestimmten Applikationen, die Dienste der UNIX-*Server* in Anspruch nehmen. Keine zentrale Systemkomponente von AX stützt sich bei der Erbringung ihrer Dienste auf die Funktionalitäten der UNIX-*Server*.

Um AX in UNIX-Umgebungen integrieren zu können, wird eine Emulation der jeweiligen UNIX *System-Call*-Schnittstelle zur Verfügung gestellt. Diese Emulation wird mit MAX realisiert. Die Abbildung der UNIX *System-Calls* kann effizient auf der Kernelebene von MAX durchgeführt werden. Ausschlaggebend hierfür ist die Tatsache, daß die *System-Call*-Schnittstelle von MAX weitestgehend mit der von UNIX Version 7 identisch ist. Damit entspricht die auf der Kernelebene realisierte Dienstabbildung dem Verfahren, wie bereits unter NUKE [Crowley 1981] die UNIX *System-Calls* an enstprechende Systemprozesse verteilt worden sind.

Über *Compatibility-Libraries* können UNIX Systemdienste in einfacher Weise auf entsprechende AX Systemdienste abgebildet werden, ohne den Laufzeitverlust einer durch Prozesse erreichten Emulation in Kauf nehmen zu müssen. Ein UNIX-Prozeß wird dabei unter AX durch einen *custodian* repräsentiert, der allein einem *Team* zugeordnet ist. Die Integration der UNIX-typischen Mechanismen zur Signalbehandlung wird dieses *Team* durch einen oder mehrere *Exception-Clients* erweitern. Diese zusätzlichen *captives* des *Teams* bleiben dem eigentlichen UNIX-Prozeß (der *custodian*) verborgen. Die *captives* werden auf der Ebene der

Compatibility-Library erzeugt und wieder zerstört.

Eine interessante Variante aufgrund des in AX realisierten *Team*-Konzeptes ergibt sich bei der Abbildung des unter 4.2BSD zur Verfügung gestellten *vfork* (siehe auch [Joy et al. 1983]). Die durch die *Compatibility-Library* realisierte Abbildung dieses Systemdienstes nutzt das *Team*-Konzept aus, um den Kopiervorgang des gesamten Datenraumes des *custodian* bei der Erzeugung eines UNIX-Prozesses zu vermeiden. Das *vfork* bedeutet zuerst die Erzeugung eines *captives*, dem erst im Falle des späteren *exec* vollständig ein neuer Datenraum zugeordnet wird. Mit diesem Ansatz wird lediglich der Stack des *custodians* dupliziert. Desweiteren wird die Ausführung des *custodians* solange unterbrochen, bis der *captive* entweder das *exec* ausgeführt hat oder terminiert ist.

6.2.2. V

Das V-System [Cheriton 1984] basiert auf einen *Message-Passing* Kernel, der einheitliche Mechanismen zur Interprozeβkommunikation für lokale- und netzwerkorientierte Systeme zur Verfügung stellt. Die Kommunikationsmechanismen des V-Kernels THOTH [Cheriton 1982] nachempfunden und im wesentlichen zur Unterstützung dezentraler- und verteilter Systeme weiterentwickelt worden.

Die zugrunde liegende Hardware-Architektur des V-Systems besteht aus einem Zusammenschluβ mehrerer Mikroprozessorsysteme, basierend auf der Motorola mc68000-Familie. Jedes Mikroprozessorsystem repräsentiert eine *Workstation* und die Verbindung zwischen den einzelnen *Workstations* erfolgt durch ein lokales Netzwerk (Ethernet) mit einer Bandbreite von 3 bis 10Mb. Die Mechanismen zur netzwerkglobalen Interprozeβkommunikation werden hauptsächlich zum *remote file access* eingesetzt, da nicht jede *Workstation* mit einem lokalen Massenspeichermedium ausgestattet ist. Zur weiteren Beschreibung sei auf [Cheriton, Zwaenepoel 1983] verwiesen, wo die Struktur und die Leistungsfähigkeit des V-Systems dargestellt ist.

Das Entwurfsprinzip des V-Systems ist, wie bei THOTH und AX, prozeβorientiert. Wesentliche Funktionalitäten eines typischen Betriebssystems sind durch *Server*-Prozesse abgekapselt. Die Menge dieser *Server*-Prozesse ist über die an dem lokalen Netzwerk angeschlossenen *Workstations* verteilt. Auf jeder *Workstation* ist die funktional identische Kopie des V-Kernels angesiedelt. Elementare Funktionalitäten zur Prozeβverwaltung (Erzeugung/Zerstörung von Prozessen, Speicherverwaltung) werden duch einen speziellen *Server*-Prozeβ, der *Kernel-Server*, realisiert. Dieser Prozeβ ist funktional dem Kernel zugeordnet und demzufolge ebenfalls auf jeder *Workstation* angesiedelt.

Mit dieser Organisation stellt das V-System ein dezentral organisiertes Betriebssystem dar. Mit Ausnahme des Kernels und des *Kernel-Servers* müssen die restlichen *Server*-Prozesse nicht auf jeder *Workstation* vorhanden sein. In diesen *Server*-Prozessen sind vorwiegend Funktionalitäten zur Gerätekontrolle und zur Dateiverwaltung angesiedelt.

Eine weiterentwickelte Version des V-Systems unterstützt die Migration von Applikationen. Ungenutzte *Workstations* werden dazu eingesetzt, Applikationen zur Ausführung zu bringen und damit jene *Workstations* zu entlasten, die mit der Ausführung von Applikationen

übermäßig beschäftigt sind. Angestrebt ist hierbei eine möglichst gleichmäßige Ausnutzung der Rechenkapazitäten, auch als *Load-Balancing* bezeichnet, der über das Netzwerk miteinander verbundenen *Workstations*. Die Migration von Applikationen kann unter V im Extremfall die Migration mehrerer Prozesse bedeuten. Dies wird dann der Fall sein, wenn eine Gruppe von Prozessen mit der Kontrolle der jeweils zu migrierenden Applikation beauftragt worden ist. Eine weitergehende Beschreibung der Migration von Applikationen unter V ist [Theimer et al. 1985] zu entnehmen.

Obgleich AX wie auch V den prozeßorientierten Systementwurf gemeinsam haben, besteht zwischen beiden Systemen ein bedeutender Unterschied. Dieser Unterschied ist konzeptioneller Natur. Strukturell wird sich ein dezentral organisiertes (oder gar verteiltes) AX-System ähnlich wie V darstellen. Der konzeptionelle Unterschied zwischen AX und V besteht darin, daß unterschiedliche Instanzen die netzwerkglobale Identifikation von und Kommunikation mit den Prozessen des Systems realisieren.

Der V-Kernel stellt netzwerktransparente Interprozeßkommunikationsmechanismen zur Verfügung. Demzufolge besitzen diese Mechanismen des Kernels sowohl *workstation*-lokale- als auch netzwerkglobale Funktionalitäten. Der AX- bzw. MOOSE-Kernel wird mit der Interprozeßkommunikation grundsätzlich nur *workstation*-lokale Funktionalitäten verbinden. Netzwerkglobale Mechanismen zur Interprozeßkommunikation werden, entsprechend des MOOSE-Konzeptes, in AX ausschließlich über entsprechende Systemprozesse zur Verfügung gestellt.

Die Adressierung eines Prozesses, der einer entfernten *Workstation* zugeordnet ist, wird unter AX durch einen *Remote-Server* erfolgen. Der *Remote-Server* wird auf jeder *Workstation* angesiedelt sein und adoptiert alle Prozesse, die nicht auf seiner *Workstation* angesiedelt sind. Die Kommunikation mit einem einer entfernten *Workstation* zugeordneten Prozeß, führt somit immer zu einer lokalen Kommunikation mit dem *Remote-Server*. Dieser Vorgang –die Adoption eines Prozesses– erfolgt transparent für die jeweiligen an der Kommunikation beteiligten Prozesse. Für diese Entwurfsentscheidung sind verschiedene Gründe von Bedeutung, die nachfolgend kurz beschrieben werden:

- Der V-Kernel muß die Entscheidung fällen, ob der bei der Interprozeßkommunikation adressierte Prozeß lokal in derselben *Workstation* oder entfernt in einer anderen *Workstation* angesiedelt ist. Diese Unterscheidung führt in jedem Fall zu einem Laufzeitverlust bei der Interprozeßkommunikation innerhalb derselben *Workstation*. Der AX-/MOOSE-Kernel wird diese Unterscheidung nie fällen müssen und demzufolge bei *workstation*-lokaler Interprozeßkommunikation von seiner Konzeption her laufzeiteffizienter als der V-Kernel sein.
- Das Modell der *Remote-Server* erlaubt es, ein sehr flexibles dezentral organisiertes Betriebssystem aufzubauen. Je nach der Konfiguration können unter AX zur Laufzeit verschiedene *Remote-Server* eingesetzt werden, die jeweils unterschiedliche Netzwerkschnittstellen ansteuern. Der V-Kernel müßte hierfür erst entsprechend konfiguriert werden.
- Eine Realisierung der netzwerkglobalen Interprozeßkommunikation innerhalb des MOOSE-Kernels würde bedeuten, daß der MOOSE-Kernel nicht mehr als Monitor [Hoare

1974] betrachtet werden darf. Bedingt durch ein im Kernel integriertes Kommunikationsprotokoll, würden erhebliche Laufzeiten bei der Interprozeβkommunikation entstehen können. Die Laufzeiten gehen auf Kosten anderer Prozesse, die Dienste des Kernels in Anspruch nehmen wollen und sind demzufolge zu vermeiden. Die Realisierung des MOOSE-Kernels auf Grundlage des Monitor-Konzeptes ist jedoch ein wesentlicher Faktor für die Laufzeiteffizienz der Interprozeβkommunikationsmechanismen. Die Aufgabe dieser Entwurfsentscheidung brächte mit sich, mit jeder Kommunikationsprimitive die explizite Synchronisation beim Zugriff auf PCBs, TCBs und dem *Dispatch-Set* zu fordern. Zusätzlich ist mit dieser Organisation grundsätzlich ein erhöhtes Fehlerrisiko gegeben.

Ob der durch den *Remote-Server* bedingte zusätzliche Laufzeitverlust tatsächlich bedeutend in Erscheinung tritt, wird von der jeweiligen Netzwerkapplikation selbst abhängen und damit applikationsorientiert zu bewerten sein. Unter PAX hat sich gezeigt, daβ der Aufwand des *remote file access*, bei einer *back-to-back* Verbindung der notwendigen Netzwerkprozesse, keinen signifikanten Unterschied zum entsprechenden lokalen Zugriff ergibt. Der *Filesystem-Administrator* selbst benötigt erheblich mehr an Zeit, als der *Remote-Server* benötigt, um den lokalen Zugriff festzustellen und diesen dem lokalen *Filesystem-Administrator* zuzustellen. Der zusätzliche Laufzeitverlust durch den *Remote-Server* bedeutet in diesem Fall ca. einen Faktor von 1.8 bzw. 2.3, wenn der Vergleich zu den Laufzeiten der *Rendezvous* durchgeführt wird. Die Funktionalität des *Remote-Servers*, hinsichtlich der Adoption der nicht auf seiner *Workstation* angesiedelten Prozesse und das Weiterleiten lokaler Dienstanforderungen, entspricht der eines Monitor-Prozesses, wie er in der vorangegangenen Fallstudie skizziert worden ist.

Die in [Cheriton, Zwaenepoel 1983] aufgeführte Begründung für die Integration der netzwerkglobalen Interprozeβkommunikation in den V-Kernel, ist im engen Zusammenhang mit der Architektur des zugrunde liegenden Prozessors bzw., im Falle des V-Systems, der zugrunde liegenden MMU zu sehen. Der Aufwand des *Remote-Servers* unter AX zur Übermittlung einer Nachricht über das Netzwerk wird dem Aufwand des V-Kernels entsprechen. In beiden Fällen wird ein zusätzliches Protokoll notwendig sein, das die netzwerkglobale Kommunikation zwischen Prozessen ermöglicht. Damit ergibt sich durch den *Remote-Server* ein Laufzeitverlust, der im wesentlichen durch den zusätzlichen Prozeβwechsel und das zusätzliche *Rendezvous* bestimmt wird. Für PAX liegt dieser Laufzeitverlust ca. zwischen 1.8 und 1.2 Millisekunden (je nach Größe der mit dem *Rendezvous* verbundenen Nachrichten). Nur wenn die Laufzeiten des Kommunikationsprotokolls – inklusive der Laufzeiten im Netzwerk – in etwa dem durch den *Remote-Server* bedingten Laufzeitverlust entsprechen, wäre die Migration der Funktionalitäten des *Remote-Servers* in den Kernel, aus Gründen der besseren Laufzeiteffizienz, sinnvoll. Ebenso jedoch, wie in [Cheriton, Zwaenepoel 1983] die Forderung nach leistungsfähigerer Hardware zur Unterstützung lokaler Netzwerkarchitekturen gestellt wird, kann die Forderung nach leistungsfähigeren Prozessoren gestellt werden, die den durch das zusätzliche *Rendezvous* bedingten Laufzeitverlust kompensieren. Beispiele solcher Prozessoren sind mit [Tyner 1981] und [Intel 1983] gegeben.

In der gegenwärtigen Entwicklungsphase von AX besitzt die Migration netzwerkglobaler Kommunikationsmechanismen in den Kernel den Stellenwert einer zu früh getroffenen

Entwurfsentscheidung. Alle bisherigen auf PAX zur Ausführung gebrachten Applikationen manifestieren das Entwurfskonzept von MOOSE. Diese Applikationen kontrollieren u.a. X.25 Netzwerkschnittstellen, ohne einen zu QNX feststellbaren Laufzeitverlust. Dies ist umso bemerkenswerter, wenn berücksichtigt wird, daß diese Applikationen unter PAX emuliert werden und nicht mit *Compatibility-Libraries* neu gebunden worden sind. Die Migration solcher Kommunikationsdienste in den Kernel würde für die Applikationen selbst keine Laufzeitverbesserung bedeuten, da die Netzwerkapplikationen unter PAX den "Engpaß" darstellen.

6.2.3. SOS

Im Rahmen des Esprit Projektes der Europäischen Gemeinschaft wird an der Entwicklung eines "intelligenten" Arbeitsplatzsystems gearbeitet, das Büroapplikationen unterstützt und den Zugang zu bzw. die Kommunikantion über ein büroglobales und organisationsübergreifendes Netzwerk ermöglicht[49]. Das Arbeitsplatzsystem wird SOMIW, von *Secure Open Multimedia Integrated Workstation,* genannt und SOS steht stellvertretend für *SOMIW Operating System.* In [Shapiro et al. 1985] sind die Konzepte von SOS beschrieben. Desweiteren ist in dieser Arbeit die Einordnung von SOS in eine UNIX-Umgebung kurz skizziert.

Die Entwicklung von SOMIW beschränkt sich nicht nur auf den software-technischen Bereich, sondern erstreckt sich ebenso über den hardware-technischen Sektor. Jede *Workstation* basiert auf einem Motorola mc68020 Mikroprozessor mit Unterstützung durch eine *Paging*-MMU. Die MMU verwaltet bis zu 4096 *pages* mit jeweils 4K-*Bytes* und ermöglicht damit einen virtuellen Adreßbereich für die einzelnen Prozesse bis zu 16M-*Bytes*.

SOS zeichnet sich durch ein objektorientiertes Entwurfsprinzip aus. Damit unterscheidet es sich grundsätzlich von UNIX, V und AX. Obgleich der objektorientierte Systementwurf als ein sehr moderner Ansatz beim Betriebssystembau zu betrachten ist, sind bereits vor geraumer Zeit erste Systeme vorgestellt worden, bei deren Realisierung ähnliche Prinzipien wie bei SOS zugrunde lagen. In [Isle et al. 1977] sind mit dem DAS-System Ideen verwirklicht worden, wie sie mit SOS ebenfalls angestrebt sind: *Replugging* von Objekten zur Laufzeit eines bestehenden Systems.

Objekte werden unter SOS physikalisch durch eine *page* beschrieben. Es wird damit auf eine herkömmliche Hardware-Architektur zurückgegriffen, die es erlaubt, Objekte gezielt abzukapseln. Dies stellt einen grundlegend anderen Ansatz dar, als er z.B. mit DAS verfolgt worden ist. Mit dem Einsatz herkömmlicher Hardware-Architekturen zur Abkapselung von Objekten stellt sich eine generelle Problematik bei SOS ein. Die Objekte müssen auf *Page-Boundaries* ausgerichtet sein, damit sie innerhalb des virtuellen Adreßraumes eines Prozesses eingeblendet werden können, wenn der betreffende Prozeß auf das Objekt Zugriff erlangen will. Verschnittprobleme werden auftreten, da nicht jedes Objekt eine 4K-*Page* vollständig ausfüllen muß –der objektorientierte Systementwurf motiviert gerade, "kleine" Objekte zugrunde zu legen. Diese Problematik bei der Objektverwaltung von SOS wird schwierig zu

[49] Esprit Projekt Nr. 367.

bewältigen sein.

Nach [Shapiro et al. 1985] stellt die Grundlage der *Paging*-MMU zur Verwaltung von Objekten eine wesentliche Entwurfsentscheidung für SOS dar. Diese Entwurfsentscheidung ist jedoch nicht als positiv zu bewerten. Vielmehr sollte bei der Objektverwaltung von einer Konzeption ausgegangen werden, die es ermöglicht, Objekte variabler Größen modellieren zu können. Die Zuordnung von *Pages* für einzelne Objekte wäre in diesem Fall eine mögliche technische Realisierung der Abkapselung von Objekten[50].

Ungeachtet der Tatsache, daß die Hardware-Architektur von SOMIW keine optimale Unterstützung für SOS darstellt, so ist mit SOS doch ein wesentlicher innovativer Aspekt zu sehen. Dieser Aspekt führt zu einer sehr konsequenten Sichtweise und Realisierung des Dienstbegriffs eines Applikations-/Betriebssystems. Die Grundlage stellt hierzu das in [Shapiro et al. 1985] vorgestellte *proxy principle*. Das *proxy principle* legt fest, wie auf Objekte zugegriffen werden kann, um damit Dienste durch Objekte zugänglich zu machen. Das grundsätzliche Verfahren des *proxy principles* bedeutet, daß

a) der Zugriff auf ein Objekt nur dann möglich ist, wenn der betreffende Prozeß ein *proxy* besitzt, das den Zugang zu dem eigentlichen Objekt regelt;

b) das *proxy* explizit durch den betreffenden Prozeß zu importieren ist und in Beziehung zu dem sogenannten *principal* des Objektes gebracht werden muß.

Das *principal* repräsentiert das jeweils betrachtete Objekt. Ein *proxy* entspricht einer *capability* auf ein Objekt (das *principal*), wobei diese *capability* lokal für jeden besitzenden Prozeß unterschiedlich definiert sein kann. Was jedoch über den Begriff der *capability* hinausgeht, ist die Tatsache, daß ein *proxy* den Zugang bzw. ein Kommunikationsprotokoll zu seinem *principal* festlegt. Wie auf das eigentliche Objekt somit zugegriffen wird und wo dieses Objekt angesiedelt ist, bleibt dem ein *proxy* besitzenden Prozeß vollkommen verborgen.

Die durch das *proxy principle* erreichte Abstraktion von dem Zugang zu einem Objekt und der Implementierung eines Objektes, ist für SOMIW von zentraler Bedeutung. Diese Abstraktion erlaubt es, Applikationen zu realisieren, die unabhängig von ihrer Zuordnung zu bestimmten *Workstations*, Büros oder Organisationen sind. Die Beziehung zwischen einem *proxy* und seinem *principal* kann

- prozeduraler Natur sein;
- auf der Kommunikation zwischen Prozessen derselben *Workstation* basieren;
- sich Dienste netzwerkglobaler Kommunikationsmechanismen zu Nutze machen.

Das *proxy* entspricht damit dem in [Schindler 1980] skizzierten Modell eines *servive access points*.

---

[50] Wenn mit SOMIW bereits eine eigene und zudem noch kostspielige Hardware-Entwicklung verbunden ist, stellt sich die Frage nach den Entscheidungsrichtlinien, die letztendlich dazu führen, SOS als objektorientiertes Betriebssystem auf herkömmliche Hardware-Architekturen aufzusetzen. Der Intel i80286 wäre zur Unterstützung eines objektorientierten Systems besser geeignet, als der mit SOMIW zum Einsatz kommende mc68020. Mit dem i80286 kann direkt ein *capability based system* aufgebaut werden, wie es eigentlich aufgrund der Konzeption für SOS zugrunde gelegt werden müßte.

Das MOOSE-Konzept sieht ähnliche Mechanismen vor, die die Beziehung zwischen einem dienstanfordernden- und diensterbringenden Prozeß, unabhängig von der Lokalisation der beiden betreffenden Prozesse, herstellen. In AX sind diese Konzepte von MOOSE in Form des *Service-Administrators* realisiert. Die Benennung eines Dienstes stellt in AX wie auch in SOS den Schlüssel dar, den Zugang zu Dienstschnittstellen abstrakt formulieren und realisieren zu können. In diesem Sinne sind beide Systeme direkt miteinander vergleichbar, nur basiert AX nicht auf der Unterstützung objektorientierter Systemarchitekturen.

Die dynamische Rekonfiguration von SOS basiert darauf, ein *principal* zur Laufzeit auszutauschen bzw. überhaupt erst verfügbar zu machen. Da ein *proxy* immer die Abstraktion von seinem *principal* ermöglicht, erfolgt der Austausch eines *principals* transparent für alle Prozesse, die im Besitz eines entsprechenden *proxy's* sind. Die Migration von Diensten wird damit in eleganter Weise unterstützt.

Die Beziehung zwischen einem *proxy* und seinem *principal* kann ihre Ausprägung auf rein virtueller Ebene besitzen. Die Benutzung eines *proxy's*, mit dem real noch kein *principal* assoziiert ist, führt zu einem Trap. Das entsprechende *principal* kann in dieser Weise nachträglich eingebunden und die virtuelle Beziehung zwischen *proxy* und *principal* in eine reale umgewandelt werden. Dieser Vorgang entspricht dem des dynamischen Bindens unter MULTICS [Organick 1972], dort durch Anwendung der *trap-on-use* Eigenschaft des zugrunde liegenden Prozessors unterstützt.

Vergleichend mit AX, können unter SOS mehrere Prozessen demselben Adreβraum zugeordnet werden. Diese Zuordnung ist, im Gegensatz zu dem in AX verwirklichten *Team*-Konzept, dynamisch für die Prozesse. Der jeweils aktuelle Adreβraum eines Prozesses wird unter SOS als *Context* bezeichnet. Die Kommunikation zwischen Prozessen, die demselben *Context* zugeordnet sind, erfolgt direkt, d.h. durch Ausnutzung des *Shared-Memory*. Die Kommunikation zwischen Prozessen die jeweils einem unterschiedlichen *Context* zugeordnet sind, ist nur über ein entsprechendes *proxy* möglich. Ist die Ausprägung des *proxy's* prozeduraler Natur, so wird der Prozeβ bei Benutzung des *proxy's* seinen *Context* wechseln können. Dies entspricht dem Domänenwechsel unter DAS [Mueller et al. 1980]. Mit jedem Objekt (*principal*) kann unter SOS ein eigener *Context* assoziiert werden, womit das betreffende Objekt vollständig abgekapselt wird.

6.3. Bewertung

Die Darstellung von UNIX, V und SOS in den vorangegangenen Abschnitten sollte bewuβt keinen vollständigen Charakter besitzen. Ein vollständige Darstellung hätte den Rahmen dieser Arbeit sicherlich gesprengt. Für eine genaue Beschreibung dieser Systeme sei daher auf die einschlägige Literatur verwiesen. Die Skizzierung der Funktionalitäten von UNIX, V und SOS erlaubt jedoch die Bewertung von AX, indem ein Vergleich ermöglicht wird, der die Einordnung von AX in den gegenwärtigen "Stand der Kunst" auf dem Betriebssystembereich gestattet.

Obgleich UNIX seit geraumer Zeit im Betriebssystembereich eine dominante und erfolgreiche Stellung einnimmt, reflektiert dieses System – als Repräsentanten seien System V und 4.2BSD genannt– nicht notwendigerweise das technisch/wissenschaftlich Mögliche in

diesem Bereich. Nur wenige UNIX-Varianten, wie z.B. die *Newcastle Connection* [Brownbridge et al. 1982] und LOCUS [Walker et al. 1983], die jedoch industriell und kommerziell keinen vergleichbaren Erfolg verbuchen können, sind in diesem Zusammenhang als vorbildlich hervorzuheben. Das V-System und SOS repräsentieren zwei Systeme, die gegenüber UNIX dem gegenwärtigen "Stand der Kunst" im Betriebssystembereich verkörpern.

AX kann allgemein dahingehend bewertet werden, daß es mit UNIX, V und SOS jeweils bestimmte Funktionalitäten gemeinsam hat. Der Bezug zu UNIX ergibt sich eindeutig durch die kompatible Dienstschnittstelle von AX; PAX stellt ein UNIX *work-alike* dar, und MAX ist als UNIX *look-alike* zu betrachten. Mit V sind der prozeßorientierte Systementwurf und die auf *Message-Passing* basierenden Mechanismen zur Interprozeßkommunikation gemeinsam. Zu SOS sind in AX vergleichbare Prinzipien des Dienstbegriffs realisiert und Konzepte zur dynamischen Rekonfiguration eines Systems zugrunde gelegt.

6.3.1. Laufzeiteffizienz

Der Vergleich zu QNX hat gezeigt, daß der MOOSE-Kernel, der AX zugrunde liegt, in allen Punkten laufzeiteffizienter ist, als der QNX-Kernel. Der Erfolg von QNX im Personal-Computer-Bereich ist u.a. gerade durch die Laufzeiteffizienz der in diesem Betriebssystem eingesetzten Mechanismen zur Interprozeßkommunikation begründet. Die für den MOOSE-Kernel durchgeführten *Benchmarks* weisen eine 10- bis 11-prozentige Laufzeitverbesserung der Interprozeßkommunikation zu QNX auf. Noch gravierender ist der Unterschied bei explizit initiierten Prozeßwechseln. Der MOOSE-Kernel verzeichnet hierbei ein zu QNX um 51 Prozent besseres Laufzeitverhalten. Ein durch einen *Clock-Interrupt* verursachter *Team*-Wechsel, ohne daß der *Clock-Interrupt* zu einem *Server*-Prozeß propagiert werden muß, beläuft sich bei PAX auf ca. 100 Mikrosekunden.

Diese Fakten untermauern den prozeßorientierten Betriebssystementwurf, bei dem die einzelnen Systemprozesse durch *Message-Passing* miteinander kommunizieren. Die im Vergleich zu prozedurorientierten Systemen (z.B. UNIX) verstärkt auftretende Systembelastung durch zusätzliche Prozeßwechsel, ist mit den Mechanismen des MOOSE-Kernels sehr niedrig gehalten worden. Die initiale- und terminale Phase eines *System-Calls* unter PAX wird sich mit einer Laufzeit von ca. 1.2 Millisekunden niederschlagen. Dieser Laufzeit-*Overhead* wird für alle *System-Calls* unter PAX etwa gleich bleiben. Die Laufzeiten des Prozeßwechels und der Nachrichtenübermittlung geben eine bestimmte obere Grenze an, bis zu der die Anzahl der Argumente von *System-Calls* den Laufzeit-*Overhead* der initialen- und terminalen Phase nicht beeinflussen.

6.3.2. Flexibilität

Der mit dem MOOSE-Konzept zugrunde gelegte prozeßorientierte Betriebssystementwurf wurde erfolgreich zur Realisierung von AX durchgeführt. Es haben sich hierbei die Ergebnisse von [Cheriton 1982] bestätigt, welche Prozesse zur Strukturierung eines Betriebssystems motivieren. In AX sind Prozesse konsequent zur Realisierung aller Systemprozesse eingesetzt worden. Hierbei hat das *Team*-Konzept von MOOSE die zentrale Position eingenommen.

Die Flexibilität von AX wird im wesentlichen durch Mechanismen des MOOSE-Kernels zur Propagation von Traps/Interrupts und zur Adoption von Prozessen unterstützt. Drei Modelle zur Annahme und Behandlung propagierter Interrupts stehen für *Server*-Prozesse zur Verfügung. Mit jedem Modell können Systeme zur Interrupt-Behandlung konstruiert werden, die jeweils bestimmten hardware-technischen Randbedingungen genügen müssen.

Das *Gate-Server*-Modell ermöglicht die vollständige Abkapselung der *Interrupt-Handler*. Die notwendigen Mechanismen zur Aktivierung eines *Gate-Servers*, aufgrund eines Interrupts, können z.B. vollständig durch entsprechende Funktionalitäten des i80286 unterstützt und damit auf moderner Hardware effizient realisiert werden. Die betriebssystemtechnischen Voraussetzungen, um solche Hardware beim Entwurf und der Realisierung eines Betriebssystems nutzen zu können, sind in AX gegeben.

Modul-Gates ermöglichen zwar nicht die vollständige Abkapselung eines *Interrupt-Handlers* durch einen eigenen Prozeβ, jedoch wird dem *Interrupt-Handler* immer der vollständige Adreβraum des *Teams* seines *Interrupt-Servers* zum Zeitpunkt des Interrupts eingeblendet. Dieses Modell ist für bestimmte MMU-organisierte Systeme geeignet, bei denen jeweils eine geringe Anzahl von MMU-Segmenten zur Adreβraumdefinition eines *Teams* verwendet wird. Damit wird die Laufzeit zur Einblendung des Adreβraumes für den *Interrupt-Handler* minimal gehalten.

Beide Modelle, *Task-* und *Modul-Gates*, ermöglichen es, daβ *Interrupt-Handler* in speziellen *Teams* angesiedelt werden und dort operieren können. Da *Teams* dynamisch erzeugt und wieder zerstört werden können, ist die dynamische Rekonfiguration eines Systems zur Interrupt-Behandlung unter AX zur Laufzeit möglich. Vergleichbare Verfahren sind unter UNIX und V nicht gegeben und in [Shapiro et al. 1985] sind für SOS ebenfalls keine vergleichbaren Techniken konkret beschrieben.

Das Modell der *Interrupt-Gates* wird zur Realisierung von MAX zugrunde gelegt. Für PAX ist erfolgreich das Modell der *Modul-Gates* (zur zeichenorientierten Ein-/Ausgabe) und das Modell der *Task-Gates* (zur blockorientierten Ein-/Ausgabe) eingesetzt worden. *Interrupt-Gates* ermöglichen die unter AX effizienteste Realisierung der Aktivierung der *Interrupt-Handler,* zumindest wenn von speziellen Hardware-Mechanismen zur Unterstützung von *Gate-Server* abgesehen wird. Zur Aktivierung der *Interrupt-Handler* wird durch den jeweiligen *Gate-Handler* des Kernels für die *Interrupt-Gates* weder ein Adreβraum eingeblendet noch ein spezieller Prozeβ (*Gate-Server*) gestartet. Es besteht damit eine direkte Beziehung zwischen dem Interrupt-Vektor und dem *Interrupt-Handler*. Die Konsequenz davon ist jedoch, daβ unter MAX die *Interrupt-Handler* im *Kernelmode* ausgeführt werden. Die dynamische Rekonfiguration eines Interrupt-Behandlungssystems ist damit nicht mit der Flexibilität verbunden, wie sie auf Grundlage der *Modul-* und *Task-Gates* gegeben ist. Der Austausch der *Interrupt-Handler* ist unter MAX nicht möglich, jedoch kann der *Interrupt-Server* und sein *Team* zur Laufzeit ausgewechselt werden. Die Datenkommunikation zwischen *Interrupt-Server* und *Interrupt-Handler* erfolgt durch explizit angefordertes *Shared-Memory*.

Die Behandlung vom MOOSE-Kernel propagierter Traps ist im Vergleich zur Behandlung von Interrupts sehr einfach. Aufgrund des Asynchronismus von Interrupts müssen zur Interrupt-Propagation explizit Synchronisationsmaβnahmen durch den MOOSE-Kernels

durchgeführt werden. Diese Maβnahmen werden zur Trap-Propagation nicht durchgeführt. Die Propagation eines Traps wird vom MOOSE-Kernel durch die (implizite) Interprozeβkommunikation des den Trap auslösenden Prozesses mit dem *Trap-Server* erfolgen. Hierfür werden die gängigen Mechanismen zur Interprozeβkommunikation des MOOSE-Kernels zugrunde gelegt. In diesem Sinne ist der einen Trap verursachende Prozeβ der *Rendezvous-Client* und der den Trap behandelnde Prozeβ der *Rendezvous-Server*.

Die Mechanismen zur Trap-Propagation durch den MOOSE-Kernel werden in AX zur dynamischen Rekonfiguration von Emulationssystemen eingesetzt. Das QNX-Emulationssystem unter PAX nutzt diese Mechanismen u.a., um *Shared-Libraries* dynamisch verwalten zu können. Der erste Trap in die *Shared-Library* wird von einem *Trap-Server* abgefangen, der daraufhin die *Shared-Library* in den Speicher lädt und die notwendige Verbindung zwischen Trap-Vektor und Einstiegspunkt der *Shared-Library* herstellt. Dieses Verfahren ist eine sehr einfache Form des dynamischen Bindens eines Systems, wie es vergleichsweise unter MULTICS [Organick 1972] stattfindet.

Die Propagation von Traps/Interrupts durch *Server-Gates* entkoppelt die Trap-/Interrupt-Behandlung vollständig vom MOOSE-Kernel. Die *Server*-Prozesse zur Behandlung der Traps/Interrupts können unter AX dynamisch rekonfiguriert werden. Ein einfaches Protokoll zum Austausch (*Replugging*) von *Server*-Prozessen führt dazu, daβ

a) die *Server*-Prozesse die Trap-/Interrupt-Behandlung einstellen und die von den *Server*-Prozessen benutzten *Server-Gates* freigegeben werden;

b) der Austausch der betreffenden *Server*-Prozesse auf die Zerstörung alter und die Erzeugung neuer *Server*-Prozesse zurückgeführt wird;

c) die Behandlung der Traps/Interupts durch den neuen *Server*-Prozeβ oder einer Gruppe von *Server*-Prozessen erneut aufgenommen wird, indem sich die betreffenden *Server*-Prozesse an die entsprechenden Trap-/Interrupt-Vektoren des realen Prozessors anbinden.

Dieses einfache Protokoll (*Replug-Protocol*) wird mit den betreffenden *Server*-Prozessen selbst abgehandelt und durch einen bestimmten Systemprozeβ kontrolliert. Die eigentliche Problematik der Rekonfiguration eines Systems zur Trap-/Interrupt-Behandlung – z.B. das "Einfrieren" jeglicher mit den Geräten verbundenen Ein-/Ausgabeaktivitäten – bleibt lokal im jeweiligen *Server*-Prozeβ angesiedelt.

Die Adoption von Prozessen ist die andere wesentliche Funktionalität des MOOSE-Kernels, die die Flexibilität von AX prägt. Der *Deblocker* ist ein sehr einfaches Beispiel, das angibt, wie die dynamische Rekonfiguration eines Systems aufgrund der Adoption eines *Rendezvous-Servers* erreicht werden kann. Die Integration eines komplexeren Systems unter AX ist durch die Aufnahme des *Remote Filesystem-Administrators* gegeben. Der lokale *Filesystem-Administrator* wird hierbei von dem *Remote Filesystem-Administrator* adoptiert, um lokale/entfernte Operationen unterscheiden und entsprechend initiieren zu können. Für den *Filesystem-Administrator* im speziellen und für das gesamte bestehende AX-System im allgemeinen, ist der *Remote Filesystem-Administrator* nicht sichtbar (transparent).

6.3.3. Portabilität

Die Realisierung von AX erstreckte sich über einen Zeitraum von ca. 2 Jahren. Dieser relativ große Zeitraum ist dadurch begründet, daß nacheinander auf drei verschiedenen Prozessoren die Realisierung von AX durchgeführt worden ist.

Die erste Version von AX basierte auf einer LSI 11. Dieser Prozessor wurde durch eine 8 segmentige MMU unterstützt, die jeweils einen vollständigen Satz an Segmentregister für den *User-* und *Kernelmode* zur Verfügung gestellt hat. Die Komplexität von AX für die LSI 11 umfaßte

- einen Kernel, dessen Mechanismen zur Interprozeßkommunikation vollständig auf *Ports* basierten und immer eine Verbindung (*Connection*) zwischen zwei *Ports* voraussetzten, bevor Prozesse miteinander kommunizieren konnten;
- einen *Team-Administrator*, der die Erzeugung/Zerstörung von Prozessen und *Teams* verwaltete und allgemein für die Speicherverwaltung des Systems zuständig war;
- ein Ein-/Ausgabesystem, das durch jeweils ein *Team* zur Verwaltung von block-/zeichenorientierten Geräten aufgebaut gewesen ist;
- einen *Filesystem-Administrator*, der UNIX Version 7 kompatibel war;
- einen UNIX-Emulator, der einen großen Teil der UNIX Version 7 *System-Calls* auf Systemdienste der zu diesem Zeitpunkt gültigen Systemdienste von AX abbildete.

Die nächste Entwicklunsphase von AX bestand in der Portierung des Systems auf einen Personal-Computer, der auf dem Intel i8086 basierte. Diese Portierung hat insbesondere zu einer vollständigen Überarbeitung des *Team-Administrators* geführt, da es sich gezeigt hat, daß das *Team*-Konzept nicht in der bestehenden Form auf einem i8086 realisiert werden konnte. Eine andere vollständige Überarbeitung ergab sich im Zusammenhang mit dem Kernel. Das verbindungsorientierte Modell [Pause 1985] wurde vollständig fallen gelassen und *Ports* wurden nur noch für die Übermittlung von Signalen, aber nicht mehr von Nachrichten, eingesetzt. Die Begründung hierfür lag ausschließlich in der Steigerung der Laufzeiteffizienz der Mechanismen zur Interprozeßkommunikation. Der auf dem verbindungsorientierten Modell basierende Kernel war, nach erfolgreicher Portierung des Kernels auf den Personal-Computer, ca. um den Faktor 2 langsamer als der QNX-Kernel. Das Ein-/Ausgabesystem konnte in vielen seiner Verwaltungskomponenten übernommen werden. Dennoch waren auch hier Änderungsmaßnahmen angesetzt, die die Adaption an neue Gerätekonfigurationen bedeuteten und die Optimierung des Ein-/Ausgabsystems zur Aufgabe hatten.

Die Überarbeitung des Kernels, *Team-Administrators* und Ein-/Ausgabesystems führte zu Systemkomponenten, die eine neue Basis für PAX definierten. Um bestehende, auf dem Personal-Computer ablaufende Applikationen auch unter PAX nutzen zu können, wurde die vollständige Emulation von QNX durchgeführt.

Die dritte Entwicklungsphase von AX wurde mit der Portierung von PAX auf einen Motorola mc68000, unterstützt durch eine mc68451 MMU, angegangen. Die Portierung des Kernels und des *Filesystem-Administrators* sowie die Adaption des Ein-/Ausgabesystems an neue Gerätekonfigurationen erfolgte beim ersten Versuch innerhalb von zwei Monaten. Die größte Problematik bei der Portierung des Kernels bestand in der Verwaltung der mc68451

MMU, um *Message-Passing* zwischen beliebigen Adreβräumen und Prozeβ-/*Team*-Wechsel realisieren zu können. Weitere Portierungsvorgänge fanden statt, immer dann, wenn signifikante Systemänderungen von AX auf dem Personal-Computer durchgeführt worden sind.

Neben der reinen Portierung von AX, vom i8086 auf einen mc68000, wurde eine dritte Redesignphase für den *Team-Administrator* angesetzt. Die Mechanismen zur Speicherverwaltung und zur Adreβraumzuordnung für Prozesse/*Teams* wurden durch einen eigenen Systemprozeβ abgekapselt. Dieser Systemprozeβ ist der *Context-Administrator*. Mit dieser Maβnahme ist der *Team-Administrator* von AX vollkommen portabel, zumindest wenn Prozessoren wie die LSI 11, der i8086/88 und der mc68000 betrachtet werden.

Dadurch, daβ frühzeitig verschiedene Prozessoren bei der Realisierung von AX betrachtet worden sind, konnte entwurfsbedingten Portierungsschwierigkeiten des Systems rechtzeitig begegnet werden. Die Entwicklung und die Portierung von AX nahmen von Anfang an zwei gleichberechtigte Stellungen in den einzelnen Entwurfsphasen ein. Der Portierungsaufwand von AX ist nach den bisherigen Erfahrungen im wesentlichen von zwei Faktoren abhängig:

a) die Verwaltung einer MMU oder vergleichbarer Hardware zur Adreβraumzuordnung. Die entsprechenden Verwaltungsmaβnahmen sind ausschlieβlich vereinzelt im Kernel und vollständig im *Context-Administrator* angesiedelt. Der Kernel nimmt hierbei eher eine passive Stellung ein, da er lediglich die durch den *Context-Administrator* definierten *Image-* und *Context-Descriptoren* liest, um, entsprechend der in diesen Strukturen abgelegten Informationen, die Hardware zur Adreβraumverwaltung des Prozessors programmieren zu können.

b) die Adaption an neue Gerätekonfigurationen. Die Gerätetreiber sind unter AX durch Systemprozesse gezielt abgekapselt. Die Rekonfiguration des Kernels wird nur dann notwendig sein, wenn die Interrupt-Behandlung aus laufzeittechnischen Gründen in den Kernel migriert werden muβ.

Die Implementierung von AX ist fast vollständig in C [Kernighan, Ritchie 1978] erfolgt. Nur der Kernel weist verschiedentlich Assemblersequenzen auf. Diese Assemblersequenzen repräsentieren die unterste Ebene der *Gate-Handler* des MOOSE-Kernels und dienen damit im wesentlichen der programmiersprachlichen Anbindung von C an die Trap-/Interruptschnittstelle des realen Prozessors.

Kapitel 7

Ausblick

Die vorliegende Arbeit wäre unvollständig, wenn nicht kurz der weitere Verlauf von MOOSE skizziert werden würde. Deshalb soll in diesem Kapitel dargestellt werden, welche zukünftigen Aktivitäten mit MOOSE verbunden sind.

Gegenwärtig kann auf ein voll einsatzfähiges Mitglied der MOOSE-Betriebssystemfamilie, AX, zurückgegriffen werden. Mit AX können erste konkrete Ergebnisse erzielt werden, die helfen, die Eignung verschiedener MOOSE-Konzepte nicht nur im experimentellen Bereich zu bewerten. Der bisherige Erfahrungsschatz über AX, sowie über die Realisierung bestimmter Konzepte von MOOSE, wird durch den beabsichtigten Einsatz im Lehrbetrieb [51] weiter ausgebaut werden können. Desweiteren wird der Einsatz von PAX unter konkreten industriellen/kommerziellen Gesichtspunkten in Erwägung gezogen, so daß zu erwarten ist, auch von dieser Seite auf eine Fülle von Erkenntnissen zurückgreifen zu können. Beide Einsatzbereiche lassen den Schluß zu, die technische Realisierung von AX sowie die Konzepte von MOOSE vervollkommnen zu können.

Neben diesen kurz skizzierten Zukunftsperspektive zeichnen sich jedoch jetzt bereits bestimmte Erkenntnisse ab, die durch das bisherigen Arbeiten mit AX gewonnen werden konnten. Diese Erkenntnisse beziehen sich ausschließlich auf die Mechanismen zur Interprozeßkommunikation. Drei Aspekte lassen sich hierbei angeben, die im wesentlichen zur Steigerung der Laufzeiteffizienz der Interprozeßkommunikation führen. Daß diese Aspekte bei der gegenwärtigen Realisierung von AX technisch nicht berücksichtigt worden sind, hat seine Ursache darin, insbesondere PAX konkret in eine QNX-Umgebung leicht integrieren zu können. Mit dem zunehmenden Austausch der QNX-Umgebung durch eine eigene Umgebung für AX, sollte jedoch die Überarbeitung der Mechanismen zur Interprozeßkommunikation von AX erfolgen.

7.1. Entkopplung des Empfangs von Signalen und Nachrichten

Der gegenwärtige Kernel gestattet es einem Prozeß, mittels *receive* zugleich auf Signale und Nachrichten warten zu können. Konsequent und mehr den Entwurfsprinzipien von MOOSE naheliegend wäre es, das *Team*-Konzept entsprechend zu nutzen und z.B. jeweils einem Prozeß den Empfang von Signalen sowie einem anderen Prozeß den Empfang von Nachrichten aufzubürden. Die gegenwärtige Implementierung von *receive* ließe sich dadurch erheblich vereinfachen, da nicht der Sonderfall eines empfangenden Signals betrachtet werden muß und

[51] Im Rahmen von projektorientierten Lehrveranstaltungen am Fachbereich 20 (Informatik) der Technischen Universität Berlin.

demzufolge auch kein Datentransfer zu erfolgen hat.

Die Erfahrungen mit dem PAX zugrunde liegenden Prozessor (i8086) zeigen, daß die Behandlung des Sonderfalls eines empfangenden Signals, die Leistungsfähigkeit der Mechanismen zur Interprozeßkommunikation insgesamt um ca. 2 % herabsetzen. Ein *Rendezvous* kostet ca. 1.2 Millisekunden und die Sonderbehandlung für den Empfang eines Signals durch *receive* kostet ca. 15-20 Mikrosekunden. Vergleichbare Aussagen lassen sich auch auf andere Prozessoren anwenden.

Ebenso lassen sich auf der Seite des signalisierenden Prozesses Laufzeitverbesserungen erzielen. Wenn ein Prozeß zu einem Zeitpunkt nur jeweils für den Empfang von Nachrichten oder für den Empfang von Signalen bereit ist, so braucht auf der signalisierenden Seite nur noch ein Zustand überprüft zu werden. Dies wird insbesondere bei der Propagation von Interrupts, d.h. bei der Übermittlung eines *Interrupt-Signals*, von Bedeutung sein.

Jedoch nicht nur die Laufzeitverbesserung wäre die Begründung für die Entkopplung des Empfangs von Signalen und Nachrichten. Die Realisierung der Mechanismen zur Interprozeßkommunikation des Kernels wird erheblich vereinfacht und die Übersichtlichkeit bzw. Wartbarkeit des Kernels selbst steigt dadurch. Dies ist ein wesentlicher Aspekt zum Entwurf und zur Realisierung eines *Security Kernels*.

7.2. Migration des Specific-Receive aus dem Kernel

Das *specific receive* des Kernls gestattet es, gezielt auf die Übermittlung der Nachricht von einem bestimmten Prozeß warten zu können. Bei der Realisierung von AX ergaben sich jedoch sehr wenige Situationen, wo der Einsatz eines *specific receive* angebracht war. Der Normalfall besteht in den üblichen *Client/Server* Beziehungen, die mit dem *nonspecific receive* modelliert worden sind.

Die sehr seltene Anwendung des *specific receive* rechtfertigt nicht, eine entsprechende Funktionalität von einem Kernel zur Verfügung zu stellen, zumal ein *specific receive* ebenso in eleganter Weise auf der *Library*-Ebene realisiert werden kann. Desweiteren würde durch den Wegfall des *specific receive* allgemein für den Kernel eine ähnliche Leistungssteigerung erzielt werden können wie sie im vorigen Abschnitt skizziert worden ist. Diese Leistungssteigerung Gute kommen.

Desweiteren ist eine Verbesserung der Laufzeit auf der Seite des sendenden Prozesses zu verzeichnen, wenn das *specific receive* nicht im Kernel integriert ist. Die Bereitschaft für ein spezifisches *Rendezvous* eines *Servers* mit dem beim *specific receive* identifizierten *Client* müßte nicht mehr gesondert überprüft werden. Genau nur ein spezifischer *Client* würde in diesem Falle das *Rendezvous* mit einem *Server* starten können und nicht der erste *Client*, der zum *Server* eine Nachricht übermittelt. Die Überprüfung dieses Sonderfalls würde nicht mehr notwendig sein, wenn nur noch ein *nonspecific receive* betrachtet zu werden braucht.

7.3. Kurze Nachrichten fester Länge

Nach der Portierung von AX von einem i8086 auf einem mc68000, der durch eine MMU vom Typ mc68451 unterstützt wird, und der Durchführung der ersten *Benchmarks* für MAX zeigte es sich, daß die Laufzeiten eines *Rendezvous* um ca. den Faktor 3 bis 4 höher waren als die von PAX. Dies ist um so mehr bemerkenswert, als der mc68000 von seiner Prozessorleistung allgemein höher einzustufen ist als der i8086. Den einzigen Ausschlag für diesen enormen Laufzeitverlust bei der Interprozeßkommunikation ergab die Tatsache, daß

a) Nachrichten, wie bei QNX, immer direkt zwischen den Adreßräumen der Prozesse ausgetauscht wurden und

b) die mc68451 sehr umständlich zu programmieren war und damit den wesentlichen Engpaß darstellte.

Würde für den Datentransfer bei der Interprozeßkommunikation keine MMU-Programmierung mehr stattfinden müssen, so könnten die Laufzeiten für die initiale und die terminale Phase eines *System-Calls* im speziellen bzw. eines *Rendezvous* im allgemeinen Fall erheblich verbessert werden.

Der Lösungsansatz ergibt sich direkt, wenn betrachtet wird, daß bei allen bekannten Hardware-Architekturen vom *Kernelmode* aus immer der direkte Zugriff auf den jeweiligen Benutzeradreßraum besteht. Zu dem Zeitpunkt eines *send* oder *receive* wäre somit immer ein Adreßraum bereits eingeblendet, der den Ursprung oder das Ziel einer Nachricht darstellt. Der Kopiervorgang einer Nachricht über den Kerneladreßraum wäre mit diesem Hintergrund immer laufzeiteffizienter als der direkte Kopiervorgang zwischen den Benutzeradreßräumen (hierzu muß immer vom Kernel ein Adreßraum explizit eingeblendet und damit die MMU programmiert werden).

Dieser enorme Laufzeitverlust bei der Interprozeßkommunikation auf der Grundlage von *Message-Passing* in MMU-organisierten Systemen ist sicherlich auch ein Grund dafür gewesen, in THOTH [Cheriton 1979] den "scheinbaren" Umweg über den Kerneladreßraum in Kauf zu nehmen. Wird jedoch über den Kerneladreßraum kopiert, so ergibt sich ein generelles Pufferungsproblem für den Kernel, insbesondere dann, wenn die zu kopierenden Nachrichten unterschiedlich groß sind. Der Laufzeitgewinn würde sich nicht auszahlen, wenn der Kernel mit unterschiedlich großen Nachrichten umgehen und demzufolge eine entsprechende Speicherverwaltung betreiben müßte. Wären die Nachrichten alle gleich groß und minimal in ihrer Größe, so könnte jedem Prozeß ein kernelinterner Bereich statisch zugeordnet werden, in dem die jeweils zu sendende/empfangende Nachricht plaziert werden kann. Es ist hierbei jedoch zu beachten, daß auch für solch eine Lösung die Mechanismen des MOOSE-Kernels zur Interprozeßkommunikation immer blockierend wirken würden.

Erste Skizzierungen der entsprechend abgewandelten Mechanismen zur Interprozeßkommunikation zeigen, daß sich dieser Ansatz nicht nur für komplexe Hardware-Architekturen eignet. Bis zu einer Größe von ca. 64 *Bytes* für eine statische Nachricht, wären auch für PAX (basierend auf einem i8086 ohne MMU) Laufzeitverbesserungen, zumindestens keine Laufzeitverluste, zu verzeichnen.

Den eigentlichen Nachteil mit diesem Ansatz scheint die feste Länge einer Nachricht darzustellen. Werden jedoch z.B. die einzelnen Nachrichten zur Formulierung der *System-*

Calls unter AX untersucht, so ist festzustellen, daß eine Nachrichtengröße von 32 *Bytes* ausreicht, um die *System-Calls* und die entsprechenden Argumente zu den jeweiligen Systemprozessen zu übermitteln. Das Prinzip der *System-Call*-Schnittstelle unter AX besteht darin, nicht unbedingt mehr an Informationen in eine Nachricht zu kodieren als die Argumentenliste bei einer prozeduralen Sichtweise des jeweiligen *System-Calls* vorgibt. Größere Datenbestände werden nicht durch *Message-Passing* sondern durch *transfer* zwischen den Prozessen ausgetauscht.

Die Zwischenpufferung einer Nachricht fester Länge im Kernel hätte noch weitere Vorteile. Zum einen wäre die Integrität einer bereits übermittelten Nachricht immer garantiert. Das *Team*-Konzept ermöglicht es generell anderen Prozessen eines *Teams*, eine bereits übermittelte Nachricht nachträglich zu manipulieren. Mit dem Kopieren einer Nachricht in den Kerneladreßraum wird die Möglichkeit zur Manipulation einer zur Übermittlung bereitgestellten Nachricht ausgeschlossen. Zum anderen läßt sich die Aus-/Einlagerung von *Teams* mit diesem Ansatz vereinfachen. Unter PAX besteht das spezielle Problem, eine *Reply-Message* an einen Prozeß, der einem ausgelagerten *Team* zugeordnet ist, zurückzugeben. Logisch gesehen ist durch das *Rendezvous*-Konzept zwar der betreffende Prozeß bereit, die *Reply-Message* aufzunehmen, jedoch verbietet es sich, diesem Prozeß physikalisch die Nachricht zuzustellen, da sein Adreßraum nicht definiert ist. Durch den für jeden Prozeß im Kerneladreßraum zur Verfügung gestellten Platz zur Aufnahme einer Kopie der betreffenden Nachricht, wird dieses Problem vollständig umgangen.

Die Entwurfsentscheidung für die Semantik der jetzigen Operationen des Kernels zum *Message-Passing*, stellen ein typisches Beispiel für den in [Parnas, Siewiorek 1972] skizzierten "Verlust an Transparenz" solcher Operationen für die applizierenden Prozesse dar. Der Transparenzverlust ergibt sich bei AX im Zusammenhang mit der Portierung des Systems auf komplexe Hardware-Architekturen. Die Mechanismen des Kernels zur Interprozeßkommunikation sind als vollständig portabel anzusehen; nur auf der untersten Ebene ergeben sich bei den Datentransferoperationen Hardware-Abhängigkeiten. Dennoch würde die gegenwärtige Realisierung dieser Mechanismen für die applizierenden Prozesse einen enormen Leistungsverlust aufzeigen können. Dies kann dazu führen, daß bei bestimmten Applikationen auf den unterschiedlichen Portierungen von AX, selbst gravierende Leistungsunterschiede zu verzeichnen sind. Diese Leistungsunterschiede können im Extremfall den "korrekten" Ablauf insbesondere zeitkritischer Applikationen nicht mehr garantieren.

Die Entwurfsentscheidung für Mechanismen zur Interprozeßkommunikation, die auf Nachrichten fester Länge basieren, würde diesen möglichen Transparenzverlust für die applizierenden Prozesse weitestgehend vermeiden. In [Cheriton, Zwaenepoel 1983] sind Ergebnisse angegeben, wo sich die Laufzeiten von *Rendezvous* mit fester Nachrichtengröße und Zwischenpufferung im Kerneladreßraum auch auf komplexer Hardware (mc68000 mit MMU) im Bereich von 1 Millisekunde bewegen.

7.4. NIXE und PEACE

Die Realisierung der in den vorherigen Abschnitten kurz skizzierten Aspekte zur Motivation der Überarbeitung der Mechanismen zur Interprozeβkommunikation von AX, wird ein wesentlicher Inhalt der nächsten Arbeit mit MOOSE darstellen. Die Änderungen des Kernels betreffen hierbei insbesondere nur den Teil der Interprozeβkommunikation, und die Änderungen in den Systemprozessen von AX beziehen sich lediglich auf wenige Stellen, an denen die Mechanismen zur Interprozeβkommunikation angewendet werden [52]. Insgesamt beläuft sich der Aufwand der Änderungsmaβnahmen auf weniger als eine Mannwoche (incl. Testen).

Diese Änderungsmaβnahmen stellen jedoch nicht die wesentlichen Inhalte der zukünftigen Arbeit mit MOOSE dar. Vielmehr gilt es, das Familienkonzept von MOOSE weiter anzuwenden und durch zusätzliche Mitglieder die MOOSE-Betriebssystemfamilie anzureichern. Konkret ist hierbei an die Vervollständigung des ersten kompletten UNIX-Mitglieds gedacht, daβ im wesentlichen Funktionalitäten entsprechend X/OPEN [Bull et al. 1985], System V [Bell 1983] und 4.2BSD [Joy et al. 1983] beherbergen wird. Ein anderes wesentliches Mitglied der Betriebssystemfamilie wird MS-DOS darstellen.

Die Hauptzielrichtung der Weiterentwicklung von MOOSE besteht jedoch darin

a) den objektorientierten mit dem prozeβorientierten Betriebssystementwurf in homogener Weise zusammenzufügen und

b) dezentrale bzw. verteilte (Betriebs-/Applikations-) Systeme unterstützten zu können.

Für den ersten Fall gilt es, die in [Shapiro et al. 1985] skizzierten Ansätze des *proxy principle* weiter zu verfolgen und mit dem Dienstbegriff unter MOOSE in Verbindung zu bringen. Der zweite Fall bedeutet, die mit dem V-System [Cheriton 1984] aufgezeigte Linie nachzuvollziehen und für eine breite Klasse von Hardware-Architekturen im Sinne von MOOSE weiter zu entwickeln. Die Verschmelzung beider Gesichtspunkte soll zu einem auf Dienst-/Prozeβebene dynamisch rekonfigurierbaren und verteilten UNIX-ähnlichen Betriebssystem führen.

Das resultierende System ist als ein weiteres Mitglied der MOOSE-Betriebssystemfamilie zu betrachten. Es wird auf den bewährten Prinzipien von AX basieren und die Netzwerkfähigkeit weiter ausbauen. Die wesentliche Voraussetzung hierbei besteht darin, daβ die Art des in Anwendung gebrachten Netzwerkes (LAN, WAN, aber auch Multiprozessorsysteme) transparent für die Prozesse eines Systems sein soll. Dieses System stellt damit ein "*Network Independent UNIX Environment*" zur Verfügung; es wird abkürzend mit NIXE bezeichnet werden.

Neben NIXE, das vorwiegend im Rahmen projektorientierter Lehrveranstaltungen von Studenten verwirklicht wird, finden die Konzepte von MOOSE, sowie die *Redesign*-Aspekte von AX, ihre Anwendung bei der Realisierung eines dezentralen/verteilten Betriebssystems zur Unterstützung numerischer Anwendungen. Dieses System definiert ein "*Process Execution And Communication Environment*" und wird mit PEACE bezeichnet. Erste Entwurfskonzepte sind [Schroeder 1986] zu entnehmen.

[52] Ebenso kann durch Anwendung einer elementaren *Compatibility-Library* die bisherige Schnittstelle zum Kernel weiterhin zur Verfügung gestellt werden.

Neben der Verteiltheit von PEACE sind besonders harte Anforderungen an die Fehlertoleranz des Gesamtsystems gestellt. Die Mechanismen zur Fehlertoleranz werden entsprechend der Tradition von MOOSE über spezielle Systemprozesse bzw. Administratoren zur Verfügung gestellt. Hierzu gehören vor allem Administratoren, die *Checkpointing* und *Recovery* von Applikations- und Systemkomponenten kontrollieren. Ein weiteres Wesensmerkmal der Fehlertoleranz in PEACE werden *Konsistenz Checks* prozeßspezifischer Datenstrukturen darstellen, die eine frühzeitige Erkennung von Fehlfunktionen des (Betriebs-/Applikations-) Systems ermöglichen sollen.

PEACE basiert auf einer Netzwerk-Architektur, die eine physikalische Transferrate von bis zu 250 M *bytes* unterstützt [Behr et al. 1986]. Das Netzwerk selbst besteht aus einer großen Anzahl von Knoten, auf denen numerische Problemlösungen ablaufen. Ein Knoten besteht hierzu aus dem mc68020 Prozessor, unterstützt von einer *paging* MMU mc68851, und einem Vektorprozessor hoher Leistungsfähigkeit.

Literaturverzeichnis

[Akkoyunlu et al. 1974]
E. Akkoyunlu, A. Bernstein, R. Schantz: **Interprocess Communication Facilities for Network Operating Systems**, IEEE Computer, June, 46-55, 1974

[Bach, Buroff 1984]
M. J. Bach, S. J. Buroff: **Multiprocessor UNIX Operating Systems**, AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8 (October), 1984

[Balzer 1971]
R. M. Balzer: **Ports – A Method for Dynamic Interprogram Communication and Job Control**, Spring Joint Computer Conference, 1971

[Bauerfeld 1981]
W. L. Bauerfeld: **A Communication Concept for Protocol Models**, Hahn-Meitner-Institut Berlin, 1981

[Behr 1983]
P. Behr: **Entwurf eines verteilten Multicomputer-Systems**, Dissertation, TU Berlin, Fachbereich 20 (Informatik), 1983

[Behr et al. 1986]
P. M. Behr, W. K. Giloi, H. Mühlenbein: **Rationale and Concepts for the SUPRENUM Supercomputer Architecture**, Gesellschaft für Mathematik und Datenverarbeitung (GMD), 1986

[Bell 1983]
Bell Telephone Laboratories: **UNIX System User's Manual System V**, AT&T Bell Telephone Laboratories, Incorporated, 1983

[Bormann 1985]
C. Bormann: **Inter-Process Communication in UNIX – A Case Study**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1985

[Brownbridge et al. 1982]
D.R. Brownbridge, L.F. Marshall, B. Randell: **The Newcastle Connection**, Software Practice and Experience, 12, 1147-1162, 1982

[Bull et al. 1985]
Bull, ICL, Nixdorf, Olivetti, Philips, Siemens: **X/OPEN Portability Guide**, Part 1, The Common Applications Environment, Elsevier Science Publishers B. V., P.O Box 1991, 1000 BZ Amsterdam, The Nederlands, 1985

[Campbell, Habermann 1974]
R. H. Campbell, A. N. Habermann: **The Specification of Process Synchronization by Path Expressions**, Lecture Notes in Computer Science, 16, Springer Verlag, Heidelberg, 1974

[Cheriton 1979]
D. R. Cheriton: **Multi-Process Structuring and the Thoth Operating System**, Dissertation, University of Waterloo, UBC Technical Report 79-5, 1979

[Cheriton 1982]
D. R. Cheriton: **The Thoth System**, Elservier Science Publishing Co, 1982

[Cheriton 1984]
D. R. Cheriton: **The V Kernel: A Software Base for Distributed Systems**, IEEE Software 1, 2, 19-43, 1984

[Cheriton 1984b]
D. R. Cheriton: **An Experiment using Registers for Fast Message-Based Interprocess Communication**, ACM Operating Systems Review, 18, 4, 1984

[Cheriton 1985]
D. R. Cheriton: **Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems**, ACM Operating Systems Review, 19, 4, 1985

[Cheriton, Zwaenepoel 1983]
D. R. Cheriton, W. Zwaenepoel: **The Distributed V Kernel and its Performance for Diskless Workstations**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 1983

[Coffman, Denning 1973]
E. G. Coffman, P. J. Denning: **Operating Systems Theory**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973

[Crowley 1981]
C. Crowley: **The Design of a New UNIX Kernel**, AFIPS, 1981

[DEC 1978]
Digital Equipment Corporation: **PDP 11 Software Handbook**, Digital Equipment Corporation, 1978

[Deitel 1984]
H. M. Deitel: **An Introduction to Operating Systems**, Addison-Wesley Publishing Company, Inc., 1984

[Dennis, van Horn 1966]
J. B. Dennis, E. C. van Horn: **Programming Semantics for Multiprogrammed Computations**, Comm. ACM, 11, 3, 143-155, 1966

[Dijkstra 1968]
E. W. Dijkstra: **The Structure of the "THE"-Multiprogramming System**, Comm. ACM, 11, 5, 341-345, 1968

[Dijkstra 1968b]
E. W. Dijkstra: **Cooperating Sequential Processes**, Programming Languages, Academic Press, New York, 1968

[Fabry 1974]
R. S. Fabry: **Capability-Based Addressing**, Comm. ACM, 17, 7, 403-412, 1974

[Fiedler 1983]
D. Fiedler: **The UNIX Tutorial**, Part 1-3, BYTE Publications Inc., October, 1983

[Gentleman 1981]
W. M. Gentleman: **Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept**, Software Practice and Experience, 11, 435-466, 1981

[Goodenough 1975]
J. B. Goodenough: **Exception Handling: Issues and a Proposed Notation**, Comm. ACM, 18, 12, 683-696, 1975

[Habermann 1976]
A. N. Habermann: **Introduction to Operating System Design**, The SRA Computer Science Series, Science Research Associates, Inc., 1976

[Habermann et al. 1976]
A. N. Habermann, P. Feiler, L. Flon, L. Guarino, L. Cooprider, B. Schwanke: **Modularization and Hierarchy in a Family of Operating Systems**, Carnegie-Mellon University, 1976

[Hansen 1970]
P. Brinch Hansen: **The Nucleus of a Multiprogramming System**, Comm. ACM, 13, 4, 238-250, 1970

[Herrtwich 1985]
R. G. Herrtwich: **Grundlagen der Programmierung nebenläufiger und verteilter Systeme**, Skript zur Vorlesung "Algorithmen 4", TU Berlin, Fachbereich 20 (Informatik), KBS, 1985

[Hoare 1974]
C. A. R. Hoare: **Monitors: an Operating System Structuring Concept**, Comm. ACM, 17, 7, 549-557, 1974

[IBM 1984]
International Business Machines: **PC-DOS Disk Operating System Version 3.0**, IBM Corporation, Technical Reference, 1984

[ISO 1985]
ISO: **Information Processing Systems – Open Systems Interconnection – Basic Reference Model**, International Standard 7498, 1985

[Intel 1983]
Intel Corporation: **iAPX 286 Programmer's Reference Manual**, Intel Corporation, Santa Clara, California 95051, USA, 1983

[Isle et al. 1977]
R. Isle, H. Goullon, K.-P. Löhr: **Dynamic Restructuring in an Experimental Operating System**, Technical Report 77-27, TU Berlin, Fachbereich 20 (Informatik), 1977

[Joy et al. 1983]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, D. Mosher: **4.2 BSD System Manual**, University of California at Berkeley, CA 94720, 1983

[Kernighan, McIlroy 1979]
B. W. Kernighan, M. D. McIlroy, et al.: **The UNIX Timesharing System: UNIX Programmer's Manual, 7th Edition**, Bell Telephone Laboratories, Inc., Murray Hill, 1979

[Kernighan, Ritchie 1978]
B. W. Kernighan, D. M. Ritchie: **The C Programming Language**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978

[Lampson 1983]
B. W. Lampson: **Hints for Computer System Design**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 10-13 October, 1983

[Lions 1977]
J. Lions: **A Commentary on the UNIX Operating System**, Department of Computer Science, The University of New South Wales, Second Printing, 1977

[Lions 1977b]
J. Lions: **UNIX Operating System Source Code Level Six**, Department of Computer Science, The University of New South Wales, Second Printing, 1977

[Liskov 1972]
B. H. Liskov: **A Design Methodology for Reliable Software Systems**, Fall Joint Computer Conference, 1972

[Liskov 1979]
B. H. Liskov: **Primitives for Distributed Computing**, Proceedings of the Seventh ACM Symposium on Operating Systems Principles, 33-42, 1979

[Liskov 1981]
B. H. Liskov: **Report on the Workshop on Fundamental Issues in Distributed Computing**, ACM Operating Systems Review, 15, 3, 1981

[Loehr 1980]
K.-P. Löhr: **Architektur von Betriebssystemen**, Informatik-Spektrum 3, 229-245, (Teil1), Informatik-Spektrum 4, 40-48, (Teil2), Springer Verlag, 1980

[Loehr 1985]
K.-P. Löhr: **Newcastle Connection**, Informatik-Spektrum 8, 153-154, Springer Verlag, 1985

[Microware 1984]
Microware Systems Corporation: **OS-9/68000 Operating System**, Microware Systems Corporation, Des Moines, Iowa, Technical Manual, Revision C, 1984

[Mueller et al. 1980]
J. Müller, A. N. Habermann, K.-P. Löhr: **Address Space Management in the DAS Operating System**, Technical Report 80-35, TU Berlin, Fachbereich 20 (Informatik), 1980

[Mullender, Tanenbaum 1986]
S. J. Mullender, A. S. Tanenbaum: **The Design of a Capability-Based Distributed**

Operating System, The Computer Journal,, Vol. 29, No. 4, 1986

[Nelson 1982]
B. J. Nelson: **Remote Procedure Call**, Carnegie-Mellon University, Report CMU-CS-81-119, 1982

[Organick 1972]
E. Organick: **The Multics System: An Examination of its Structure**, MIT Press, 1972

[Panzieri, Shrivastava 1982]
F. Panzieri, S. K. Shrivastava: **Reliable Remote Calls for Distributed UNIX: An Implementation Study**, 2nd Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, USA, 1982

[Parnas 1974]
D. L. Parnas: **On a 'Buzzword': Hierarchical Structure**, Information Processing 74, North-Holland Publishing Company, 1974

[Parnas 1975]
D. L. Parnas: **On the Design and Development of Program Families**, Forschungsbericht BS I 75/2, TH Darmstadt, 1975

[Parnas 1976]
D. L. Parnas: **Some Hypotheses about the 'uses' Hierarchie for Operating Systems**, Report, TH Darmstadt, 1976

[Parnas, Siewiorek 1972]
D. L. Parnas, D. P. Siewiorek: **Use of the Concept of Transparency in the Design of Hierarchically Structured Systems**, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213, 1972

[Pause 1985]
H. Pause: **Entwurf und Implementierung eines "Message Passing" Systems**, Studienarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1985

[Popek et al. 1981]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel: **LOCUS: A Network Transparent, High Reliability Distributed System**, ACM Operating Systems Review, 15, 5, Proceedings of the Eigth Symposium on Operating Systems Principles, Asilomar Conference Grounds, Pacific Grove, California, 1981

[Powell, Miller 1983]
M. L. Powell, B. P. Miller: **Process Migration in DEMOS/MP**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 1983

[Quantum 1984]
Quantum Software Systems: **QNX Operating System User's Manual**, Version 2.0, Quantum Software Systems Ltd., Toronto, Canada, 1984

[Schindler 1978]
S. Schindler: **Das DAS-System**, GI Jahrestagung, Kurzvorträge, TU Berlin, 1978

[Schindler 1979]
S. Schindler: **Specification of Data Types for Distributed Systems**, Technischer Bericht 79-13, TU Berlin, Fachbereich 20 (Informatik), KBS, 1979

[Schindler 1979b]
S. Schindler: **Issues in Strong Data Typing**, Proceedings of the Louisiana Computer Exposition, Lafayette, Louisiana, March 2-3, 1979

[Schindler 1980]
S. Schindler: **Distributed Abstract Maschine**, Computer Communications, Vol. 3, No. 5, 1980

[Schindler 1983]
S. Schindler: **Grundlagen der Rechnerorganisation**, Skript zur Vorlesung "Rechnerorganisation 3", TU Berlin, Fachbereich 20 (Informatik), KBS, 1983

[Schroeder 1984]
W. Schröder: **Das MOOSE-System**, TU Berlin, Fachbereich 20 (Informatik), 1984

[Schroeder 1986]
W. Schröder: **Concepts of a Distributed Process Execution and Communication Environment**, (PEACE), Technical Report, GMD FIRST an der TU Berlin, 1986

[Schroeder et al. 1986]
W. Schröder, A. Stockmeier: **MOOSE Programmer's Manual, 1st Edition**, TU Berlin, Fachbereich 20 (Informatik), 1986

[Shapiro et al. 1985]
M. Shapiro, S. Habert, R. Dumeur, J.-D. Fekete: **SOMIW Operating System Design**, INRIA, Esprit project no. 367, "Secure Open Multimedia Integrated Workstation", Subtask 2: Operating System, 1985

[Shatz 1984]
S. M. Shatz: **Communication Mechanisms for Programming Distributed Systems**, IEEE Computer, June, 21-28, 1984

[Shaw 1974]
A. C. Shaw: **The Logical Design of Operating Systems**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974

[Tanenbaum 1984]
A. S. Tanenbaum: **Structured Computer Organization**, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984

[Theimer et al. 1985]
M. M. Theimer, K. A. Lantz, D. R. Cheriton: **Preemptable Remote Execution Facilities for the V-System**, ACM Operating Systems Review, 19, 5, Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985

[Thompson, Ritchie 1974]
K. Thompson, D. M. Ritchie: **The UNIX Timesharing System**, Comm. ACM, 17, 7, 365-375, 1974

[Tsichritzis, Bernstein 1974]
D. C. Tsichritzis, P. A. Bernstein: **Operating Systems**, Academic Press , 1974

[Tyner 1981]
P. Tyner: **iAPX 432 General Data Processor Architecture Reference Manual**, Intel Corporation, Santa Clara, California 95051, USA, 1981

[Walker et al. 1983]
B. Walker, G. Popek, R. English, C. Kline, G. Thiel: **The LOCUS Distributed Operating System**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 1983

[Wulf et al. 1973]
Wulf, Cohen, Corwin, Jones, Levin, Pierson, Pollack: **Hydra: The Kernel of a Multiprocessor Operating System**, Department of Computer Science, Carnegie-Mellon University, 1973

MOOSE Literaturverzeichnis

[Dehne 1986]
P. Dehne: **Emulation von Betriebssystemdiensten – exemplarische Realisierung einer QNX-Domäne in MOOSE**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), 1986

[Dunckern 1985]
C. Dunckern: **Entwurf und Implementierung eines Prozeßsystems**, Studienarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1985

[Durchgraf 1986]
G. Durchgraf: **Die Realisierung von Shared- und Virtual Memory Konzepten in einer prozeßorientierten Betriebssystemumgebung auf Grundlage des Message-Passing Konzeptes**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Kulessa 1986]
C. Kulessa: **Überwachung und Analyse der Aktivitäten eines prozeßorientierten und dezentralen Betriebssystems**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Mehls 1986]
K.-U. Mehls: **Applikationsorientierte und dynamisch rekonfigurierbare Ein-/Ausgabesysteme in Message-Passing orientierten Betriebssystemen**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Nolte 1986]
J. Nolte: **Entwurf und Implementierung eines Trap-/Interrupt-Schnittstellenmonitors zur Unterstützung der Fehleranalyse von Betriebssystemsoftware in einer QNX-Umgebung**, Studienarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Oestmann 1986]
B. Oestmann: **Remote Device und Remote File Access Mechanismen in Message-Passing orientierten Betriebssystemen**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Panzram 1986]
B. Panzram: **Accounting von Systemaktivitäten in einer prozeßorientierten Systemumgebung auf Grundlage des Message-Passing Konzeptes**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Patzelt 1986]
T. Patzelt: **Entwurf, Implementierung und Integration einer Treiberkomponente zur Unterstützung frameorientierter Geräte**, Studienarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Pause 1986]
H. Pause: **Eine Familie von Kommunikationssystemen zur Unterstützung eines dezentralen, dynamisch rekonfigurierbaren Betriebssystems**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Sander 1986]

M. Sander: **Generierungs- und Bootstrapverfahren von prozeβorientierten Betriebssystemen**, Studienarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

[Stockmeier 1986]

A. Stockmeier: **Dynamisch rekonfigurierbare Swap- und Schedulestrategien in einem prozeβorientierten Betriebssystem**, Diplomarbeit, TU Berlin, Fachbereich 20 (Informatik), KBS, 1986

Lebenslauf

Wolfgang Schröder, geboren am 13.11.55 in Berlin.

Nach Besuch von Grundschule, Realschule und Gymnasium Reifeprüfung 1974 in Berlin. Studium der Informatik in Berlin, mit abschließender Diplomprüfung (Dipl.-Inform.) an der Technischen Universität Berlin.

Bis 1981 wissenschaftlicher Mitarbeiter im Forschungsprojekt "Management in offenen Kommunikationssystemen" an der Technischen Universität Berlin im Fachbereich Informatik, Institut für Software und Theoretische Informatik, Fachgebiet Kommunikations- und Betriebssysteme. Übernahme als wissenschaftlicher Mitarbeiter mit Lehraufgaben im selbigen Institut.

Seit Juli 1986 als wissenschaftlicher Mitarbeiter bei der Gesellschaft für Mathematik und Datenverarbeitung (GMD) in der Forschungsstelle GMD FIRST an der Technischen Universität Berlin beschäftigt.