Das MOOSE-System

Wolfgang Schröder

Technische Universität Berlin
Fachbereich 20 (Informatik)
Institut für Software und Theoretische Informatik
Fachgebiet Kommunikations- und Betriebssysteme
Prof. Dr.-Ing. Sigram Schindler
Sekretariat FR 6-3
Franklinstraße 28/29
1000 Berlin 10

ZUSAMMENFASSUNG

In dieser Arbeit werden die Grundkonzepte eines prozeßorientierten Betriebssystems vorgestellt. Dieses Betriebssystem zeichnet sich durch eine konsequente Anwendung von Prozessen als Hilfsmittel zum strukturierten Systementwurf aus. Die einzelnen Prozesse des Betriebssystems stehen in einer eindeutigen Benutztrelation zueinander und kommunizieren, entsprechend der Benutztrelation, miteinander durch Anwendung des Message-Passing Modells. Es wird dargestellt, wie Multiprozessor- und Netzwerksysteme auf der Grundlage dieses prozeßorientierten Betriebssystem realisiert werden können.

Inhaltsverzeichnis

1. EINLEITUNG

Die bislang dominierenden Betriebssysteme – hervorgehoben sei hier das UNIX-System –[1] weisen einen prozedurorientierten Entwurf auf. Dies führt zu einer sehr komplexen Systemstruktur, da alle wesentlichen Komponenten solcher Betriebssysteme direkt, d.h. prozedural, in gegenseitiger Beziehung zueinander stehen. Demgegenüber steht die allgemeine Forderung nach der Korrektheit eines Programmsystems. Diese Forderung gilt im erhöhten Maße für ein Betriebssystem, da dessen Aufgabe es erfordert, die dem jeweiligen Rechnersystem zugeordneten Hardware-Resourcen zu verwalten. Fehler im Zusammenhang mit den jeweiligen Verwaltungsmaßnahmen können weitreichende Konsequenzen für das Gesamtsystem haben. Bei einem prozedurorientierten Betriebssystem wird gerade der Nachweis der Korrektheit erheblich dadurch erschwert, daß die zentralen Komponenten des Systems untereinander einen engen Zusammenhalt erfordern. Dies gilt ebenso im Zusammenhang mit dem Austesten einzelner Komponenten während der Entwicklungsphase des Betriebssystems oder im Zuge der funktionalen Anreicherung eines bestehenden Systems, durch zusätzliche Komponenten.

Der Nachweis der Korrektheit oder die Verifikation allgemein, stellen nicht allein die Problempunkte dar, die mit einem prozedurorientierten Betriebssystem verbunden sind. Ebenso ist ein solches System nur bedingt flexibel. Die enge Kopplung von Komponenten zur Fileverwaltung mit z.B. Komponenten zur Prozeßverwaltung, erschwert das Herauslösen bzw. Austauschen solcher Komponenten. Ohne spezielle Hardware-Unterstützung führen Änderungen in zentralen Komponenten eines prozedurorientierten Betriebssystems immer zur vollständigen, d.h. alle Komponenten betreffenden, Generierung eines Gesamtsystems[2].

Die Problematiken, die das prozedurorientierte Modell bei einem Betriebssystem mit sich bringen, können durch einen anderen Ansatz in elementarer Weise umgangen werden. Werden die Komponenten eines Betriebssystems als autonome Prozesse verstanden, ergibt sich die Möglichkeit der direkten Entkopplung der Komponenten voneinander. Desweiteren ermöglicht diese Sichtweise eine Flexibilität dahingehend, daß Betriebssystemkomponenten durch andere Komponenten ersetzt werden können, indem bereits bestehende Prozesse durch neue Prozesse repräsentiert werden.

Das Prinzip des prozeßorientierten Betriebssystems definiert hierzu den geeigneten Rahmen. Es bringt aber ebenso mit sich, daß die Interaktionen zwischen den Komponenten (bzw. zwischen Benutzerprozessen und dem Betriebssystem) über andere Mechanismen bewerkstelligt werden müssen, als dies bei prozedurorientierten Betriebssystemen der Fall ist. Da die Betriebssystemkomponenten durch Prozesse realisiert werden, müssen solche Interaktionen auf die Kommunikation zwischen Prozessen zurückgeführt werden. Hierfür stellt das message passing Modell den adäquaten Mechanismus zur Verfügung, der zudem bereits durch einen sehr elementaren Kernel realisiert werden kann. Dieser Kernel hat nur die Aufgabe die Kommunikation innerhalb des Gesamtsystems auf elemtare Art und Weise zu ermöglichen.

Das hier vorgestellte Modell eines Betriebssystems bezieht sich einerseits auf eine streng prozeßorientierte Systemstruktur und basiert andererseits auf einem Message-Passing Kernel. Eine wesentliche Anforderung an das System ist es, Betriebssystemkomponenten durch Prozesse in der Art realisiert zu wissen, daß ein breites Einsatzspektrum des Systems möglich ist. Dies bedeutet etwa, daß für spezielle Applikationen das optimale Betriebssystem zusammengestellt werden kann. Der Name des Systems soll diesen Ansatz verdeutlichen: MOOSE , für "**M**essage **O**riented **O**perating **S**ystem **E**nvironment", definiert den Rahmen, durch den applikationsspezifische Betriebssysteme geformt werden können, und steht somit stellvertretend für eine Familie von Betriebssystemen.

---

[1] UNIX ist ein eingetragenes Warenzeichen von *A T & T Bell Laboratories* . Siehe auch [Thompson, Ritchie 1974].

[2] Das Multics-System [Organick 1972] zeigte bereits, welcher Aufwand an Hardware und Software notwendig ist, um ein dynamisch rekonfigurierbares prozedurorientiertes Betriebssystem zu realisieren.

Die Realisierung des ersten Prototyps eines MOOSE-Betriebssystems orientierte sich im wesentlichen an zwei bestehenden Betriebssystemen. Die Systemschnittstelle dieses Prototyps ist sehr stark durch die des UNIX-Betriebssystems geprägt. Damit ist die Vorraussetzung gegeben, von dem gegenwärtig enormen Erfolg des UNIX-Systems dahingehend Nutzen ziehen zu können, prinzipiell auf eine groβe Menge bestehender Dienstprogramme zurückzugreifen. Das "Innenleben" des MOOSE-Prototyps entspricht jedoch anderen konzeptionellen und technischen Maximen. Den prägenden Einfluβ hatte hierbei das Thoth-System [Cheriton 1982], von dem im wesentlichen das dort verwirklichte Team-Konzept (als die besondere Variante eines Prozeβmodells) übernommen und weiterentwickelt worden ist. Dieser Prototyp wird einerseits wegen seiner Systemschnittstelle und andererseits wegen seiner technischen Repräsentation Advanced UNIX, kurz AX , genannt.

AX stellt die konsequente Weiterentwicklung des prozeβorientierten und auf einen Message-Passing Kernel basierenden Thoth-Systems dar. Dies zeigt sich insbesondere im Zusammenhang mit der Behandlung von Traps und Interrupts. Das hierfür bei AX in Anwendung gebrachte Modell stellt einen wesentlichen Beitrag zur Flexibilität, Leistungsfähigkeit und Sicherheit eines Betriebssystems dar.

2. KONZEPTE

Die Konzepte, nach denen MOOSE aufgebaut ist, bzw. allgemein die Grundlage für einen flexiblen Betriebssystementwurf legen, sollen anhand einiger Anforderungen herausgearbeitet werden.

2.1. Abstraktion durch Prozeβhierarchien

Das Konzept einer Familie von Betriebssystemen bedeutet, unterschiedliche Systeme aufbauen zu können, indem auf identische oder gleichartige Komponenten zurückgegriffen wird. Basierend auf einer elementaren Menge von Bausteinen werden komplexe Komponenten geschaffen, die entweder der gewünschten Funktionalität entsprechen oder selbst wieder als Bausteine noch komplexerer Komponenten verstanden werden. Um solch einen Ansatz verwirklichen zu können, muβ in erster Linie ein streng modularer Systementwurf im Vordergrund stehen. Ebenso ist es aber wesentlich, die geeigneten technischen Hilfsmittel zur Verfügung zu haben, um die einzelnen Bausteine derart miteinander zu verknüpfen, daβ ein bestimmtes funktionales System gebildet werden kann. Diese Hilfsmittel unterscheiden sich prinzipiell darin, zu welchem Zeitpunkt die Verknüpfung der einzelnen Bausteine erfolgt. Ein hohes Maβ an Flexibilität wird erreicht, wenn der Bezug zwischen den jeweiligen Bausteinen nicht bereits bei der Generierung des entsprechenden Systems erfolgt, sondern erst dann, wenn das System erzeugt und zur Ausführung gebracht wird.

Zwischen den einzelnen Komponenten eines Systems finden Interaktionen statt, um bestimmte Funktionalitäten zu erbringen. Die Komponenten stellen Dienste zur Verfügung, die von anderen Komponenten benutzt werden können um komplexere Dienstanforderungen zu erfüllen. Die prozedurale Realisierung (auf der Ebene des realen Prozessors betrachtet) dieser Benutztbeziehung zwischen den betreffenden Komponenten und das Ziel, den Bezug zwischen benutzender und benutzter Komponente nicht bereits bei der Generierung des jeweiligen Systems statisch herzustellen, bedingt spezielle Anforderungen an die zugrunde liegende Hardware [Organick 1972]. Die Verwaltung dieser Hardware stellt sich als sehr komplex dar und kann nur von hierarchisch sehr tief angeordneten Komponenten des Systems erbracht werden.

In MOOSE werden die Komponenten zum Aufbau von Mitglieder einer Betriebssystemfamilie durch spezielle Prozesse, *Administratoren* genannt, realisiert. Demzufolge wird die Interaktion zwischen den Komponenten auf die Kommunikation zwischen den jeweiligen Prozessen zurückgeführt. Diese Aufgabe, d.h. die Realisierung der Inter-Prozeβ Kommunikation, übernimmt ein sehr elementarer Kernel auf der Grundlage des Message-Passing Modells. Mit diesem Ansatz ergibt sich direkt die Möglichkeit, ein dynamisch rekonfigurierbares System auf der Ebene von Prozessen zu realisieren, ohne dazu auf spezielle Hardware zurückgreifen zu müssen. Damit ist die wesentliche Grundlage geschaffen, insbesondere auch für den PC-Bereich [3], ein Betriebssystemmodell einheitlich auf einer breiten Palette von Hardware realisieren zu können, um somit auf eine Vielzahl von (sehr unterschiedlichen) Systemen dieselben, zumindest jedoch sehr ähnliche Dienste anzubieten.

In einem prozeβorientierten Betriebssystemmodell wird die funktionale Anreicherung eines Systems durch die zusätzliche Integration entsprechender Administratoren erlangt. Ebenso bedeutet das Entfernen von Administratoren aus einem bestehenden Systemkomplex, daβ die Menge der zur Verfügung stehenden Systemdienste entsprechend verkleinert wird. Nur wenn Systemdienste in Anspruch genommen werden, besteht die Notwendigkeit, den entsprechenden Administrator in dem jeweiligen Systemkomplex zu halten.

[3] Personal Computer.

2.2. Inter-Prozeß Kommunikation

Die Aufgabe des MOOSE-Kernels besteht darin, allgemeine Kommunikationsmechanismen zur Verfügung zu stellen. Diese Mechanismen ermöglichen es, daß

a) Prozesse anderen Prozessen Nachrichten bzw. Signale übermitteln können,

b) aufgrund eines Traps für den auslösenden Prozeß eine Trap-Nachricht dem entsprechenden Trap-Server zugesandt wird,

c) auf Anforderung von Interrupt-Handlern den Interrupt-Servern Signale zugestellt werden.

Das Grundprinzip des Kernels basiert auf dem Message-Passing Modell. Prozesse kommunizieren hierbei untereinander durch den Austauch von Nachrichten. Eine besondere Form von Nachrichten stellen hierbei Signale dar: sie repräsentieren Nachrichten ohne Inhalt, d.h. für ihre Übermittlung brauchen keine Transfer-Operationen angewendet werden.

Nachrichten und Signale werden vom Kernel durch bestimmte Deskriptoren verwaltet, die allgemein als *Ports* bezeichnet werden. Jeder Prozeß erhält bei seiner Erzeugung einen Deskriptor, den *Domain-Port* , fest zugeordnet. Dieser Port wird vom Kernel verwendet, um Nachrichten und den Kommunikationspartner identifizieren zu können. Um Signale empfangen zu können, fordert der entsprechende Prozeß einen freien Port vom Kernel an. Sobald dieser Port bekannt ist, können Prozesse auf diesen Port Signale absetzen und sich dadurch mit dem Empfängerprozeß synchronisieren.

2.2.1. *Das Rendezvous-Konzept*

Das Message-Passing Modell des MOOSE-Kernels bedeutet, daß miteinander kommunizierende Prozesse ein *Rendezvous* eingehen müssen, damit der Transfer der Nachrichten erfolgen kann. Diese Prozesse müssen somit aufeinander synchronisiert sein, was zur Folge hat, daß die entsprechenden Kommunikationsprimitiven blockierend für den jeweils anwendenden Prozeß wirken, wenn sein Kommunikationspartner nicht zum Rendezvous bereit ist.

Ein Rendezvous findet dann statt, wenn

a) der die Nachricht übermittelnde Prozeß (Sender) einen empfangsbereiten Prozeß, oder

b) der eine Nachricht empfangende Prozeß (Empfänger) mindestens einen sendebereiten Prozeß

adressiert. Das Rendezvous ist dann beendet, wenn der Empfänger den Sender explizit reaktiviert. Dazu gibt der Empfänger eine *Reply-Message* an den Sender zurück. Bild 2.1 skizziert die Zustände, die Sender und Empfänger während eines Rendezvous' durchlaufen.

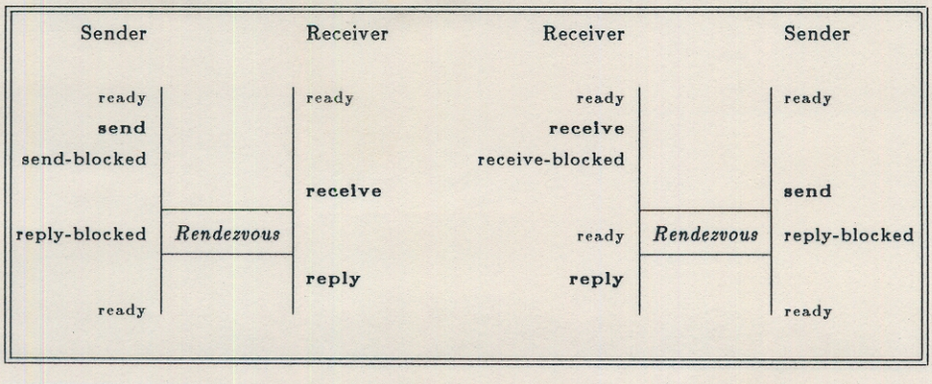

Bild 2.1: Zustände während eines Rendezvous'

Das Rendezvous-Konzept, bei dem Sender und Empfänger jeweils blockieren, gestattet eine sehr effiziente Realisierung des Message-Passing Modells. Zum anderen erfordert eine Realisierung dieses Konzeptes einen minimalen Aufwand und trägt somit erheblich dazu bei, den Kernel sehr elementar gestalten zu können. Der wesentliche Aspekt ist, neben der Laufzeiteffizienz, daß der Kernel die Nachrichten nicht zwischenspeichern muß, was bei nicht blockierenden Kommunikationsmechanismen der Fall wäre. Zwischenspeichern der Nachrichten erhöht prinzipiell das Risiko von *Deadlocks* innerhalb des Kernels, da "jederzeit" Pufferplatz zur Aufnahme der Nachrichten bereitstehen muß.

2.2.2. *Broadcasting*

Ein prozeßorientiertes Betriebssystem ist mit der Problematik konfrontiert, daß jeder System-Administrator spezifische Informationen über alle im System existierenden Prozesse halten muß. Der Filesystem-Administrator benötigt z.B. bestimmte Informationen, um Zugriffsrechte der jeweiligen Prozesse auf Files überprüfen zu können. In dem NUKE-System [Crowley 1981], daß ein Redesign des UNIX-V6 Kernels darstellt, ist es den Administratoren ermöglicht worden, auf eine zentrale, jedem Prozeß zugeordnete *User Structure* zugreifen und diese entsprechend modifizieren zu können. Diese Lösung bedeutet aber, daß

a) mit jeder Anforderung eines Prozesses, der jeweilige Administrator die User-Structure des Prozesses in seinen Adreßraum einblenden muß, was zusätzliche Laufzeit bei der Erfüllung der Anforderung bedeutet;

b) die Realisierung eines dezentralen oder gar verteilten Betriebssystems auf dieser Grundlage mit eheblichen Problemen konfrontiert wird.

Der MOOSE-Kernel stellt deshalb einen *Broadcast* -Mechanismus zur Verfügung, der von den Administratoren in Anspruch genommen werden kann, um für jeden Prozeß (auch und gerade in einem verteilten System) entsprechende administrative Maßnahmen durchführen zu können. Nach seiner Erzeugung initiiert ein Prozeß die Broadcast-Nachricht, die dann allen Administratoren des bestehenden MOOSE-Systems übermittelt wird. Die gleiche Maßnahme findet bei der Zerstörung eines Prozesses statt. Hierbei wird dafür gesorgt, daß, bevor der Prozeß tatsächlich zerstört d.h. aus dem Speicher genommen wird, sich der zu terminierende Prozeß selbst beim System abmeldet. Bild 2.2 verdeutlicht den Aufbau einer solche Kette von Administratoren, die an einem Broadcast beteiligt sind.

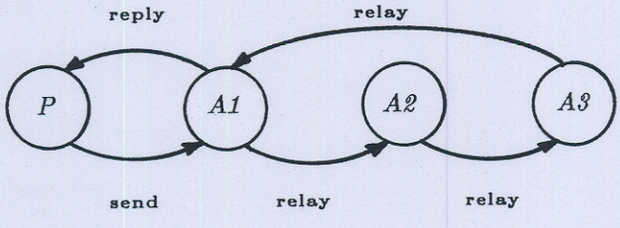

Bild 2.2: Administratoren während eines Broadcasts

Der Broadcast wird dadurch aufrecht erhalten, daß jeder Administrator der eine Broadcast-Nachricht empfangen hat, diese an den nächsten weitergibt. Die Selektion des nächsten Administrators übernimmt der Kernel. Beendet wird der Broadcast dadurch, daß ein Administrator den den Broadcast initiierenden Prozeß durch eine Reply-Message reaktiviert.

Die Administratoren können selektiv auf bestimmte Broadcast-Nachrichten warten. Jede Broadcast-Nachricht ist einer bestimmten Klasse zugeordnet, z.B. Zerstörung eines Prozesses, und die Administratoren können bestimmen, auf welche Klassen von Nachrichten sie reagieren wollen.

2.3. Behandlung von Traps und Interrupts

Die Trap-/Interrupt-Behandlung des MOOSE-Systems findet unter Kontrolle von Server-Prozessen statt und ist somit aus dem Kernel herausgezogen. Die Aufgabe des Kernels in diesem Zusammenhang ist es, eine Verbindung zwischen Trap-/Interrupt-Vektoren des realen Prozessors und bestimmten Server-Prozessen herzustellen und die Aktivierung dieser Server-Prozesse zu bewirken, wenn der jeweilige Trap/Interrupt über den entsprechenden Vektor signalisiert wird. Aufgrund der unterschiedlichen Eigenschaften von Traps (*synchron*) und Interrupts (*asynchron*), findet die Aktivierung des jeweiligen Server-Prozesses über verschiedene Mechanismen statt.

2.3.1. *Traps*

Die Trap-Behandlung innerhalb des MOOSE-Kernels beschränkt sich darauf, eine Kommunikation zwischen dem Prozeß, der den Trap ausgelöst hat, und einem entsprechenden Trap-Server zu ermöglichen. Diese Kommunikation bedeutet wiederum ein Rendezvous zwischen beiden betrachteten Prozessen, mit dem Unterschied, daß dieses Rendezvous transparent für den den Trap auslösenden Prozeß stattfindet.

Trap-Server sind spezielle Prozesse, denen es gestattet ist, sich an Trap-Vektoren des zugrunde liegenden Prozessors anzubinden. Wird ein Trap über einen Vektor angenommen, an dem ein Trap-Server angebunden ist, baut der MOOSE-Kernel eine Trap-Nachricht auf und sendet diese, unter der Identifikation des den Trap auslösenden Prozesses, dem entsprechenden Server-Prozeß zu. Die Trap-Nachricht enthält den vollständigen Prozessorzustand des unterbrochenen Prozesses. Der Trap-Server kann anhand dieser Nachricht den jeweils ausgelösten Trap identifizieren und eine entsprechende, evtl. benutzerspezifische, Behandlung einleiten. Die Trap-Nachricht wird im Zuge des Rendezvous' an den unterbrochenen Prozeß zurückgesandt und der Prozessorzustand anhand dieser Nachricht für diesen Prozeß wieder hergestellt.

Welche Semantik mit den jeweiligen Traps assoziiert wird, bestimmen allein die Server-Prozesse. Mit diesem Aspekt wird es erheblich vereinfacht, *Emulatoren* bestimmter Betriebssystemdienste[4] in das MOOSE-System zu integrieren.

2.3.2. *Interrupts*

Innerhalb des MOOSE-Kernels finden keine zur Behandlung von Interrupts notwendigen und geräteabhängigen Operationen statt[5]. Vielmehr werden diese Operationen durch bestimmte Administratoren vollzogen. Im Zuge der Behandlung von Interrupts ermöglicht es der Kernel hierbei, daß

a) Interrupt-Handler in dem Adreßraum des entsprechenden Administrators aktiviert werden können, und

b) eine Kommunikation zwischen Interrupt-Handler und dem entsprechenden Server-Prozeß stattfindet.

Die Verwaltung der Geräte findet unter Kontrolle eines Server-Prozesses (dem *Interrupt-Server*) statt, der vom Eintreten des Interrupts benachrichtigt werden muß. Diese Benachrichtigung sollte es erlauben, den Interrupt-Handler schnellst möglich wieder terminieren zu lassen, so daß er sich nach kurzer Laufzeit wieder auf einen neuen Interrupt synchronisieren kann. Aus diesem Grunde scheiden die nachrichtenorientierten Kommunikationsprimitiven des MOOSE-Kernels aus, da durch den Nachrichtentransfer unbestimmte Laufzeiten für den Interrupt-Handler auftreten können. Desweiteren führen diese Primitiven, aufgrund des Rendezvous-Konzeptes, zur

[4] Solche Dienste werden üblicherweise durch besondere Trap-Instruktionen eingeleitet, um den Übergang vom Benutzer- in den Systemadreßraum zu erreichen. Demzufolge muß ein Trap-Handler existieren, der eine Abbildung des vom Benutzer geforderten Systemdienstes auf systeminterne Operationen vollzieht.

[5] Die einzige Ausnahme stellen die Operationen dar, die die Realtime-Clock des Kernels programmieren.

Blockierung des Interrupt-Handlers, bzw. des von dem Interrupt unterbrochenen Prozesses. Der Interrupt-Handler benachrichtigt seinen Server-Prozeß durch die Übermittlung eines Signals oder er leitet Nachrichten bereits blockierter Prozesse weiter. Diese Kommunikationsvarianten führen nicht zur Blockierung des übermittelnden Prozesses und es werden ebenfalls keine Nachrichten transferiert.

Zur Durchführung der Interrupt-Behandlung stellt der MOOSE-Kernel zwei voneinander unabhängige Varianten zur Verfügung:

Interrupt-Links dienen zur prozeduralen Aktivierung des Interrupt-Handlers, d.h. der Interrupt wird unter Kontrolle des gegenwärtig unterbrochenen Prozesses behandelt;

Interrupt-Tasks dienen zur vollständigen Entkopplung der Interrupt-Behandlung von den restlichen Systemaktivitäten, d.h. der Interrupt wird immer unter Kontrolle eines eigens dafür gestarteten Prozesses behandelt.

Beide Varianten der Interrupt-Behandlung unter MOOSE zeichnen sich dadurch aus, daß die jeweiligen Interrupt-Handler demselben Adreßraum zugeordnet sind wie ihre Server-Prozesse. Die Manipulation von Datenstrukturen, die etwa die Ein-/Ausgabeoperationen steuern sollen, kann durch beide Komponenten (Interrupt-Handler und Server-Prozeß) direkt und somit effizient erfolgen. Der Kernel stellt sicher, daß zum Zeitpunkt des Interrupts der dem Interrupt-Handler zugeordnete Adreßraum für den Handler verfügbar ist. Die Identifikation des jeweiligen Adreßraumes richtet sich nach dem Server-Prozeß, der sich an den entsprechenden Interrupt-Vektor angebunden hat.

Die Aktivierung eines Interrupt-Handlers nach dem Interrupt-Task Modell bedeutet das Starten der entsprechenden Interrupt-Task. Der Interrupt-Server erzeugt vorher die jeweilige Interrupt-Task und gibt ihre Identifikation dem Kernel bekannt. Das Interrupt-Link Modell bedingt einen *virtuellen Interrupt-Vektor* im Adreßraum des entsprechenden Interrupt-Servers. In diesem Vektor sind die Adressen der Interrupt-Handler verzeichnet. Vor Anbindung an den realen Interrupt-Vektor "i" des zugrunde liegenden Prozessors, definiert der Interrupt-Server seinen virtuellen Interrupt-Vektor "i" mit der Adresse des entsprechenden Interrupt-Handlers. Aufgrund der direkten Beziehung zwischen realen und virtuellen Interrupt-Vektor, ist es dem Kernel zum Zeitpunkt des Interrupts möglich, den jeweiligen Interrupt-Handler eindeutig zu identifizieren.

2.3.3. *Synchronisation innerhalb des Kernels*

Die Kommunikation zwischen Interrupt-Handler und Interrupt-Server wird dadurch erschwert, daß sie mit den restlichen Aktivitäten des Systems synchronisiert werden muß. Diese wesentliche Aufgabe übernimmt der Kernel. Um die Kommunikationsprimitiven für ihre häufige Anwendung so effizient wie möglich gestalten zu können, wird der gesamte MOOSE-Kernel als kritischer Abschnitt betrachtet; er definiert einen *Monitor* im Sinne von [Hansen 1973] bzw. [Hoare 1974]. Die Motivation für diesen Ansatz ergibt sich aufgrund der nachfolgend skizzierten Fakten:

a) der MOOSE-Kernel ist sehr elementar aufgebaut und seine Laufzeiten sind deterministisch;

b) die Laufzeiten für die Kommunikationsprimitiven und die Komplexität des Kernels würden sich erhöhen, wenn die kritischen Operationen jeweils selbst die Synchronisation initiieren;

c) die Synchronisation der Kommunikationsaktivitäten sollte nur dann angewendet werden, wenn dies notwendig ist, d.h. wenn Interrupt-Handler eine Kommunikation mit ihren Server-Prozessen anfordern;

d) die Markierung, wann eine Synchronisation bestimmter Kernelprimitiven stattfinden muß, ist an der Trap-/Interrupt-Schnittstelle des Kernels sehr effizient realisierbar.

Fordert ein Interrupt-Handler die Kommunikation mit seinem Server-Prozeß an, wird diese dann direkt stattfinden, wenn nur ein Kernel-Prozeß [6] aktiv ist. Ist mehr als ein Kernel-Prozeß aktiv, wird die Benachrichtigung des Server-Prozesses zurückgestellt und der von dem Interrupt unterbrochene Kernel-Prozeß reaktiviert. Die Abarbeitung aller zurückgestellten Benachrichtigungen wird von dem Kernel-Prozeß durchgeführt, der als letzter den Kernel verläßt.

Da die Interrupt-Handler immer über den Kernel aktiviert werden (unabhängig von der in Anwendung gebrachten Variante), ist es zweckmäßig, die Kommunikation mit ihren Server-Prozessen zum Zeitpunkt der Termination der Interrupt-Behandlung durchzuführen. Dann nämlich wird der Kernel wiederum aktiv (um den durch den Interrupt unterbrochenen Prozeß zu reaktivieren) und es kann in kontrollierter Weise die Kommunikation eingeleitet werden. Hierzu definieren die Interrupt-Handler bei ihrer Terminierung ob und in welcher Form (Signale oder weiterleiten von Nachrichten) ihre jeweiligen Server-Prozesse benachrichtigt werden.

2.4. Identifikation von Prozessen

Im Zusammenhang mit der Inter-Prozeß Kommunikation ist es notwendig, den jeweiligen Kommunikationspartner identifizieren zu können, bevor der eigentliche Transfer der Nachrichten erfolgt. Um dabei ein hohes Maß an Flexibilität zu erreichen, muß die Identifikation des Kommunikationspartners eine Abstraktion von seiner Lage im System und von seiner Struktur ermöglichen. Dies bedeutet, daß der Mechanismus zur Inter-Prozeß Kommunikation allgemein eine abstrakte Maschine bzw. verteilte abstrakte Maschine darstellt, die Prozessen Dienste an bestimmten "service access points" zur Verfügung stellt [Schindler 1980].

Die Prozeßidentifikation wird im MOOSE-System durch zwei voneinander unabhängige Mechanismen realisiert, die jeweils in einem Administrator und dem Kernel angesiedelt sind.

2.4.1. *Service Mapping*

Administratoren stellen allgemein bestimmte Dienste zur Verfügung, die von anderen Prozessen (bzw. Administratoren) in Anspruch genommen werden können. Dazu muß aber in jedem Fall eine Kommunikation zwischen dienstanfordernden und diensterbringenden Prozeß ermöglicht werden. Entsprechend des Message-Passing Konzeptes des MOOSE-Kernels wird der jeweilige diensterbringende Prozeß mit seiner Process-ID adressiert. Damit stellt sich für den anfordernden Prozeß die Notwendigkeit, dem Dienst eine Process-ID zuordnen zu können. Diese Zuordnung wird durch einen speziellen Administrator des MOOSE-Systems (der *Service-Administrator*) realisiert.

Der Service-Administrator verzeichnet alle Dienste, die von der gegenwärtigen Konfiguration des MOOSE-Systems zur Verfügung gestellt werden. Anfordernde Prozesse erhalten vom Service-Administrator die Identifikation des Administrators, der für die Erbringung eines bestimmten Dienstes verantwortlich ist. Die Administratoren haben dafür zu sorgen, daß die durch sie zur Verfügung gestellten Dienste beim Service-Administrator verzeichnet sind.

Die Zuordnung von Diensten zu Prozessen wird selbst durch Dienste des Service-Administrators erbracht. Dies bedeutet aber auch, daß die dienstanbietenden bzw. dienstanfordernden Prozesse mit dem Service-Administrator kommunizieren und damit seine Identifikation kennen müssen. Diese Identifikation kann aber ohne eine Kommunikation mit dem Service-Administrator nicht erhalten werden. Um dieses offensichtliche Identifikations-Problem zu umgehen, bindet sich der Service-Administrator an einen festen *Port* [7] des MOOSE-Kernels an. Prozessen wird hiermit die Möglichkeit zur Verfügung gestellt, die Identifikation des jeweils

[6] Kernel-Prozesse repräsentieren Aktivitäten des Kernels, die allgemein aufgrund von Traps und Interrupts hervorgerufen werden.

[7] Ports stellen die Objekte des Kernels dar, über die allgemein die zu übermittelnden Nachrichten bezeichnet werden (*Message Descriptor*).

angebundenen Prozesses (in diesem Fall der Service-Administrator) vom Kernel zu erfahren.

2.4.2. *Process Mapping*

Die Mechanismen zur Inter-Prozeβ Kommunikation des MOOSE-Kernels basieren auf der Adressierung der an einer Kommunikation beteiligten Prozesse, also über ihre Process-ID's. Die Identifikation der jeweiligen Kommunikationspartner erfolgt im MOOSE-Kernel indirekt über eine sogenannte *Process Map* . Mit jeder Process-ID wird direkt ein Eintrag in der Process-Map assoziiert, der dann den tatsächlichen Kommunikationspartner identifiziert. Diese Abbildung ermöglicht einen sehr effizienten Zugriff auf kernelinterne Datenstrukturen zur Verwaltung der einzelnen Prozesse. Von größerer Bedeutung ist hierbei jedoch die Abstraktion von dem jeweiligen Kommunikationspartner. Die Einträge in der Process-Map können von speziellen Server Prozessen modifiziert werden, um sich damit vor den eigentlichen empfangenden Prozessen zu schalten.

2.4.2.1. *Unabhängigkeit von der System-Topologie*

Die Realisierung von dezentral organisierten oder gar verteilten Systemen wird durch Server-Prozesse unterstützt. Diese Administratoren sorgen allgemein dafür, eine systemübergreifende Kommunikation zwischen Prozessen zu ermöglichen. Dies bedeutet etwa, bestimmte Netzwerkdienste zur Verfügung zu stellen.

Im Zusammenhang mit der Process-Map wird gerade bei der Verwirklichung eines verteilten MOOSE-Systems eine wesentliche Funktionalität erbracht. Für jeden Prozeβ existiert ein Eintrag in der Process-Map, nur identifizieren nicht alle Einträge notwendigerweise unterschiedliche Prozesse. In jedem Fall ist jeder durch die Process-Map identifizierte Prozeβ lokal einem bestimmten System zugeordnet. Alle nicht lokal zugeordneten Prozesse erfahren eine Abbildung auf einen bestimmten lokalen Server-Prozeβ. Die Aufgabe dieses Server-Prozesses besteht dann darin, eine Kommunikation zwischen unterschiedlichen Systemen zu ermöglichen. Bild 2.3 skizziert diesen Sachverhalt.

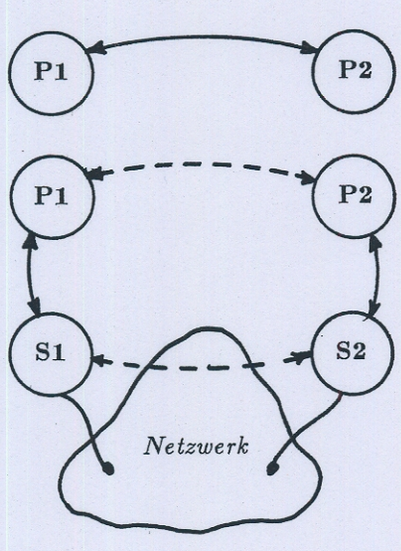

Bild 2.3: Abstraktion von der System-Topologie

In diesem Beispiel stellen die Prozesse **S1** und **S2** jeweils Server-Prozesse dar, die die Verbindungsglieder zwischen "entfernten" Teilen des Systems repräsentieren. Je nach Ausprägung der System-Topologie, kommunizieren die Prozesse **P1** und **P2** direkt miteinander (und dazu allein reichen die Primitiven des MOOSE-Kernels aus) oder sie kommunizieren indirekt

(für beide jedoch transparent) und mit Hilfe von Server-Prozessen miteinander. Die Server-Prozesse stellen in diesem Zusammenhang komplexe Kommunikationsmechanismen zur Verfügung und erlauben damit eine funktionale Anreicherung der im Kernel realisiert Kommunikationsdienste.

Bei der Kommunikation innerhalb eines verteilten Systems ist somit die Zuordnung von Prozessen zu Systemen transparent für die jeweiligen Kommunikationspartner. Diese Zuordnung kann weiterhin dynamisch gehalten werden, d.h. ohne Einfluß auf den jeweiligen Kommunikationspartner kann ein Prozeß zu unterschiedlichen Zeitpunkten verschiedenen Systemen zugeordnet sein.

2.4.2.2. *Überprüfbarkeit der Kommunikationsaktivitäten*

Eine weitere interessante Applikation, die durch die Process-Map unterstützt wird, ist die Überwachung von Kommunikationsaktivitäten innerhalb eines bestimmten Systems. Ein Server-Prozeß, der als *Monitor* fungiert, kann sich anstelle des eigentlichen Empfängerprozesses in eine bestehende Kommunikationsbeziehung einschalten. Für den Sender vollkommen transparent, empfängt der Monitor-Prozeß somit die Nachricht und kann etwa untersuchen,

- ob die vereinbarte Schnittstellenkonvention zwischen den Prozessen eingehalten wird,
- welche Funktionen wie häufig und mit welchen Parametern angesteuert werden.

Nach einer evtl. Untersuchung reicht der Monitor die entsprechende Nachricht an den eigentlichen Empfänger weiter.

Während der Testphase aber auch im laufenden Betrieb von MOOSE ist es sinnvoll, Kommunikationsaktivitäten zwischen den einzelnen Prozessen durch einen Monitor-Prozeß aufzeichnen zu können. Diese Möglichkeit gestattet bereits frühzeitig evtl. Fehler in der Benutzung der von bestimmten Prozessen zur Verfügung gestellten Dienste feststellen zu können. Ebenso kann die Verbesserung des Laufzeitverhaltens von MOOSE erreicht werden, wenn bekannt ist, wie ein typisches Anforderungsprofil zu bestimmten Systemprozessen (repräsentiert über entsprechende Nachrichten) aufgebaut ist.

2.5. Prozesse als Strukturierungshilfsmittel

Der Rahmen zur Modellierung von Prozeßgruppen stellt das Team-Konzept [Cheriton 1982] dar. Ein Team definiert einen Adreßraum, dem ein Prozeß oder mehrere Prozesse zugeordnet werden können. Diese Prozesse haben somit innerhalb eines Teams Zugriff auf gemeinsame Datenbestände, bzw. können dieselben Instruktionssequenzen zur Modifikation dieser Datenbestände anwenden. Jedem Prozeß ist jedoch ein eigener Stack zugeordnet, so daß sie jeweils einen eigenen Laufzeitkontext besitzen.

Das Team-Modell besitzt für viele Applikationen wesentliche Vorteile. Im Rahmen des MOOSE-Systems findet es in fast allen Bereichen seine Anwendung. Beispiele sind hierbei insbesondere im Zusammenhang mit der Verwaltung von Ein-/Ausgabesystemen, wie auch allgemein in der Realisierung spezifischer Dienste, etwa mit Hilfe von Bibliotheksfunktionen, gegeben.

2.5.1. *Dispatching*

Die Aktivierung und Blockierung von Prozessen findet grundsätzlich innerhalb des Kernels statt. Jede Blockierung eines Prozesses führt dazu, daß der Kernel einen anderen Prozeß, der nicht blockiert ist, zu aktivieren versucht. Hierbei werden zwei Ebenen bei der Blockierung eines Prozesses betrachtet. Auf der Prozeßebene wird versucht, möglichst innerhalb des aktiven Teams die Ausführung fortsetzen zu können. Dies bedeutet, daß bei der Blockierung eines Prozesses kein Teamwechsel stattfindet, wenn innerhalb seines Teams noch mindestens ein Prozeß zur Ausführung bereitsteht. Erst wenn alle Prozesse eines Teams blockiert sind oder die Zeitscheibe

für das Team abgelaufen ist, wird auf der Teamebene versucht, einen Prozeß eines anderen Teams zu aktivieren und somit einen Teamwechsel zu bewerkstelligen.

Alle ausführbereiten Prozesse eines Teams werden in einer teamlokalen Ready-Liste gehalten, die über den *Team Control Block* (TCB) des jeweiligen Teams referenziert wird. Der jeweils am Kopf dieser Liste stehende Prozeß gilt als der aktive Prozeß des Teams. Jedes Element dieser Liste entspricht dem *Process Control Block* (PCB) eines Prozesses. Bild 2.4 zeigt den prinzipiellen Aufbau dieser Liste.

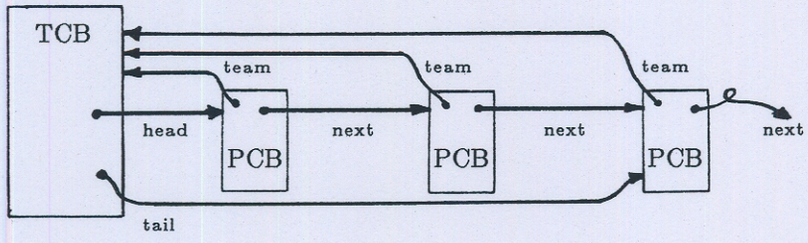

Bild 2.4: Verwaltung ausführbereiter Prozesse eines Teams

Je nachdem welches Attribut mit dem Prozeß verknüpft ist, wird er mit seiner De-Blockierung am Kopf oder am Ende der Liste eingetragen. Nur Interrupt-Prozesse werden am Kopf dieser Liste vermerkt, für alle anderen Prozesse gilt das first-come-first-serve Prinzip. Welche Liste hierzu verwendet wird, richtet sich nach der Zuordnung des Prozesses zu einem Team. Diese Zuordnung ist in jedem PCB durch die Referenz auf einen TCB definiert.

Alle Teams, die ausführbereit sind, d.h. in denen mindestens ein Prozeß in der teamlokalen Ready-Liste existiert, werden in einem *Dispatch-Set* zusammengefaßt. Dieser Dispatch-Set wird durch eine zyklische Verkettung von Teams, d.h. TCB's, repräsentiert. Mit jedem Teamwechsel wird das nächste Team des Dispatch-Set aktiviert. Die De-Blockierung eines Teams führt dann dazu, daß dieses Team vom Kernel als Vorgänger des aktiven Teams in den Dispatch-Set mit aufgenommen wird. Entsprechend wird bei seiner Blockierung das jeweilige Team aus dem Dispatch-Set herausgenommen. Bild 2.5 zeigt, wie der Dispatch-Set vom Kernel aus verwaltet wird.

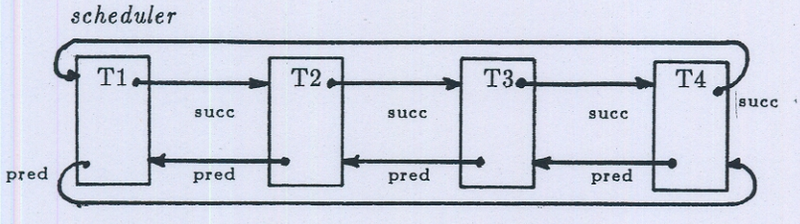

Bild 2.5: Dispatch-Set des Kernels

In diesem Beispiel wird insbesondere deutlich, wie komplexe Schedulestrategien in ein MOOSE-System integriert werden können. Der Scheduler selbst ist in dem Dispatch-Set als Team enthalten und wird somit zum gegebenen Zeitpunkt vom Kernel aktiviert. Diesem Schedule-Prozeß wird ermöglicht, den Dispatch-Set neu bestimmen zu können und die Kontrolle zur Abarbeitung des Dispatch-Sets an den Kernel zu übergeben. Der Umfang des Dispatch-Sets und die Position des Schedulers darin bestimmen, wann der Scheduler reaktiviert wird.

2.5.2. *Verarbeitung asynchroner Ereignisse*

Im Zusammenhang mit der Verwaltung von Geräten ist es notwendig, innerhalb fest vorgegebener Zeitspannen auf bestimmte Ereignisse reagieren zu können, wobei es oftmals bereits ausreicht, die Ereignisse nur aufzunehmen und nicht vollständig zu verarbeiten. Das Wesensmerkmal solcher Ereignisse ist, daß sie *asynchron* zu den restlichen Aktivitäten des Systems stattfinden. Dies erschwert in erhötem Maße die rechtzeitige Aufnahme und anschließende Verarbeitung der jeweiligen Ereignisse. Bild 2.6 skiziert grob diesen Sachverhalt.

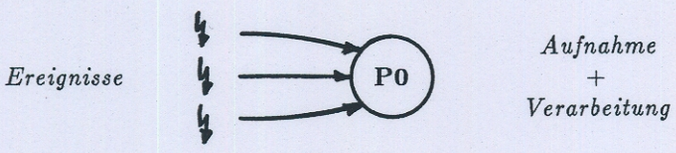

Bild 2.6: Gekoppelte Ereignisaufnahme und -verarbeitung

Die Realisierung eines Systems zur Ereignisaufnahme und -Verarbeitung kann erheblich dadurch unterstützt werden, daß an ein asynchron stattfindendes Ereignis jeweils ein bestimmter Prozeß gekoppelt wird (wie Bild 2.7 zeigt).

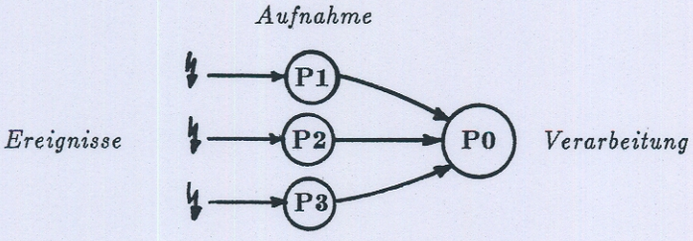

Bild 2.7: Entkoppelte Ereignisaufnahme und -verarbeitung

Der dem Ereignis zugeordnete Prozeß läuft *synchron* zu diesem Ereignis ab, womit folgendes erreicht wird:

a) die Ereignisaufnahme kann sehr schnell geschehen, da immer ein Prozeß dazu bereitsteht;

b) die Ereignisaufnahme ist von der Verarbeitung entkoppelt. Damit besteht für die eigentliche Verarbeitung (oftmals mehrerer Ereignisse) nicht mehr die Problemtik der Synchronisation zu den asynchron eintretenden Ereignissen.

Das Team-Modell ermöglicht hierbei eine effiziente Lösung. Aufnahmeprozeß und Verarbeitungsprozeß sind dem selben Team zugeordnet. Der Aufnahmeprozeß wird mit jedem eintretenden Ereignis gestartet und leitet eine entsprechende Behandlung ein. Statusinformationen bzw. Eingabedaten können dem Gerät entnommen und dem Verarbeitungsprozeß über einen globalen Datenbestand zugeführt werden. Entsprechendes gilt für die Ausgabe auf das Gerät. Jedem Ereignis kann ein Prozeß zugeordnet werden, der von seinem Verarbeitungsprozeß die für das Gerät bestimmten Informationen durch einen globalen Datenbestand annimmt. Die Kommunikationsprimitiven des MOOSE-Kernels würden in diesem Beispiel vorwiegend nur als Synchronisationsoperationen von den im Team existierenden Prozessen angewendet werden, um einen kontrollierten Zugriff auf den gemeinsamen Datenbestand zu ermöglichen.

2.5.3. *Nicht blockierende Inter-Prozeß Kommunikation*

Der Message-Passing Mechanismus des MOOSE-Kernels führt grundsätzlich zur Blockierung des sendenden bzw. empfangenden Prozesses, wenn zum Zeitpunkt der Kommunikation kein Rendezvous zwischen beiden an der Kommunikation beteiligten Prozessen möglich ist. Dies bedeutet, daß

a) der Sender blockiert, wenn der Empfänger nicht empfangsbereit ist, und

b) der Empfänger blockiert, wenn keine Nachricht zu ihm übermittelt worden ist, d.h. wenn kein Sender auf ihn blockiert ist.

Der wesentliche Vorteil dieses Verfahrens ist, daß eine elementare Realisierung angegeben werden kann, die zudem eine sehr effiziente Kommunikation zwischen den jeweiligen Prozessen ermöglicht. Der Kernel braucht keine Nachrichten zu speichern, da die Prozesse in jedem Fall ein Rendezvous eingehen müssen, um die Nachrichten zu senden bzw. zu empfangen.

Dennoch existieren eine Vielzahl von Applikationen, wo die Blockierung bei der Kommunikation zwischen Prozessen unerwünscht ist. Das Team-Modell von MOOSE stellt gerade im diesen Zusammenhang interessante Aspekte zur Verfügung. Durch geeignetem Aufbau eines Teams besteht die Möglichkeit, nicht blockierende Kommunikationsmechanismen zu schaffen. Bild 2.8 skizziert den dazu notwendigen Aufbau eines Teams, in dem Prozesse bei dem Empfang und während der Verarbeitung von Nachrichten nicht blockieren.

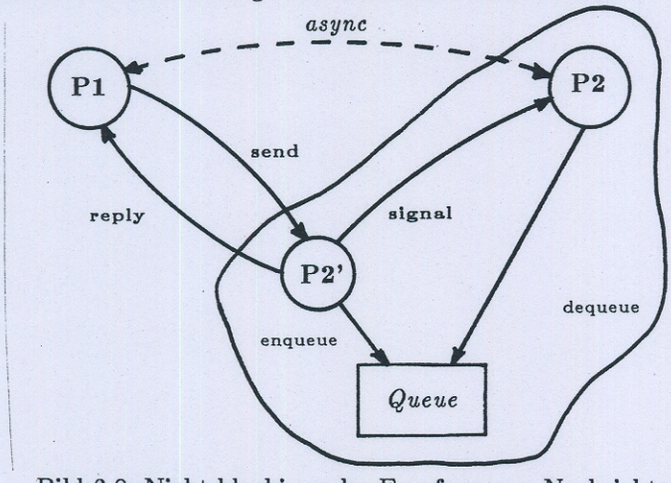

Bild 2.8: Nicht blockierender Empfang von Nachrichten

Prozeß **P1** will mit Prozeß **P2** nicht blockierend kommunizieren. Dazu wird **P2** als Team, bestehend aus zwei Prozessen, realisiert. Die Aufgabe des zusätzlichen Prozesses **P2'** innerhalb dieses Teams besteht nur darin, die von **P1** übermittelten Nachrichten abzunehmen, und zur späteren Verarbeitung (durch **P2**) zwischenzuspeichern. Dieses Verfahren entspricht der grundlegenden Semantik nicht blockierender Kommunikationsprimitiven. Der sendende Prozeß ist hierbei nur während der Annahme der Nachricht, nicht jedoch während ihrer gesamten Verarbeitung blockiert.

Ein anderer interessanter Aspekt ergibt sich aus der Anzahl der Kopiervorgänge, die notwendig sind, um die Nachrichten von **P1** nach **P2** zu übermitteln. Wie bei dem im Kernel realisierten Rendezvous-Konzept, brauchen die Nachrichten nur einmal kopiert zu werden. Da **P2'** immer empfangsbereit und zugleich im selben Team mit dem eigentlichen Verarbeitungsprozeß **P2** angesiedelt ist, stellt somit der Kernel bei dem Rendezvous zwischen **P1** und **P2'** die Daten, durch den gemeinsamen Adreßraum von **P2** und **P2'**, gleichzeitig **P2** zur Verfügung. Das lokale Protokoll zwischen **P2'** und **P2** kann davon profitieren und übergibt **P2** die Adresse der Nachricht, anstatt einen weiteren Kopiervorgang zu starten.

2.6. Multiprozessorapplikationen

Multiprozessorsysteme werden mit dem MOOSE-System in gleicher konzeptioneller Weise betrachtet wie es mit Netzwerkapplikationen der Fall ist. Server-Prozesse bestimmen die System-Topologie und bieten die nötige Unterstützung zur Verwaltung des jeweiligen Systems. Der wesentliche Unterschied zwischen Netzwerksystemen allgemein und Multiprozessorsystemen, der eine andere Verwaltungsstrategie motiviert, liegt in dem engen Bezug, dem die einzelnen Prozesse eines Multiprozessorsystems unterworfen sind. Ein bedeutendes Wesensmerkmal dieses Bezuges stellt gerade das *Shared Memory* Prinzip eines solchen Systems dar.

Jedem Prozessor eines Multiprozessorsystems ist der gleiche Kernel zugeordnet. Der Kernel selbst arbeitet jedoch nur auf lokalen Datenbeständen, die insbesondere die Kontrollstrukturen für die einzelnen Prozesse darstellen. Auf jedem System ist ein spezieller Administrator angesiedelt, der allgemein als *Multiprocessor Scheduler* bezeichnet werden kann. Diese Amdinistratoren stehen untereinander über ein Shared-Memory Modul in Verbindung, das u.a. Informationen enthält, die die jeweilige Konfiguration des Systems beschreiben. Bild 2.9 stellt eine solche Konfiguration vor.

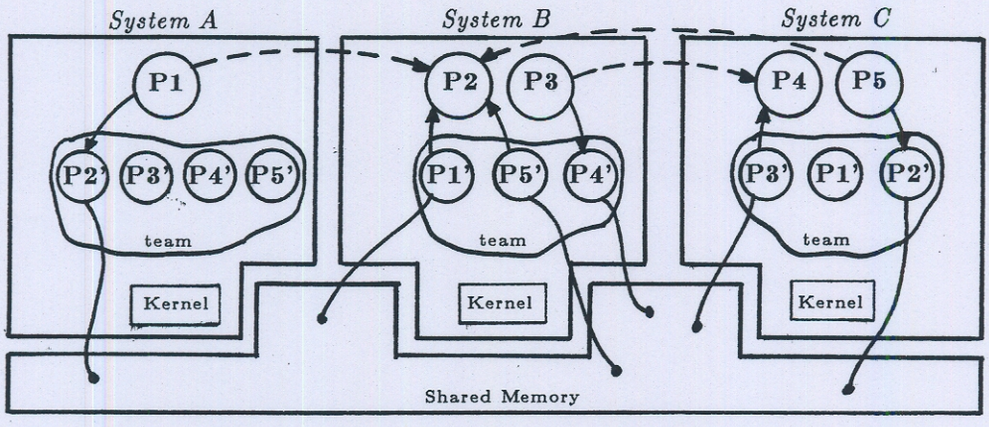

Bild 2.9: Multiprozessorsysteme unter MOOSE

Die Multiprocessor-Scheduler sorgen dafür, daß Prozesse aktiviert werden können, die anderen Prozessoren zugeordnet sind. Dies wird immer dann erfolgen müssen, wenn z.B. eine Kommunikation zwischen Prozessen unterschiedlicher Prozessorsysteme erfolgt.

Auf jedem Prozessor existiert ein Abbild des Prozesses, der auf einem anderen Prozessor angesiedelt ist und dort Dienste für eine bestimmte Applikation erbringt. Das Abbild wird durch einen Prozeß (jeweils **P1'**, **P2'**, **P3'**, **P4'** und **P5'**) innerhalb des Multiprocessor-Scheduler Teams realisiert. Dieser Prozeß bindet sich an den jeweiligen Eintrag des Prozesses in der Process-Map an, für den er das Abbild darstellt. Jeder Kommunikationswunsch mit einem in dieser Form abgebildeten Prozeß, z.B. ausgelöst durch **P1**, führt direkt zur Aktivierung des Multiprocessor-Schedulers, da ein Server-Prozeß (**P2'**) seines Teams reaktiviert wird. Dieser Server-Prozeß besitzt Informationen darüber, auf welchen anderen Prozessor ein Abbild des Initiators der Kommunikation (**P1'**) existiert. Damit ist gleichzeitig der entsprechende Multiprocessor-Scheduler identifiziert.

Da die Multiprocessor-Scheduler über ein Shared-Memory Modul miteinander in Verbindung stehen, können sie direkt auf Informationen zugreifen, die Aufschluß darüber geben, welcher Server-Prozeß innerhalb ihres Teams aktiviert werden soll. Diese Informationen werden von jedem Multiprocessor-Scheduler entsprechend aufbereitet. Nach Aktivierung eines Server-Prozesses, z.B. **P1'**, wird dieser die Kommunikation für den eigentlichen Initiator (**P1**) durchführen. Dies bedeutet, daß z.B. **P1'** für **P1** eine Nachricht, die sich im Shared-Memory Bereich befindet, zu **P2** sendet. Desweiteren wird **P1'** dafür sorgen, daß die durch **P2** als beendet markierte Kommunikation, **P1** durch **P2'** mitgeteilt wird.

3. ADMINISTRATOREN

Die Grundvoraussetzung bei der Konzeption des MOOSE-Systems besteht darin, daß die einzelnen Bausteine eines spezifischen Betriebssystems, in einer eindeutigen Benutztrelation zueinander stehen. Die sich daraus ergebende Hierarchie von Abstraktionsebenen zeichnet sich dadurch aus, daß die Komplexität der einzelnen Bausteine nach unten hin, d.h. zu den hierarchisch tiefer liegenden Ebenen, abnimmt. Mit Ausnahme des Kernels, werden alle Bausteine des MOOSE-Systems über Prozesse bzw. Teams realisiert und als System Administratoren bezeichnet.

3.1. Semantik der Dienstschnittstellen

Dienste der Administratoren werden von anderen Prozessen durch den Austausch von Nachrichten in Anspruch genommen. Aufgrund des Rendezvous-Modells hat dies zur Konsequenz, daß der anfordernde Prozeß blockiert. Wie lange dieser Prozeß blockiert bleibt, bestimmt der Administrator im Zusammenhang mit dem zu erbringenden Dienst. Grundsätzlich gilt jedoch für MOOSE, daß ein Prozeß erst dann von einem Administrator reaktiviert wird, wenn der zu erbringende Dienst erfüllt worden ist oder wenn der Administrator feststellt, daß der Dienst gegenwärtig bzw. generell nicht erbracht werden kann.

Der Grund für die blockierende Dienstschnittstelle der Administratoren ergibt sich aus der hierarchischen Anordnung, in der sich beide Prozesse, der Administrator und der anfordernde Prozeß, gegenüberstehen: der anfordernde Prozeß benutzt den Administrator.

3.2. Der Team-Administrator

Der Team-Administrator nimmt die unterste Ebene in der Hierarchie der System-Administratoren ein. Er sorgt allgemein für die Verwaltung von Prozessen bzw. Teams. Dazu zählt das Erzeugen und Zerstören von Prozessen, sowie die Verwaltung des für sie zur Verfügung stehenden Speicherplatzes.

Die Prozesse werden in bestimmte Beziehungen zueinander gebracht, sodaß mehrere Prozesse ein Team bilden und mehrere Teams eine Prozeßgruppe definieren können. Mit der Zerstörung von Prozessen oder Prozeßgruppen wird dafür gesorgt, daß alle von den betroffenden Prozessen belegten Resourcen dem System zurückgegeben werden.

3.2.1. *Prozeßmodelle*

Der Ausgangspunkt bei der Modellierung von Prozeßbeziehungen stellt das Team-Konzept dar. Jeder Prozeß ist während seiner Existenz statisch genau einem Team zugeordnet[8)].

Bei der Erzeugung von Prozessen wird unterschieden, ob der neue Prozeß einem bestehenden (dem erzeugenden) Team zugeordnet wird, oder ob dieser Prozeß ein neues Team eröffnet. Für den letzteren Fall bedeutet die Erzeugung das Einrichten einer vollständigen Kopie des erzeugenden Teams. Diese Variante entspricht dem *fork* -Konzept, wie es in UNIX seine Anwendung findet. Der so erzeugte Prozeß wird als Supervisor des entsprechenden (seines) Teams

[8)] Eine Ausnahme findet hierbei im Zusammenhang mit der Behandlung von Interrupts nach dem Interrupt-Link Modell statt. Zur Behandlung des Interrupts wird dabei kein Prozeßwechsel, sondern nur ein Adreßraumwechsel vom Team des unterbrochenen Prozesses zum Team des Interrupt-Handlers vollzogen. Das hat zur Konsequenz – da der Interrupt-Handler unter Kontrolle des unterbrochenen Prozesses läuft – daß dieser Prozeß während der Dauer der Interrupt-Behandlung einem anderen Team zugeordnet werden muß.

bezeichnet.

Um ein Team aufbauen zu können, müssen neue Prozesse einem Supervisor-Prozeß zugeordnet werden. Diese Prozesse und der Supervisor besitzen den gemeinsamen Zugriff auf den dem Team zugeordneten Adreßraum und sind somit in der Lage dieselben Daten- wie auch Code-Segmente zu referenzieren. Prozesse innerhalb eines Teams unterscheiden sich hinsichtlich ihrer Zugriffsrechte auf Objekte des Teams dadurch, ob ihre Stack-Segmente anderen Prozessen zugänglich oder vor Zugriffen durch andere Prozesse geschützt sind.

Soll das Stack-Segment eines Prozesses geschützt werden, so wird ihm bei seiner Erzeugung eine vollständige Kopie des Stack-Segmentes des erzeugenden Prozesses zugeordnet. Dies enstpricht dem Erzeugen eines neuen Teams, wobei jedoch der Datenbereich nicht dupliziert wird.

Bei beiden Varianten der Prozeßerzeugung gilt jedoch, daß alle logischen Adreßbezüge der entsprechenden Segmente aufrechterhalten werden müssen. Daraus ergibt sich die Einschränkung, daß ein solches Prozeßmodell nur durch Unterstützung entsprechender Hardware, etwa einer *Memory Management Unit* (MMU), realisiert werden kann. Applikationen, deren kontrollierende Prozesse ausgiebig von diesem Modell Gebrauch machen, grenzen somit ihr Einsatzgebiet dadurch ein, daß sie eine bestimmte Klasse von Hardware-Systemen voraussetzen. Andererseits ist ein solcher Schutzmechanismus höchst wünschenswert für die Applikationen, wie auch für das Betriebssystem selbst.

Ein weiterer Vorteil des eben dargestellten Modells liegt darin, daß dem neuen Prozeß der Laufzeitkontext des erzeugenden Prozesses übertragen wird. Dieser Laufzeitkontext wird zum Zeitpunkt der Erzeugung des neuen Prozesses fixiert. Mit diesem Modell können Prozesse erzeugt werden, deren einzige Aufgabe z.B. darin besteht, den aktuellen Zustand einer bestimmten Applikation (die durch den erzeugenden Prozeß gesteuert wird) zu sichern und damit *Check Points* für diese Applikationen abzusetzen.

Soll bzw. kann – aufgrund der nicht vorhandenen Unterstützung spezieller Hardware – das Stack-Segment eines Prozesses nicht geschützt werden, so wird ihm ein neuer Laufzeitkontext zugeordnet. Zum Zeitpunkt der Erzeugung des entsprechenden Prozesses, wird eine Prozedur angegeben, die bei Aktivierung des Prozesses gestartet wird. Der Prozeß baut somit seinen Laufzeitkontext neu auf, wenn mit der Ausführung der Prozedur begonnen wird.

Diese Form der Erzeugung eines Prozesses ermöglicht somit auch Prozeßmodelle auf Prozessoren, die keine spezielle Harware-Unterstützung anbieten. Bei einem Prozessor etwa, der weder über Segment-Register, noch über eine MMU verfügt, wird der gesamte lineare (reale) Adreßraum einem Team zur Verfügung gestellt und somit für dieses Team generell ein logischer Adreßbereich definiert. In diesem Team können dann mehrere Prozesse existieren, mit der Einschränkung, daß sie sich ihren Laufzeitkontext selbst aufbauen müssen. Die Stack-Segmente liegen somit sämtlichst an unterschiedlichen logischen Adressen innerhalb des Teams.

3.2.2. *Speichermodelle*

Generell wird durch das Prozeßmodell bereits ein Speichermodell vorgegeben, indem innerhalb eines logischen Adreßraumes verschiedene Objekte (Stack-Segmente) zu verwalten sind. Da Prozesse während ihrer Laufzeit Speicher vom System anfordern bzw. dem System zurückgeben können, erweitert sich die Verwaltung des logischen Adreßraumes eines Teams dahingehend, daß verschiedene Typen von Objekten (Stack- und Daten-Segmente) erfaßt werden müssen.

Die Daten-Segmente sind jeweils nur dem eigenen Team zugänglich. Dennoch besteht die Möglichkeit, unterschiedlichen Teams direkten Zugriff auf gemeinsame Daten-Segmente zu verschaffen. Dies bedeutet, daß zwischen Teams Shared Memory ermöglicht wird. Die jeweiligen Shared-Segmente werden durch entsprechende Objekte verwaltet, über die ein kontrollierter Zugriff ermöglicht wird.

Um den gemeinsamen Zugriff auf Shared-Segmente zu ermöglichen, ist in jedem Fall die Anforderung an unterstützende Hardware zu stellen. Shared-Memory ist deshalb kein festes

Bestandteil des MOOSE-Systems und wird in letzter Konsequenz auch nicht vom Team-Administrator realisiert. Der Team-Administrator stellt hierzu lediglich die entsprechenden Speicher-Bereiche für die jeweiligen Teams zur Verfügung. Der Mechanismus des Shared-Memory Prinzips wird durch einen eigenen Administrator realisiert.

3.2.3. *Behandlung von Ausnahmesituationen*

Die von dem Team-Administrator zu behandelnden Ausnahmesituationen beziehen sich auf Traps und andere systemspezifische Ereignisse. Traps werden i.A. bei der Ausführung von Prozessen erzeugt und können den Abbruch dieser Prozesse zur Folge haben. Systemspezifische Ausnahmesituationen stellen z.B. das Absetzen von Signalen über das dem Prozeß zugeordnete Terminal dar.

Der Abbruch der Prozesse richtet sich nach der Klasse der aufgetretenden Ausnahmesituation, d.h. es richtet sich danach, ob eine Behandlung dahingehend möglich ist, daß die Ausführung des betroffenden Prozesses wieder aufgenommen werden kann. Sogenannte *Page-Faults* führen i.A. nicht zur Termination des den Trap auslösenden Prozesses, wohingegen das Absetzen privilegierter Instruktionen den Abbruch des ausführenden Prozesses mit sich bringt.

Prozessen wird prinzipiell die Möglichkeit zur Verfügung gestellt, Ausnahmesituationen selbst behandeln zu können. Dazu wird dem behandelnden Prozeß innerhalb seines Teams auf Anforderung ein weiterer Prozeß zugeordnet, der nur für die Annahme der Ausnahmesituation zuständig ist. Dieser Prozeß ist auf den Team-Administrator blockiert und wird dann aktiviert, wenn die Ausnahmesituation eingetreten ist. Der Team-Administrator entscheidet, ob die Behandlung der Ausnahmesituation dem auslösenden Team weitergereicht werden kann. Bild 3.1 stellt das Modell der Behandlung von Ausnahmesituationen durch Prozesse dar.

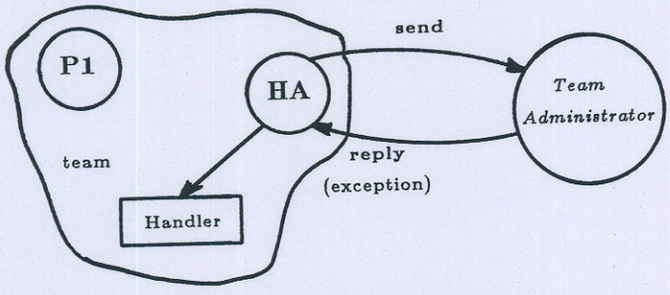

Bild 3.1: Benutzerspezifische Behandlung von Ausnahmesituationen

Der im Zuge der benutzerspezifischen Behandlung von Ausnahmesituationen von dem Prozeß **P1** zusätzlich eingerichtete Prozeß, wird als *Handler-Administrator* (**HA**) bezeichnet. Der Team-Administrator übermittelt dem Handler-Administrator eine Identifikation der aufgetretenden Ausnahmesituation. Dieser entscheidet sodann, welche besonderen Maßnahmen innerhalb des behandelnden Teams zu treffen sind. Üblicherweise wird zu diesem Zeitpunkt eine spezielle Behandlungsprozedur ("Exception-Handler") des Teams vom Handler-Administrator gestartet. Mit Termination dieser Prozedur ist der Handler-Administrator wieder bereit weitere Ausnahmesituationen anzunehmen und ihre Behandlung einzuleiten.

3.3. Administratoren zur Geräteverwaltung

Das Modell in MOOSE sieht prinzipiell vor, für jede Klasse von Geräten mindestens einen eigenen *Device-Administrator* vorzusehen. An unterschiedliche Klassen werden hierbei zeichenorientierte-, blockorientierte-, und frameorientierte Geräte betrachtet.

3.3.1. *Problematik blockierender Dienstschnittstellen*

Eine besondere Problematik entsteht durch die blockierend wirkende Dienstschnittstelle der Device-Administratoren. Diese Administratoren sind von externen Ereignisse abhängig, die zur Erbringung bestimmter Dienste eintreten müssen. Diese Ereignisse werden durch Interrupts erzeugt, die die Durchführung von Ein-/Ausgabeoperationen für einen anfordernden Prozeß als beendet markieren. Dies bedeutet, daß diese Administratoren asynchrone Operationen, gesteuert durch Interrupts, auf synchrone Operationen, angesteuert durch die Dienstschnittstelle, abbilden müssen. Das Rendezvous zwischen anfordernden Prozeß und Administrator dauert demzufolge solange an, wie die zur Erbringung des jeweiligen Dienstes notwendigen asynchronen Operationen noch nicht beendet sind.

Da die Device-Administratoren sequentiell ablaufende Prozesse darstellen, würde das Warten auf die externen Ereignisse generell alle Prozesse (und nicht nur den gegenwärtig anfordernden Prozeß) blockieren. Dies bedeutet, daß auch Prozesse blockiert werden, ohne eine Information darüber zu besitzen, welchen Dienst sie von dem Administrator in Anspruch nehmen wollen. Nicht jeder Dienst wird jedoch das Erwarten von Interrupts mit sich bringen, und andererseits muß es möglich sein, daß Prozesse, die z.B. unterschiedlichen zeichenorientierten Geräten zugeordnet sind, auch "gleichzeitig" Ein-/Ausgabeoperationen initiieren können.

Demzufolge darf die Blockierung nur für den direkt anfordernden Prozeß gelten und nicht für Prozesse, die zeitlich nach dem gegenwärtig anfordernden Prozeß ihre Dienstanforderung stellen. Um dies zu bewerkstelligen, bieten sich generell zwei Lösungsmöglichkeiten an.

Die eine Variante zeichnet sich durch eine konsequente Anwendung des Team-Konzepts aus. Da die asynchronen Ereignisse immer mit bestimmten Geräten verknüpft sind und die Ein-/Ausgabeoperationen im Zusammenhang mit einem Gerät sequentiell stattfinden, wird jedem Gerät ein Prozeß innerhalb des Administrators zugeordnet. Dieser Prozeß ist auf das Eintreten der jeweiligen Ereignisse synchronisiert. Die Anforderungen der Prozesse werden dann dahingehend klassifiziert, ob sie auf einen Prozeß blockierend oder nicht blockierend wirken. Die nicht blockierenden Anforderungen werden durch den Supervisor des Device-Administrators abgearbeitet, wohingegen die Geräteprozesse die blockierenden Anforderung direkt durch den anfordernden Prozeß oder indirekt durch den Supervisor zugestellt bekommen. Die Geräteprozesse arbeiten dann die jeweiligen Anforderung sequentiell ab und reaktivieren zum gegebenen Zeitpunkt den anfordernden Prozeß. Bild 3.2 skizziert einen solchen Device-Administrator.

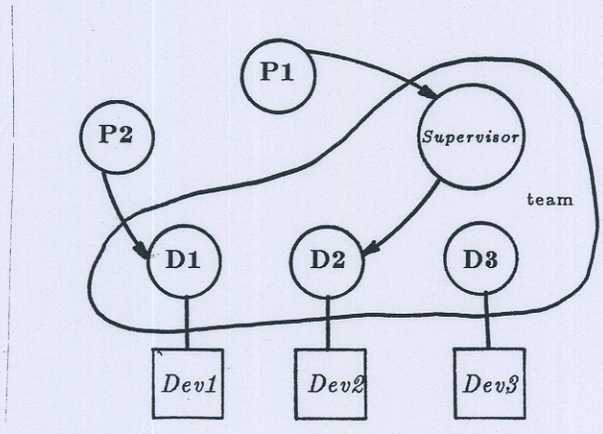

Bild 3.2: Administrator mit verschiedenen Geräteprozessen

Die andere Variante vermeidet die Notwendigkeit von Geräteprozessen dann, wenn dem Device-Administrator eine Verwaltungskomponente zugeordnet wird, die alle noch in Arbeit befindlichen Anforderungen der Prozesse verzeichnet. Der Device-Administrator ist nach Vermerk einer solchen Anforderung bereit, weitere Anforderungen entgegenzunehmen. Sobald der Dienst

erbracht werden konnte, wird der jeweils anfordernde und wartende Prozeß reaktiviert.

Die Verwaltungskomponente stellt einen Zustandsautomaten dar, dessen Semantik darin besteht, mehrere "nebenläufige" Aktivitäten innerhalb eines Teams durch einen Prozeß steuern zu können. Obgleich mit dieser Variante nur ein Prozeß benötigt wird, um mehrere Anforderung "gleichzeitig" bearbeiten zu können, ist ihr dies nicht unbedingt zum Vorteil gegenüber der Variante mit den einzelnen Geräteprozessen auszulegen. Zustandsautomaten sind mit der Problematik behaftet, schnell an Komplexität zu gewinnen und damit komplizierter und fehleranfälliger zu werden, je mehr Zustände zu betrachten sind. Die Zustände ergeben sich konsequenterweise aus der Anzahl der Geräte, denen jeweils eine bestimmte Menge von Prozessen zugeordnet werden können, wobei die Prozesse wiederum unterschiedliche Anforderungen in Verbindung mit den Geräten stellen.

3.3.2. *Behandlung von Interrupts*

In jedem Device-Administrator ist generell ein Server-Prozeß angesiedelt, der zur Kommunikation mit dem Interrupt-Handler bereitsteht. Ob dieser Interrupt-Server den einzigen Prozeß des Teams darstellt oder ob noch andere Prozesse in dem Team angesiedelt sind (z.B. die Geräteprozesse), ist für die Behandlung der Interrupts unerheblich. In jedem Fall ist der Interrupt-Server derjenige Prozeß, der die Anbindung an die Interrupt-Vektoren durchführt, die im Zusammenhang mit der Verwaltung von bestimmten Geräten betrachtet werden müssen. Diese Maßnahme findet unter Kontrolle des Kernels statt, sodaß mit der Anbindung gleichzeitig die jeweilige Strategie zur Interrupt-Behandlung dem Kernel bekannt gegeben werden kann.

Zur Behandlung der Interrupts stehen den Device-Administratoren grundsätzlich zwei Modelle zur Verfügung: Interrupt-Links und Interrupt-Tasks. Beide Modelle (in Bild 3.3 skizziert) gestatten für den Interrupt-Handler die gleiche Art der Geräteverwaltung.

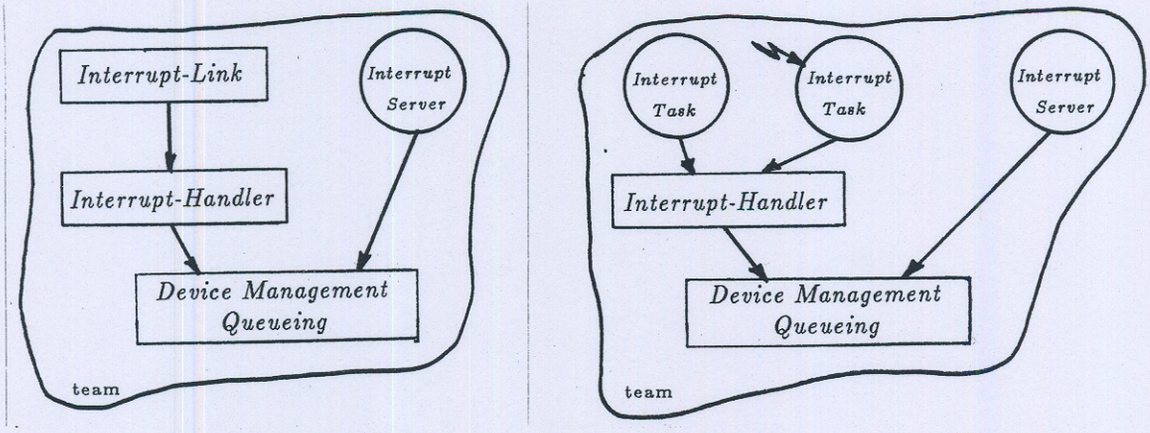

Bild 3.3: Modelle der Interrupt-Behandlung

Die Interrupt-Handler werden prozedural aktiviert und erhalten dabei die Identifikation des Gerätes, das den Interrupt produziert hat. Diese Identifikation ergibt sich anhand des Interrupt-Vektors, an dem das Gerät physikalisch angeschlossen und mit dem ein bestimmter Interrupt-Link oder eine bestimmte Interrupt-Task assoziiert ist. Üblicherweise wird diese Identifikation die Adresse der jeweils dem Gerät zugeordneten kontrollierenden Datenstruktur darstellen und vom Interrupt-Link bzw. von der Interrupt-Task erzeugt. Terminiert der Interrupt-Handler, d.h. verläßt er seine Inkarnationsstufe, so übergibt er der aufrufenden Instanz (Interrupt-Link oder Interrupt-Task) die Information darüber, ob eine Kommunikation mit seinem Interrupt-Server stattfinden soll. Über dieses Verfahren können beide, Interrupt-Handler und Interrupt-Server, den Kontroll- und Datenfluß von/zum Gerät steuern.

3.4. Administratoren zur Fileverwaltung

Die Grundkonzeption des MOOSE-Filesystems basiert auf einer hierarchisch ausgelegten Struktur zur Verwaltung der Datenbestände, wie es das UNIX-Filesystem kennt. Der wesentliche Unterschied zum UNIX-System liegt hierbei darin, daß aufgrund der prozeßorientierten Struktur von MOOSE die Integration neuer Filesysteme auf die Identifikation der für diese Systeme zuständigen Administratoren zurückgeführt wird. Dies gilt insbesondere auch für Geräte, die in MOOSE wie in UNIX durch Files repräsentiert werden. Das Eröffnen eines solchen *special device files* durch den Filesystem-Administrator führt zur Identifikation des enstprechenden Administrators.

Der Filesystem-Administrator führt selbst nur Ein-/Ausgabeoperationen auf den ihm bekannten blockorientierten Geräten aus [9]. Alle anderen Ein-/Ausgabevorgänge, die nicht mit diesen Geräten verbunden sind, werden direkt ihren zuständigen Administratoren zugeleitet. Diese Administratoren können insbesondere selbst wieder Filesystem-Administratoren darstellen, die eine andere (z.B. keine hierarchische) Struktur des Filesystems auf dem jeweiligen blockorientierten Gerät implementieren. Ebenso können diese Administratoren einen remote device access Mechanismus ermöglichen und somit ein dezentral organisiertes Filesystem oder die Realisierung eines verteilten Filesystems unterstützen.

Die Identifikation der jeweiligen Administratoren zur Durchführung von Ein-/Ausgabevorgängen wird durch Bibliotheksfunktionen ermöglicht. Damit kommunizieren Prozesse direkt mit den Administratoren, die die jeweils angeforderten Dienste zur Verfügung stellen.

3.5. Administratoren zur Verwaltung von Applikationen

Der Aufbau neuer Applikationen wird nicht allein dadurch ermöglicht, daß neue Prozesse erzeugt werden. Vielmehr ist dafür Sorge zu tragen, daß Prozesse andere Programme ausführen können. Dies erreichen die Prozesse, indem sie ihren Adreßraum mit dem neuen Programm überlagern. Die Kombination der Erzeugung eines Prozesses und der anschließenden Überlagerung seines Adreßraums, stellt der grundlegende Mechanismus in MOOSE dar, neue Applikationssysteme zu erstellen.

3.5.1. *Betriebssystem-Domänen*

Applikationen sind bestimmten *Domänen* zugeordnet. Jede Domäne definiert eine bestimmte Menge von Diensten, die von den Applikationen benutzt werden können.

Im Sinne des MOOSE-Konzepts, steht eine Domäne stellvertretend für ein bestimmtes Betriebssystem der MOOSE-Familie. Die Realisierung einer bestimmten Betriebssystem-Domäne kann in MOOSE durch die Integration von Administratoren erfolgen, die dann die jeweiligen Dienste der so geschaffenen Domäne zur Verfügung stellen.

Eine besondere Position nehmen in diesem Zusammenhang *Emulatoren* ein. Diese Administratoren sorgen für eine Abbildung von Diensten einer bestimmten Domäne (z.B. UNIX) auf Dienste einer anderen Domäne (z.B. MOOSE). Sie finden ihre Anwendung überall dort, wo Programmsysteme direkt, d.h. ohne diese Systeme neu zu generieren und somit an eine bestimmte Domäne zu adaptieren [10], zur Ausführung gebracht werden müssen.

---

[9] Der Filesystem-Administrator benötigt mindestens den Zugriff auf ein blockorientiertes Gerät um die Identifikation anderer Administratoren zur Geräteverwaltung ermöglichen zu können. Auf diesem Gerät befindet sich das sogenannte Root Filesystem, in dem die jeweiligen "special device files" verzeichnet sind.

[10] Die Adaption von Programmsystemen an neue Betriebssystem-Domänen wird in erster Linie dadurch erfolgen, daß entsprechende Bibliotheken, sogenannte *compatibility libraries*, zur Verfügung gestellt werden.

Die Emulatoren binden sich an die Trap-Vektoren an, die von den jeweiligen Applikationen benutzt werden um die Dienste ihrer (zu emulierenden) Domäne in Anspruch zu nehmen. Der MOOSE-Kernel stellt hierzu die entsprechenden Mechanismen zur Verfügung und sorgt dafür, daß, mit Auslösen eines bestimmten Emulator Traps, dem Emulator eine Trap-Nachricht übermittelt wird.

3.5.2. *Erzeugung von Applikationen*

Die Erzeugung von Applikationen wird auf technischer Ebene dadurch unterstützt, daß Teams ihren Adreßraum mit einem neuem Programm überlagern können. Dazu wird in erster Linie ein *Loader* benötigt, der das Ladeformat für dieses Programm entsprechend interpretieren kann. Gerade im Zusammenhang mit der Emulation von Systemdiensten wird es notwendig sein, für Betriebssystem-Domänen jeweils eigene Loader zur Verfügung zu stellen. Diese domänspezifischen Loader interpretieren direkt das jeweilige Ladeformat eines Programms und initiieren die jeweilige Adreßraumüberlagerung des entsprechenden Teams. Die Adreßraumüberlagerung wird effektiv vom Team-Administrator durchgeführt.

Der Loader einer MOOSE-Domäne sorgt dafür, daß Prozesse ihren Adreßraum mit einem neuen Image überlagern können. Hierbei wird unterschieden, ob die Überlagerung nur das Code-Segment, das Daten-Segment oder beide Segmente betrifft. Insbesondere wird durch den Loader kontrolliert, welche Segmente bereits geladen sind und ob diese bei der Überlagerung eines Prozesses "geshared" werden können.

Ein wesentlicher Aspekt bei der Erzeugung von Applikationen, die aus mehreren Prozessen aufgebaut sind, besteht darin, die Initialisierungsreihenfolge der einzelnen Prozesse festlegen zu können. Diese Aufgabe übernimmt ein *Sequencer* , indem er eine Spezifikation interpretiert. die der jeweils zu erzeugenden Applikation zugeordnet ist. Die Spezifikation definiert, wann ein bestimmter Prozeß der jeweiligen Applikation erzeugt werden muß und von welchen anderen Prozessen dieser Prozeß abhängig ist. Somit wird sichergestellt, daß, bevor ein bestimmter Prozeß aktiviert wird, die jeweils notwendige Umgebung (z.B. Emulatoren, bestimmte Administratoren zur Ein-/Ausgabe von Daten, etc.) für einen Prozeß definiert ist.

3.5.3. *Hierarchien von Betriebssystem-Domänen*

Die Anbindung an die Trap-Vektoren (um die Emulation bestimmter Systemdienste zu ermöglichen) führt dann zu einem Problem, wenn verschiedene Betriebssystem-Domänen den selben Vektor benutzen um bestimmte Dienste zur Verfügung zu stellen, diese Dienste jedoch mit unterschiedlicher Semantik behaftet sind. In diesem Fall sorgt ein *Domain-Switch* für die entsprechende Zuordnung von Dienst (Emulator-Trap) zu Domäne (der jeweilige Emulator). Bild 3.4 skizziert einen solchen Aufbau, der verschiedene Betriebssystem-Domänen enthält. Die zu emulierenden Dienste stellen in diesem Beispiel QNX [11)] und UNIX dar. Der Domain-Switch verteilt die jeweils zu emulierenden Dienste an die entsprechenden Emulatoren. Dazu besitzt der Domain-Switch Informationen darüber, welcher Domäne ein bestimmter Prozeß zugeordnet ist.

Je stärker sich die Dienste der einzelnen Domänen von den Diensten der MOOSE-Domäne unterscheiden, um so mehr Administratoren würden in den jeweiligen Domänen angesiedelt sein, um die zusätzlichen Dienste zu erbringen. Die QNX-Domäne stellt solch einen Fall dar. Da Loader und Filesystem der MOOSE-Domäne mit denen der UNIX-Domäne identisch sind, d.h. den selben Dienst erbringen, muß die QNX-Domäne (im Gegensatz zur UNIX-Domäne) einen eigenen Loader und Filesystem-Administrator enthalten.

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

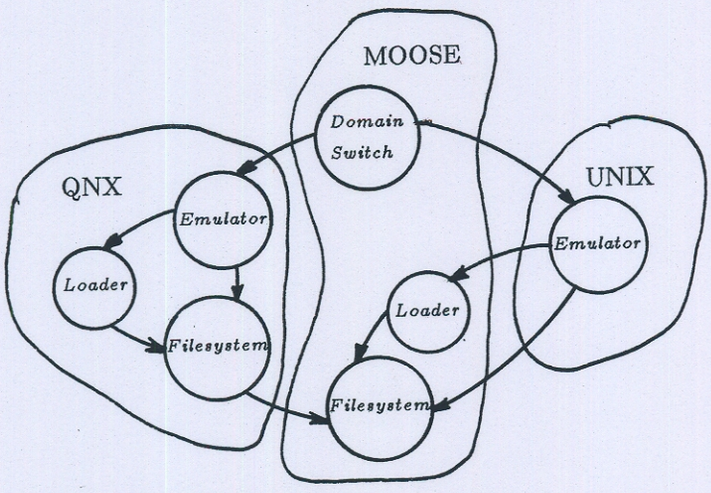

Bild 3.4: Hierarchien von Betriebssystem-Domänen

3.6. Administratoren zum Scheduling

Der Kernel des MOOSE-Systems implementiert eine nur sehr elementare Schedule-Strategie. Er verarbeitet den Disptach-Set streng nach dem round-robbin Prinzip. Für komplexere Mechanismen zum Scheduling wird davon ausgegangen, daß ein geeigneter Server-Prozeß im System vorhanden ist. Der Scheduler stellt solch einen Server-Prozeß dar, der vom Kernel zu bestimmten Zeitpunkten gestartet wird. Diesen Zeitpunkt kann der Scheduler selbst bestimmen, indem er bei jeder Reaktivierung den Dispatch-Set für den Kernel neu aufbaut.

Nach der durch den Scheduler implementierten Strategie nimmt er ausführbereite Teams in den neuen Dispatch-Set mit auf und sorgt dafür, daß er selbst als letztes Team darin enthalten ist. Die Art und Weise wie der Kernel Prozesse bzw. Teams "dispatched", garantiert die Aktivierung des Schedulers dann, wenn alle Teams seit dem letzten Re-Schedule abgearbeitet worden sind.

Neben dem Scheduling kann, je nach Konfiguration des MOOSE-Systems, in dem Schedule-Administrator ein *Swapper* enthalten sein, d.h. Scheduler und Swapper bilden zusammen ein Team. Ob beide Funktionalitäten durch einen oder durch getrennte Prozesse erbracht werden, ist für das System unerheblich. Wenn es sich ergeben sollte, daß kein Prozeß in dem System mehr aktiv ist, da insbesondere Scheduler und Swapper durch einen Prozeß kontrolliert werden und der Swapper in diesem Team zuletzt aktiv gewesen ist, führt der Kernel selbst eine *Idle-Phase* durch. Zu diesem Zeitpunkt können nur Interrupts dafür sorgen, daß Prozesse und somit Teams wieder aktiv werden.

Zur allgemeinen Verwaltung der Systemzeit, aber auch zur Unterstützung zeitgesteuerter Applikationen, dient der *Clock-Administrator* . Er steht in direkter Beziehung zu der im Kernel angesiedelten Realtime Clock. Diese Beziehung wird aufgebaut, indem sich der Clock-Administrator an das Clock-Module des Kernels anbindet. In diesem Zusammenhang wird dem Kernel der Port des Clock-Administrators mitgeteilt, an dem mit jedem "clock-tick" ein Signal abgesetzt wird.

Das im Kernel angesiedelte Clock-Modul ist notwendig, um in zyklischen Abständen den Dispatcher des Kernels aktivieren zu können und somit die Abarbeitung des Dispatch-Set zu gewährleisten. Die Abstände, in denen das Clock-Modul Signale abgibt, sind mit dem Clock-Administrator abgestimmt, bzw. können für bestimmte Applikationen explizit bestimmt werden.

4. REVIEW

Die Administratoren des MOOSE-Systems stehen in einer eindeutigen Benutztrelation [Parnas 1976] zueinander. Jeder Administrator definiert eine bestimmte Abstraktionsebene. Er kann Dienste hierachisch tiefer angeordneter Adminstratoren in Anspruch nehmen, um selbst Dienste für hierarchisch höher angeordneter Administratoren bzw. Benutzer-Prozesse zur Verfügung zu stellen. Die Hierachie der hier vorgestellten Administatoren ist in Bild 4.1 dargestellt.

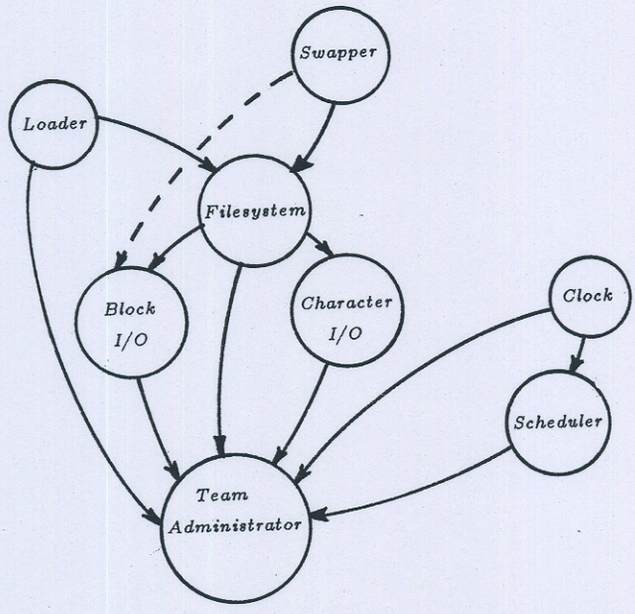

Bild 4.1: Hierarchie von System-Administratoren

In dieser Skizze ist, der Übersichtlichkeit halber, der Kernel und der Service-Administrator nicht enhalten. Der Kernel wird von allen Prozessen (und damit auch von allen Administratoren) benutzt, um die Kommunikation untereinander zu ermöglichen. Der Service-Administrator wird nur von den Prozessen in Anspruch genommen, die in keinem Vater-Sohn Verhältnis (eingerichtet durch die Erzeugung eines neuen Prozesses) zueinander stehen. Dies ist der Fall bei all den hier vorgestellten System-Administatoren.

Für ein prozeßorientiertes Betriebssystem ergibt sich, im Hinblick auf eine strenge Benutztrelation zwischen den einzelnen Komponenten, eine wesentliche Eigenschaft, die bei prozedurorientierten Systemen nicht in dem Maße hervortritt:

a) die in der Hierarchie tiefer angeordneten Prozesse sind unabhängig von Prozessen, die höheren Abtsraktionsebenen zugeordnet sind.

b) Prozesse besitzen eine allgemeine Dynamik dadurch, daß sie erzeugt und zerstört werden können.

Mit diesen beiden Voraussetzungen ist somit ein elementarer Ansatz gegeben, bestehende Betriebssysteme zur Laufzeit rekonfigurieren, zumindest jedoch, durch hinzunahme weiterer Bausteine, funktional anreichern zu können. Entsprechendes Vorgehen bei prozedurorientierten Betriebssystemen ist nur mit Hilfe unterstützender Hardware möglich.

Der Team-Administrator nimmt die unterste Ebene der Hierarchie der Administratoren ein. Durch ihn (und auf Grundlage des Service-Administrators und der Process-Map des Kernels) wird ein wesentlicher Abstraktionsschritt von dem zugrunde liegenden Prozessor erbracht. Alle hierarchisch höher angeordneten Administratoren, können somit von der jeweiligen System-Topologie abstrahieren.

Die Abhängigkeit zwischen dem Clock-Administrator und dem Scheduler ergibt sich dann, wenn Prozesse für eine bestimmte Zeit vom Scheduling ausgeschlossen werden sollen. Der Scheduler ist vom Team-Administrator abhängig, da er innerhalb seines Teams evtl. weitere Prozesse mit aufnehmen wird (z.B. durch die direkte Integration einer Swap-Komponente), zumindestens aber für seine Listen-Verwaltung dynamischen Speicherplatz vom Team-Administrator anfordert.

Die Device-Administratoren für die zeichen- und blockorientierte Ein-/Ausgabe ergeben sich aufgrund des in MOOSE verwirklichten File-System Konzeptes und aufgrund der Tatsache, daß Benutzerprozesse sicherlich bestimmte Ein-/Ausgabeoperationen absetzen werden. Die Geräte werden vom Filesystem-Administrator als *special device files* verwaltet. Jeder erstmalige bzw. letztmalige Zugriff auf die Geräte führt zu einer Benachrichtigung des zugeordneten Device-Administrators.

Der Loader und der Swapper benutzen das Filesystem, um neue Applikationen aufzubauen bzw. um Prozesse auf dem *Swap-Device* abzulegen und von diesem wieder in den Hauptspeicher zu transferieren.

Unabhängig von den hier vorgestellten Administratoren ist jederzeit die Möglichkeit gegeben, anwendungsspezifische Systeme aufzubauen. Je nach erwarteten Dienstleistungen für ein solches System kann eine Auswahl der notwendigen Amdministratoren getroffen werden. Ebenso können weitere, systemspezifische Administratoren integriert werden und z.B. die Rolle spezieller Device-Administratoren (etwa Netzwerk-Server) übernehmen. Der Kernel und der Service-Administrator des MOOSE-Systems stellen hierzu Mechanismen zur Verfügung, um eine transparente Kommunikation zwischen Prozessen zu ermöglichen.

Literaturverzeichnis

[Cheriton 1982]
D. R. Cheriton: **The Thoth System**, Elservier Science Publishing Co, 1982

[Crowley 1981]
C. Crowley: **The Design of a New UNIX Kernel**, AFIPS, 1981

[Hansen 1973]
P. Brinch Hansen: **Operating System Principles**, Prentice-Hall, 1973

[Hoare 1974]
C. A. R. Hoare: **Monitors: an operating system structuring concept**, Comm. ACM, 17, 7, 549-557, 1974

[Organick 1972]
E. Organick: **The Multics System: An Examination of its Structure**, MIT Press, 1972

[Parnas 1976]
D. L. Parnas: **Some Hypotheses about the 'uses' Hierarchie for Operating Systems**, Report, TH Darmstadt, 1976

[Schindler 1980]
S. Schindler: **Distributed Abstract Maschine**, Computer Communications, Vol. 3, No. 5, 1980

[Thompson, Ritchie 1974]
K. Thompson, D. M. Ritchie: **The UNIX Timesharing System**, Comm. ACM, 17, 7, 365-375, 1974