GNU - Hurd
ein Mach basiertes Multiserver Betriebssystem

Manuel Gorius

8. Oktober 2004

Inhaltsverzeichnis

1 Wissenswertes über Hurd

GNU/Hurd ist ein auf Mach Microkernel aufbauendes Multiserver Betriebssystem. Hurd wurde von der Free Software Foundation (FSF) als Unix-Ersatz entwickelt. Es besteht ausschließlich aus freier Software. Schon im Namen wird auf die Architektur des Betriebssystems in Multiserver Microkernel Organisation mit Hilfe zweier gegenseitig rekursiver Akronyme hingewiesen: "Hurd" steht für "Hird of Unix Replacing Daemons". Eine "Herde von Gnus" solle den Microkernel Mach dabei unterstützen, die volle Unix-Funktionalität bereitzustellen. Dabei steht "Hird" nochmals abkürzend für "Hurd of Interfaces Representing Depth".
Die Spuren von Hurd reichen zurück ins Jahr 1983. Damals Gründete Richard Stallman das GNU-Projekt "GNU's Not Unix". Stallman verfolgte damit das Ziel ein vollständiges Unix-ähnliches, freies Betriebssystem zu entwickeln. Mit großem Eifer wurden erste Ansätze für das neue Betriebssystem implementiert, jedoch über den mitunter wichtigsten Anteil, den Systemkern, war man lange Zeit unentschlossen. Die Entscheidung fiel 1988 auf den damals in der Version 3.0 vorhandenen Microkernel Mach. Gründe hierfür werden später näher erörtert. Hilfreich war sicherlich, dass Mach 3.0 1991 erstmals unter einer Lizenz herausgegeben wurde, die es erlaubte, ihn im Rahmen des GNU-Projektes zu verwenden. 1991 wurde dann durch Thomas Bushnell das GNU/Hurd-Projekt selbst ins Leben gerufen. Bis 1994 schaffte man es Hurd erstmals bootfähig zu machen. Das erste Release in der Version 0.2 von Hurd wurde erst 1997 herausgegeben. Mit der Erstellung eines Debian hurd-i386 Archivs in 1998 sollte schließlich eine benutzerfreundlichere Version des Betriebssystems verbreitet werden. Hierbei setze man zum einen auf die bereits bekannte Umgebung aus der Debian-Linux-Distribution, aber auch auf eine erleichterte Installationsroutine.

1.1 Motivationen für GNU/Hurd

Die Idee zur Entwicklung von GNU/Hurd erwuchs weitgehend aus dem Verdruss über in den meisten Unix-Systemen verbreitete Design-Probleme. Man war mit der starken Beschneidung der Berechtigung "normaler" User nicht einverstanden. Gerade die anspruchsvolleren Anwender, die gerne ihr System nach ihren Vorstellungen gestalten möchten, waren durch traditionelle Unix-Restriktionen gehindert. So ist z. B. zu bemängeln, dass ein einfacher User nicht in der Lage ist, neue, nicht vom Kernel unterstützte Dateisysteme zu mounten. Steht er vor einem derartigen Problem, braucht er Superuser-Berechtigungen, um den entsprechenden Dateisystem-Treiber in den Kernel nachzuladen, oder gar im Ernstfall einen neuen, anders konfigurierten, Systemkern zu kompilieren.
Um den erkannten Missständen in den meisten Unix-Systemen entgegen zu wirken, sollte ein völlig neues Design verwirklicht werden. Man spielte mit Gedanken an eine objektorientierte Organisation, die die Gesamtfunktionalität des Betriebs-

systems in viele Einzelkomponenten zerlegen sollte. Diese, auch aus den objektorientierten Programmiersprachen bekannte Idee, bringt zahlreiche Vorteile mit sich: Man verspricht sich einfache Erweiterung und Skalierbarkeit des Systems. Zudem sollte Hurd durch derartiges Design hohe Stabilität erreichen. Dennoch sollte bei allen diesen Neuerungen die Kompatibilität zu Unix sowie das Bewahren des POSIX-Standard nicht verloren gehen.
Aus den gesammelten Erfahrungen und Anregungen bildeten sich konkrete Ziele, wie Hurd gestaltet werden sollte. Wichtigstes Ziel war es, den vor unprivilegierten Anwendern zu schützenden System-Code, so begrenzt wie möglich zu halten. Nur das Notwendigste sollte im Kernel selbst implementiert werden. Demnach sollte der gesamte Rest des Systems dynamisch ersetzbar sein. Dies sollte der geforderten besseren "Update-Fähigkeit" des Betriebssystems genügen. Des weiteren würde es durch die Auslagerung von Systemfunktionalität aus dem geschützten Systemkern heraus jedem Anwender möglich, sein System durch neue Komponenten oder Features zu bereichern. Trotz all dem wollte man es schaffen, die System-Sicherheit durch diese Zugeständnisse an die Anwender, nicht zu gefährden.
Möglichst wenig Code im Kernel hat auch andere Vorteile: Weniger Programmcode beinhaltet gewöhnlich weniger Fehler und ist überschaubar sowie leichter zu debuggen. Mit Verwendung eines kompakten Kernels wollte man so auch für höhere Stabilität des Systems sorgen.

2 Mach Microkernel

Mach entstand 1985 an der Carnegie Mellon University. Zunächst wurde er dort unter Verwendung des BSD UNIX Kernel entwickelt, um volle Kompatibilität zu diesem zu wahren. Ziel war es einen kleinen kompakten Kernel zu entwickeln. Bis zur Version 2.5 entfernte man sich immer weiter von diesem Ziel. Der Funktionsumfang von Mach wuchs stark an und damit auch seine Größe. In der Version 2.5 war er bereits umfangreicher als der damalige Unix-Kernel. Man erkannte diese Entwicklung und arbeitete mit der Version 3.0 wieder auf das ursprüngliche Ziel hin, indem man sämtlichen Unix-Code aus dem Kernel entfernte.

2.1 Mach Abstraktionen

Mach baut grundsätzlich auf sechs Abstraktionen auf:

- Task
- Thread
- Port
- Message

- Memory Object
- Device

Task und Thread sind Bestandteile eines jeden Unix-Prozesses. Dabei nimmt die Task eine mehr passive, der Thread die mehr aktive Stellung innerhalb eines Prozesses ein. Die Task stellt sämtliche Ressourcen bereit, die im Rahmen der Ausführung des Prozesses benötigt werden. Hierzu gehört z. B. die Verwaltung des Adressraums mit Programmtext-Segment, Daten und Stacks. Man kann sie als Container für Threads bezeichnen. Anders als bei Unix-üblichen Tasks enthalten Mach Tasks keine derart umfangreiche Environment: Es fehlen beispielsweise User- und Group-IDs, Wurzel- und Arbeitsverzeichnis sowie Filedeskriptoren.
Der Thread als der aktive Anteil stellt den auf der CPU zum Ablauf kommenden ausführbaren Code mit Programmzähler und Registern dar. Von einer Task können mehrere Threads abhängen, ein Thread gehört jedoch immer zu genau einer Task. Mehrere Threads einer Task teilen sich dann den selben virtuellen Adressraum und können demnach äußerst effizient miteinander Informationen austauschen.
Ein ähnliches Paar bilden Port und Message. Hier stellt der Port die Infrastruktur für Interprozesskommunikation bereit. Da Ports ebenfalls zu Ressourcen eines Prozesses zu zählen sind, werden sie der Task zugeordnet. Ports sind Einweg-Kommunikationskanäle, die dem Informationsaustausch zwischen Threads dienen. Sie lassen sich mit den bekannten Unix-Pipes vergleichen. Durch ihre Implementation als FIFO-Queue wird garantiert, dass Nachrichten nicht verloren gehen und überdies in der richtigen Reihenfolge beim Empfänger ankommen.
Message erklärt sich nun fast von selbst: Hiermit werden die Informationen (Nachrichten) beschrieben, die durch die Ports im Rahmen der Interprozesskommunikation übermittelt werden. Hier unterscheidet man zwischen einfachen und komplexen Nachrichten, wobei komplexe Nachrichten aus höheren Datenstrukturen bestehen.
Eine weitere Abstraktion des Mach Microkernel sind Memory Objects. Memory Objects dienen der Verwaltung des virtuellen Speichers im User-Level. Man kann hier von einer Art Datenstruktur sprechen, die die Grundlage der Speicherverwaltung bildet. Nach außen bietet die Mach Speicherverwaltung darauf aufbauend Interfaces für den Einsatz externer Pager an. Das heißt der Kernel selbst beherbergt nach Vorbild des Microkernel keinen eigenen Pager, sondern stützt sich bereits auf ausgelagerte Komponenten mit der entsprechenden Funktion und besitzt Kontrollfunktionen über diese.
Eine Mach Besonderheit und damit die sechste Abstraktion ist die Unterbringung von Geräten bzw. Device Drivers im Kernel selbst. Diese Eigenschaft ist für Microkernel untypisch. War es doch Ziel der meisten Microkernel-Projekte, gerade den Kern so hardwareunabhängig wie möglich zu gestalten, also sämtliche Hardwareunterstützung aus dem Kern zu entfernen und in Server-Applikationen auszulagern, so hielt man in Mach sicherlich aufgrund seiner Entstehungsgeschichte

und -Umstände (siehe oben: Anwachsen, danach nur Entfernen des Unix-Codes) an der Hardwareintegration fest.

3 Singleserver vs. Multiserver

Mach passte in das Gesamtkonzept von Hurd. Als Multiserver-Ausprägung des Microkernel wurde er dem Wunsch nach Objektorientierung im Hurd Betriebssystem gerecht. Über Vor- und Nachteile sowohl des Singleserver, als auch des Multiserver-Ansatzes wurde lange diskutiert. Im folgenden hier eine kleine Gegenüberstellung der Eigenschaften beider Architekturen:

3.1 Singleserver

Der Singleserver-Ansatz unterscheidet sich nur unwesentlich vom monolithischen System. Hier wird ein beliebiger monolithischer Kern so modifiziert, dass er als User-Level-Applikation unterhalb eines Microkernel ablauffähig ist. Der Microkernel ist dabei die einzige Schnittstelle zur Hardware. In diesem Fall stellt also eine einzige Task, der in Ausführung befindliche monolithische Kern, die gesamte Betriebssystemfunktionalität bereit. Applikationen kommunizieren ausschließlich mit diesem einen Server (daher Singleserver). Um Applikationen des monolithischen Systems, dessen Kern den Singleserver darstellt, ohne Modifikation ablauffähig zu machen, wird eine Emulationslibrary eingesetzt, die sämtliche Umgebung des monolithischen Systems für den entsprechenden Prozess beinhaltet. Beispiel für die Konstruktion eines Singleserver-Systems ist das Projekt 'Linux on L4' der TU Dresden. Demnach ist ein Singleserver-System nichts anderes als die Emulation eines monolithischen Systems unterhalb eines Microkernels. Singleserver sind bis jetzt aufgrund aufwendigerer Systemaufrufe (Trap) zwar leicht schlechter in der Performance im Vergleich zu Monolithen, bieten daher aber andere Vorteile: Möchte man etwa ein Echtzeitsystem basierend auf einem Linux-Kernel implementieren, so ist dies durch lange, ununterbrechbare Pfade von Funktionsaufrufen im Linux-Kernel in Form eines monolithischen Systems nicht möglich. Läuft der Linux-Kernel jedoch als User-Level-Task unter einem Microkernel, so sind diese Funktionen unterbrechbar, da sie nicht mehr im privilegierten Kernel-Modus ablaufen.
Die oben genannten Einbußen in der Geschwindigkeit basieren auf dem kostenintensiveren Modell der Behandlung von Traps im Microkernel-System, dem so genannten Trampolin-Mechanismus: Der Trap eines emulierten Unix-Prozesses erreicht wie gewohnt zunächst den Systemkern, also den Microkernel. Da dieser aber den Unix-Aufruf nicht verarbeiten kann, leitet er ihn sofort an die Emulationsbibliothek weiter. Diese stellt fest, um welche Art von Systemaufruf es sich handelt und beauftragt den Unix-Singleserver, ihn auszuführen. Der Singleserver

kehrt mit einem Rückgabewert schließlich zum aufrufenden Unix-Prozess zurück.

3.2 Multiserver

Die Konstruktion einer Multiserver-Ausprägung eines Microkernel Betriebssystems ist zunächst einmal der Singleserver-Form ähnlich. Sie unterscheidet sich im Grunde darin, dass hier für die Betriebssystemfunktionalität nicht mehr allein ein Server verantwortlich ist, sondern mehrere voneinander abgegrenzte kleinere Server für einzelne Funktionen zuständig sind und miteinander kooperieren. Hier laufen also mehrere Tasks parallel unterhalb des Microkernel ab. Zu unterscheiden sind horizontale und vertikale Multiserver-Systeme. In der horizontalen Form gibt es keinerlei Interserverkommunikation. Hier verlaufen logische Kommunikationswege ausschließlich zwischen Applikationen und Servern, bzw. Servern und Microkernel. Fragt also ein Benutzerprozess einen Dienst an, an dem mehrere Server beteiligt sein müssen, so findet direkte Kommunikation nie zwischen den Servern statt, sondern der Prozess muss sich mit sämtlichen Servern selbst in Verbindung setzen. Dieses Verfahren hat den Vorteil, dass mögliche Engpässe die andernfalls bei Interserverkommunikation auftreten könnten, vermieden werden. Anders die vertikale Form des Multiserver-Systems. Hier sind durchaus längere Kommunikationsketten von einem Prozess und mehreren hintereinander gereihten Servern denkbar. Benötigt hier ein Server zum Anbieten eines Dienstes die Unterstützung eines weiteren Servers, so kontaktiert er ihn selbst und wälzt diese Aufgabe nicht auf die Benutzerapplikation ab.
In Hurd werden beide Formen des Multiservers verwirklicht. An gegebener Stelle werden entsprechende Beispiele genannt.
Auch Multiserver erlauben den Einsatz einer Emulationslibrary, um Unix-Prozessen ihre übliche Umgebung zu bieten. Systemaufrufe werden wie im Singleserver per Trampolin-Mechanismus bearbeitet.

3.3 Vergleich Singleserver - Multiserver

Ein großer Vorteil von Multiserver-Systemen liegt darin, dass jeder Server seinen eigenen abgegrenzten Speicherbereich besitzt. Somit laufen verschiedene Betriebssystemfunktionen auch in unterschiedlichem Speicher ab und nicht wie beim Singleserver alles in der einer gemeinsamen Task. Dies schafft eine Ausführungsintegrität für die einzelnen Server und damit für mehr Stabilität im gesamten System.
Klare Vorteile hingegen liegen beim Singleserver in der Effizienz der Interprozesskommunikation. Wie beim monolithischen Kernel sind innerhalb des Kerns nur kurze Kommunikationswege zu erwarten. Die Struktur des Multiserver zwingt zur Verwendung eines aufwendigeren Kommunikationsaustauschs, so genanntes

Message Passing, da hier längere Kommunikationswege vorliegen.
Ebenfalls für den Singleserver spricht seine zunächst einfachere Konstruktion. Wie bereits erwähnt genügt es hier, einen modifizierten monolithischen Systemkern als Singleserver zu verwenden. Um die einzelnen Server eines Multiserver-Systems zu erstellen, ist mit wesentlich mehr Aufwand zu rechnen. Hier können oft nur kleine Code-Fragmente wiederverwendet werden. Weitgehend sind die Server jedoch neu zu implementieren.

3.4 Hurd verwendet Multiserver

Für Hurd einigten sich die Entwickler auf die Verwendung der Multiserver-Variante. Wichtigster Grund hierfür war, dass diese Architektur am besten zu den Zielen des Hurd-Projektes passte: Die Aufteilung der Gesamtfunktionalität in einzelne wohldefinierte Servereinheiten trifft genau den Gedanken eines objektorientierten Designs. Man unterschied sowohl statische als auch dynamische Eigenschaften, die für Hurd als vorteilhaft erachtet wurden.
Die statischen Eigenschaften zeigen sich während der Entwicklung: Durch die Modularisierung im Multiserver-System ist eine getrennte Entwicklung aller Einzelkomponenten möglich. Man kann einzelne Server implementieren, versuchsweise im laufenden System starten und somit auch testen. Durch klare Trennung von Funktionen sieht der Entwickler auch eine weitaus geringere Komplexität vor sich als bei einem ganzen monolithischen Kernel. Meist genügt es einzelne Server oder gar nur deren Interface zu kennen.
Dynamische Eigenschaften stellen sich zur Laufzeit des Systems heraus: Durch klar getrennte Verantwortlichkeiten und Speicherbereiche beeinflussen sich Serverapplikationen nicht direkt gegenseitig. Illegale Speicherzugriffe zwischen einzelnen Funktionen sind beispielsweise unterbunden. Dadurch kann eine höhere Stabilität des Gesamtsystems erwartet werden. Weiterhin spricht man von einer besseren Ausfalltoleranz bei der Multiserver-Ausprägung. Versagt ein Server seinen Dienst, so werden die übrigen im allgemeinen nicht davon beeinträchtigt. Sie laufen weiter und das System bleibt weitgehend verfügbar. “Abgestürzte“ Server können natürlich zur Laufzeit des Systems wieder neu gestartet werden. Was im Hinblick auf die Schwächen von gängigen Unix-Systemen auch wichtig war, ist die verbesserte Erweiterungtoleranz durch den Multiserver-Betrieb. Neue Server können einfach zur Laufzeit ins System gestartet, aber auch wieder gestoppt werden. Weiterhin ist diese Möglichkeit für jeden Benutzer gegeben, ohne dass dieser besondere Privilegien besitzt.

4 Ausgewählte Server und Funktionen

Im folgenden sollen einige Besonderheiten von Hurd dargestellt werden.

4.1 IPC in Hurd

Zur Interprozesskommunikation verwendet Hurd Mach-Ports. Ports stellen im Mach Microkernel unidirektionale Kommunikationskanäle dar, für die Zugriffsrechte vergeben werden können. Man unterscheidet “send“, “send-once“ und “receive“. “send“-Rechte können für einen Port in beliebiger Anzahl vergeben werden. Sie ermöglichen dem Berechtigten, Nachrichten in einen Port zu senden, “send-once“ mit der Besonderheit, dass nur einmal gesendet werden darf. “receive“ ist für jeden Port nur einmal zu vergeben. Es gibt also immer nur einen Server, der das “receive“-Recht für einen Port inne hält. Dieser empfängt die Messages, die von sämtlichen Sendern in den Port abgegeben werden.
Man bezeichnet Mach-Ports auch als “Message Queues“. Dieser Begriff sagt bereits einiges über die Funktionsweise der Ports aus. Ports sind in der Lage, Nachrichten, die an den Empfänger gerichtet sind, der das “receive“-Recht für diesen Port besitzt, zwischen zu speichern. Wie bereits erwähnt, sind sie in Form von FIFO-Queues implementiert. Auf diese Weise ist asynchrone IPC möglich: Prozesse, die das “send“-Recht innehalten, sind nicht gezwungen, zu warten, bis der “Receiver“ die für ihn bestimmte Nachricht abruft. Somit ist keine Blockierung der “Sender“, wie es bei synchroner IPC praktiziert wird, notwendig. Über die Vorteile bezüglich der Systemperformance lässt sich hierbei streiten. Es hängt vielmehr von Größe und Anzahl der Nachrichten ab. Geht bei “optimalen“ Bedingungen das Prinzip der asynchronen IPC auf, so ist mit einer Performance-Steigerung zu rechnen, da nur wenig gegenseitige Blockaden zwischen Prozessen notwendig sind. Führt entsprechend ungünstiger Nachrichtenaustausch zu schneller Füllung der begrenzten Message Queues, so muss auch in Hurd auf synchrone IPC umgeschaltet werden. Synchrone IPC hat ihren entscheidenden Vorteil in der Einfachheit der Umsetzung. Sicherlich ist es einfacher, für Mechanismen zu sorgen, mit denen sich Prozesse gegenseitig blockieren können, als aufwendige Message Queues zu unterhalten und die Eventualität eines Wechsels von asynchroner IPC in synchrone IPC zu beachten zu müssen.
In Mach verwaltet ein systemweit eindeutiger Nameserver die Vergabe von send-Rechten. Dieser Nameserver überwacht eine Liste von allen im System registrierten Servern. Der Port zu diesem Server wird direkt durch den Mach Kernel angeboten. Die Anfrage nach einem send-Recht auf einen Port zu einem registrierten Server verarbeitet der Nameserver durch einfaches Lookup in der Serverliste.
In Hurd hat man das von Mach vorgegebene Nameserver-Prinzip leicht modifiziert. Die Rolle des Nameservers spielt hier das Dateisystem. Die gesamte Verzeichnisstruktur ist also nichts anderes als ein allumfassender Server-Namespace. Zum einen ist jeder Server an einer Stelle im Dateisystem eingehängt, andererseits ist aber auch jedes gewöhnliche Verzeichnis Server des darunter liegenden Dateisystems. Somit ist tatsächlich jeder Server in Hurd vom Root-Dateisystem aus zu erreichen. Das send-Recht auf einem Port zum Root-Server ist jeder Hurd-Task in ihrer Environment mitgegeben. Möchte also eine Applikation einen be-

stimmten Server kontaktieren, so muss der Pfad zu diesem Server ausgehend vom Root-Server rekursiv über die C-Funktion “hurd_file_name_lookup()“ aufgelöst werden. Diese Operation gleicht nun mehr einem Lookup in einem baumartigen Verzeichnis statt in einer linearen Liste, was sicherlich eine gewisse Performance-Steigerung im Server-lookup mit sich bringt.
Ein Beispiel soll die Funktionsweise von “hurd_file_name_lookup()“ näher verdeutlichen: Angenommen, eine Applikation möchte ein send-Recht auf einen Port zum Server mit dem Suchpfad “/path/to/server/“. Bisher kennt der Prozess durch seine Environment nur den Root-Server. Entsprechend wendet er sich mit dem oben genannten Funktionsaufruf zunächst an den Root. Innerhalb von “hurd_file_name_lookup()“ wird “dir_lookup(root port, “/path/to/server/“)“ aufgerufen. Dies ist nun explizit die Anfrage an den Root-Server, entweder einen Teil oder den gesamten Pfad zum gesuchten Server aufzulösen. Offensichtlich ist Root nicht der gesuchte Server, so dass er nur einen Teil des Pfades, das Unterverzeichnis (den Server) “/path/“ finden kann. Entsprechend gibt er eine Nachricht an die Applikation zurück in Form des Funktionsaufrufs “retry(child port, “to/server/“)“. Die Applikation weiß hiermit nun den Port zu dem nächsten, gemäß dem gesuchten Pfad, dem Root-Server untergeordneten Server: Er entspricht dem Argument “child port“. Zudem lässt der Root-Server den Anwendungsprozess wissen, welcher Restpfad noch aufzulösen ist: In diesem Fall “to/server“. Nun liegt es wieder bei der Applikation, sich erneut mit “dir_lookup (child port, “to/server“)“ an einen Nameserver zu wenden. In diesem Fall der child-Server. Wenn man davon ausgeht, dass der gesuchte Server nun direkt im Dateisystem des child-Servers liegt, kann dieser jetzt den vollen Restpfad auflösen und den gesuchten Server-Port, bzw. das send-Recht auf diesen Port an den aufrufenden Prozess zurückgeben. Damit ist der Applikation die Möglichkeit gegeben, mit dem gewünschten Server direkt zu kommunizieren.
Als kleine Randbemerkung: In diesem Beispiel stellt sich die horizontale Ausprägung einen Multiserver-Systems heraus. Direkte Kommunikation findet immer nur zwischen Applikation und Server statt, nicht jedoch zwischen einzelnen Servern. So wäre es ja auch gut möglich, dass der Root-Server auf die Anfrage der Applikation hin, den vollständigen Pfad selbständig auflöst, d. h. sich mit den unter ihm liegenden Servern in Verbindung zu setzen, bis er den Port des gewünschten Servers übergeben bekommt.

4.2 Translator-Mechanismus

Ein für Anwender recht angenehmes Feature in Hurd ist der Einsatz so genannter Translatoren. Translatoren sind Hilfsprogramme, die es ermöglichen sollen, mit nahezu jedem Server (Dateisysteme, FTP, ...) auf gleiche Art kommunizieren zu können. Ein Translator soll möglichst immer ein aus dem Standard-Dateisystem bekanntes Benutzerinterface anbieten. Der Translator-Mechanismus ist eine Er-

weiterung der zuvor genannten Prozedur, Server in Hurd zu kontaktieren. Allerdings wird hier durch den angesprochenen Server kein Port an die Applikation selbst zurückgegeben, stattdessen wird ein Translatorprogramm gestartet, das den Port zum Server erhält. Das Translatorprogramm wiederum, natürlich auch wieder als Server im Dateisystem repräsentiert, gibt nun einen Port an den Benutzerprozess zurück, der die Anfrage gestartet hat. Ab jetzt kann die Applikation über den Translator mit dem gewünschten Server kommunizieren. Interessant wird dieser Mechanismus vor allem beim Mounten fremder Dateisysteme: Der Anwender sowie Anwenderprozesse können auf die gemounteten Dateisysteme über den Mountpunkt hinweg, mit den üblichen Operationen zugreifen, indem der Translator die bekannten in die eigentlich benötigten Anweisungen übersetzt. Verwirklicht wird dieses Prinzip unter anderem auch in Transparentem FTP und Archivierungsprogrammen.
Zu unterscheiden sind aktive und passive Translatoren. Aktive Translatoren werden bei Bedarf gestartet und nach Verwendung wieder beendet. Sie überdauern damit auch keinen Systemneustart. Anders die Passiven Translatoren: Sie sind zunächst Einträge ins Dateisystem, die bei Nachfrage nach einem entsprechenden Translator veranlassen, dass dieser als aktiver Translator gestartet wird. Nach diesem ersten Starten steht der Translator für die gesamte Systemlaufzeit zur Verfügung. Die Passiven Translatoren, die ja somit nichts anderes als im Dateisystem gespeicherte Befehle darstellen, überdauern Systemneustarts.
Entsprechend dem letzten Beispiel zu möglichen Interserverbeziehungen handelt es sich hier um die vertikale Ausführung eines Multiserver-Systems. Sowohl das Dateisystem der angeforderten Datei als auch der Translator sind Server-Applikationen. Es ist sogar denkbar, dass mehrere Translatoren hintereinander geschaltet sind: Beispielsweise beim Zugriff auf eine Datei in einem tar-Archiv unter einem hinzu gemounteten Dateisystem. Man sieht, dass hier jeweils Kommunikation der Server untereinander notwendig ist.

4.3 Authentication Server

Um die Sicherheit in Hurd zu gewährleisten, ist vom System ein eindeutig bestimmter Authentication Server vorgegeben. Natürlich können Anwender auch eigene Authentication Server implementieren und zum Ablauf bringen, jedoch sorgen Mechanismen, wie z. B. die "Trusted Connection" zwischen Usern und Servern dafür, das derartige Eigenimplementierungen allenfalls mit den System eigenen koexistieren können, diese aber niemals in der Funktion ersetzen können, was ja als hochgradiges Sicherheitsrisiko einzuschätzen wäre. Ein Anwender könnte also höchstens seine eigenen Server darauf abrichten, dass sie seinen eigenen Authentication Server anerkennen.
Der Authentication Server verwaltet User-Identitäten. Eine Identität ist vergleichbar mit dem send-Recht auf einen Port zum Authentication Server. Dieser verwaltet für jeden User vier Mengen von User-IDs: "effective user ids", "available

user ids“, “effective group ids“, “available group ids“. Dabei repräsentiert die ID 0 stets Superuser-Berechtigung. “effective“ bedeutet, dass die entsprechenden IDs direkt zum Überprüfen von Berechtigungen verwendet werden können. “available“ können auf Anfrage in “effective“ umgewandelt werden.
Der Authentication Server lässt mehrere Operationen auf User-Identitäten zu, die sich jeweils auf die Mengen der IDs einer Identität auswirken. So ist es möglich, die IDs zweier Identitäten in einer neuen zu vereinigen. Es entsteht also eine User-Identität, die alle Berechtigungen der beiden ursprünglichen Identitäten enthält. Ebenso können eingeschränkte Identitäten geschaffen werden, indem man nur eine Teilmenge der vorhandenen IDs übernimmt. Die Erstellung völlig neuer Identitäten ohne Verwendung der beiden genannten Operationen obliegt natürlich nur dem Superuser.
Ebenfalls als Operation durch den Authentication Server gegeben ist die Möglichkeit, die bereits genannten 'Trusted Connections' zu knüpfen: Möchte ein User mit einem Server kommunizieren, der Authentifizierung benötigt, so verlangt der Server das der User mit ihm eine Trusted Connection eingeht. Dazu wendet sich der User zunächst durch die C-Funktion “io_reauthenticate()“ an den Server und verlangt Re-Authentication. Dabei sendet er einen so genannten Rendezvous-Port an den Server. Gleichzeitig übermittelt der User einem Authentication Server diesen Rendezvous-Port sowie seine Identität durch “auth_user_authenticate()“. Entsprechend liefert der Zielserver durch “auth_server_authenticate()“ seinen eigenen receive-Port, sowie ebenfalls den Rendezvous-Port an einen Authentication Server. Die Verbindung kann nur dann zustande kommen, wenn beide denselben Authentication Server angesprochen haben. Der Zielserver, der ja Authentifizierung verlangt, würde sicherlich den System eigenen Authentication Server verwenden. Der User hingegen könnte seine eigene Implementierung ansprechen. In diesem Fall wäre keine Trusted Connection möglich, da der Authentication Server nur in der Lage ist, die Verbindung zu vollenden, wenn er von beiden, von User und Server, den Rendezvous-Port bekommen hat. Der Authentication Server prüft die ihm übermittelten Rendezvous Ports auf Gleichheit. Im positiven Falle vergibt er dann den Server Port an den User und entsprechend dessen Identität an den Zielserver. Damit ist die Trusted Connection etabliert und beiden können direkt kommunizieren, d. h. der Zielserver akzeptiert nun, da ihm die Identität des Users vorliegt, die von ihm eingehenden Nachrichten.

4.4 Process Server

Dem Process Server in Hurd kommt wiederum eine Buchhalter- und Nameserverfunktion zu. Zunächst einmal vergibt er für jeden Prozess eine systemweit eindeutige PID, mit deren Hilfe der Superuser jederzeit Kontrolle über alle im System ablaufenden Tasks behalten kann. Durch den Process Server werden darüber hinaus zusätzliche Informationen über jeden Prozess bereitgehalten: So speichert er auch Environment-Pointer (envp) und Argumentvektor (argv), Werte, die dem

Prozess bei seiner Generierung durch “fork()“ oder “exec()“ übergeben wurden. Über die PID ist es weiterhin möglich, ein send-Recht auf einen Port zu jedem Prozess zu erhalten. Hier kommt die Nameserverfunktion zum tragen.

4.5 Programmausführung

An der Erstellung einer neuen Task, z. B. durch einen “exec()“-Aufruf, sind drei Komponenten des Hurd Systems beteiligt: Die Emulationslibrary (GlibC), der Fileserver sowie der Exec-Server. Die Library ist, wie bereits der Name andeutet, dafür verantwortlich zwischen der ablaufenden Task und dem Mach Kernel zu vermitteln. Soll etwa ein Unix-Programm im Rahmen einer Mach-Task zum Ablauf gebracht werden, so muss die Library für den Unix-Prozess dessen gewohnte Umgebung emulieren. Zudem sorgt die Emulationslibrary auch für die Verwaltung von Environment und Argumentvektor. Die Rolle des Fileservers bei der Programmausführung besteht weitgehend darin, Präfix-Pfade aufzulösen und Ausführunsberechtigungen zu überprüfen. Der Exec Server erstellt schließlich die Task und bringt den Prozess zum Ablauf. Unix-typisch ist die Eigenschaft, dass der Exec Server mehrere ausführbare Formate sowie auch shell-Scripte mit dem entsprechenden Interpreter ausführen kann.

4.6 Aufbau einer Unix-Task

Wie bereits erwähnt spielt die Emulationslibrary im Rahmen der Ausführung von Unix-Prozessen eine Entscheidende Rolle. Sie wird bei der Erstellung einer neuen Task jeweils in deren virtuellen Speicher eingeblendet. Ein Thread enthält hier also nicht nur den ausführbaren Unix-Programmcode, sondern zusätzlich die hinzu gelinkte Library. Für einen Unix-Prozess werden dadurch die benötigten Komponenten, die er sonst im Unix-Kernel wiederfinden würde simuliert. So ist es für Unix-Prozesse üblich, dass sie beim Öffnen einer Datei einen Filedeskriptor zurückgegeben bekommen. Hurd kennt allerdings nur Ports. Daher linkt die Library die entsprechenden Unix-System-Aufrufe auf die Mach-spezifischen um. Der Port auf eine Datei wird intern auf eine virtuelle OpenFileTable und FileDeskriptorTable umgelegt. Ebenso werden Präfixpfade, die einem Unix-Prozess grundsätzlich zur Verfügung stehen, wie z. B. “CWD“ für “Current Working Directory“, mit Hilfe des File Servers aufgelöst und dem Prozess in Unix-üblicher Form durch die Library bereitgestellt. Auch die Überwachung des Signal-Status, d. h. die Überprüfung, ob für den Prozess ein Unix-Signal evtl. von einem anderen Prozess anliegt, übernimmt die Library. Hier steht sie mit dem Exec-Server in Verbindung, und übersetzt die eingehenden Mach-Signale in Unix-Signale. Nicht zuletzt müssen auch Prozess-Kreierungs bzw. Beendungsaufrufe, wie “exec()“, “fork()“ oder “kill()“ auf das zugehörige Mach-Äquivalent gemapped werden.
Durch den Einsatz einer Emulationslibrary erreicht man ein, für von Grund auf neu entwickelte Betriebssysteme, äußerst wichtiges Ziel: Kompatibilität zu be-

reits bestehenden, verbreiteten Systemen. Auf diese Weise spricht man sogleich ein breiteres Anwenderspektrum an und kann mit größerer Verbreitung und Unterstützung seines Projektes rechnen.

5 Ausblick

Im Hinblick auf obige Feststellung über die Unix-Kompatibilität lässt sich feststellen, dass GNU/Hurd sicherlich noch nicht den erwarteten Stellenwert beim Anwender als Unix-Ersatz eingenommen hat. Es ist im Moment auch trotz des interessanten Designs des Betriebssystems unwahrscheinlich, dass es sich gegenüber dem schnell erstarkten Konkurrenten "Linux" behaupten kann. Es ist verständlich, dass kein Bedarf nach einem System besteht, dass sicherlich noch viele Schwächen in der Hardwareunterstützung aufweist, wenn man Unix-Ersatz auch unterhalb des mittlerweile sehr umfangreichen und mächtigen Linux-Kernels bekommen kann. Sogar Entwickler des GNU/Hurd-Projektes gaben zu: "Linux war einfach früher in einem interessanten Stadium. Wäre Linux ein bis zwei Jahre früher gekommen, so wäre das Hurd-Projekt nie entstanden." Die Zukunft wird zeigen, ob Hurd das Potential besitzt, sich in der Anwenderwelt eine bedeutsame Position neben anderen Unix-Derivaten zu verschaffen.

Literatur

[1] Trent Fisher/Free Software Foundation: Towards a New Strategy of OS Design, http://www.gnu.org/software/hurd/hurd-paper.html

[2] Marcus Brinkmann: Hurd-Talk, http://www.gnu.org/hurd/hurd-talk.html

[3] The GNU Hurd, http://www.gnu.org/software/hurd/hurd.html

[4] Rheinhard Spurk: Der Microkernel-Ansatz, Vorlesungsfolien Betriebssystem-Praxis, LS Prof. Scheidig, Universität des Saarlandes

[5] Lars Reuther, Martin Pohlack, Alexander Warg: Microkernel-Based Systems, TU Dresden Operating Systems Group