# Ein zuverlässigeres Unix

**Joachim Junk**
j.junk@dieseitenmacher.de

**ABSTRACT**
Monolitische Betriebssysteme bilden einen einzigen Prozess, der im Kerneladressraum läuft. Der Kernel enthält u. a. Funktionen für Speicher-, Prozessverwaltung und Prozessinterkommunikation, sowie Hardwaretreiber. Aus der gemeinsamen Nutzung des Kerneladressraums resultiert die privilegierte Prozessausführung im Kernelmodus. Dabei können sich alle Funktionen gegenseitig aufrufen und auf beliebige Dateien zugreifen, weil sie mit mehr Privilegien laufen als sie tatsächlich benötigen. Das ist problematisch, weil das Betriebssystem selbst beschädigt werden kann. Ferner können abgestürzte Komponenten evtl. nicht neu gestartet werden, weil sie quasi im Kernelmodus eingefroren wurden.

Die Lösung des Problems besteht darin, große Teile bisheriger Betriebsarchitekturen aus dem Kerneladressraum in den Benutzeradressraum zu verlagern, wie z. B. Hardwaretreiber. Benutzerprozesse werden immer mit niedrigeren Privilegien ausgeführt, wodurch ein Neustart abgestürzter Komponenten erleichtert wird. Andere Prozesse bleiben von einem Prozessabsturz weitgehend unberührt, weshalb solche Mikrokernelsysteme zuverlässiger und sicherer sind.

**KEYWORDS**
Monolitisches Betriebssystem, Mikrokernel, Privilegien, Benutzeradressraum, separater Benutzerprozess, Zuverlässigkeit, Sicherheit.

## 1 EINLEITUNG

Computersysteme werden seit Jahrzehnten von Menschen verwendet. In diesem Zeitraum haben sich die Anforderungen an Computer verändert, aber die Veränderungen der Computer und damit der Betriebssysteme, haben nur zum Teil mit den veränderten Benutzererwartungen Schritt gehalten.

Bis in die 70er und 80er Jahre wurden Computer vorwiegend von technisch interessierten und versierten Benutzern verwendet. Für diese Benutzer waren Computerabstürze und deren Behebung nichts Ungewöhnliches. Zu dieser Zeit hatten die Betriebssysteme bereits ihre heutige Form und wurden auf die PCs portiert. Weil Computer zu dieser Zeit sehr langsam waren und man dennoch hohe Ausführungsgeschwindigkeiten auf ihnen erzielen wollte, wurden Betriebssysteme als monolithische Programme realisiert. Außerdem waren die damaligen Betriebssysteme noch sehr klein und die damaligen Computerbenutzer waren technisch versierte Anwender, die etwaige Fehler selbst beheben konnten. So wurde UNIX als ein interaktives Betriebssystem entworfen, um möglichst viele erfahrene Benutzer gleichzeitig zu verwalten. Ziel von UNIX war es, diesen erfahrenen Benutzern ein System zur Verfügung zu stellen, mit dem sie zusammenarbeiten und kontrolliert Nachrichten austauschen konnten.

Zunächst orientierte sich das Design des Betriebssystems an der damals noch teuren Hardware. Mit dem POSIX-Standard wurden dem Betriebssystem, zunächst nur für UNIX, Bibliotheksfunktionen hinzugefügt, die jedes konforme Betriebssystem anbieten sollte. Dieser Standard wird heute auch von anderen Betriebssystemen eingehalten und ermöglicht es Softwareherstellern Programme zu entwickeln, die bei Verwendung vordefinierte Prozeduren, auf jedem konformen System laufen.

Seit Anfang der 90er Jahre steigt die Zahl der Computerbenutzer stark an. Gründe hierfür waren und sind sinkende Hardwarepreise, neue Hardware und erfolgreiche Marketingstrategien einiger großer Softwarehersteller. Unterstützt wurde diese Entwicklung durch die Verwendung mausgesteuerter GUIs und neuer kostengünstiger Peripheriegeräte, wie Scanner, USB, Digitalkameras usw., wobei diese Gerätetreiber jetzt einen Großteil des Betriebssystems ausmachen.

Entsprechend hat sich auch das Profil der Computerbenutzer, weg vom versierten Enthusiasten, hin zum durchschnittlichen Anwender, verändert. Der durchschnittliche Anwender erwartet keine Systemabstürze oder Fehlermeldungen seiner Anwendungsprogramme. Die von ihm ausgeführten Aktionen sollen, wie von ihm erwartet und schnell ausgeführt werden, so wie er es von anderen elektrischen Geräten kennt. Auch wenn Systemabstürze durch die Software und nicht durch die Hardware erzeugt werden, erwartet der Anwender die fehlerfreie Ausführung seines Computersystems. Dafür muss aber die Zuverlässigkeit der Computer verbessert werden, und diese beginnt bei der Zuverlässigkeit der Betriebssysteme.

### 1.1 Zuverlässigkeit und Sicherheit

Die Gründe für die Systemabstürze von Betriebssystemen sind zwei grundlegende Designfehler, welche sich die meisten heutigen Betriebssystem teilen. Zu viele Programme haben Privilegien des Kerns und es fehlt eine angemessene Fehlerisolierung für diese Programme. In heutigen Betriebssystemen werden viele Module gegen einen gemeinsamen Adressraum gelinkt, um das Betriebssystem als einen großen Prozess im Kernelmodus auszuführen. Verstärkt wird dieser Effekt durch neue Softwareprodukte die Neuerungen oder Erweiterungen enthalten, wodurch diese Software komplexer und unzulässiger werden kann als ihr Vorgängerprodukt.

Ein Fehler in einem Modul kann Datenstrukturen, auch unbeteiligter Module oder sogar des Kerns, und damit das gesamte System zum Erliegen bringen. Denn aus Performancegründen wurden und werden alle Module in einem gemeinsamen Adressraum zusammengefasst, ohne

die Adressbereiche dieser Module mit effizienten Schutzmechanismen auszustatten. Damit ist gemeint, dass Module mit mehr Rechten ausgeführt werden als sie tatsächlich benötigen, und zwar mit den Privilegien des Betriebssystemkerns.

Die Fehlerisolation zwischen den Modulen ist aber notwendig, weil kein Modul fehlerfrei ist. Der Linuxkernel besteht aus 2,5 Millionen [1] und der Windowskernel sogar aus 5 Millionen [1] Codezeilen. Studien zufolge liegt die maximale Fehlerrate bei 16 bis 75 Fehler pro 1000 Zeilen [2] Code. Neuere Studien gehen von 6 bis 16 Bugs pro 1000 Zeilen [1] Code aus. Selbst bei einer optimistischen Annahme von nur 6 Fehlern pro 1000 Zeilen Code, hat der Windowskernel insgesamt 30.000 Bugs, währen der Linuxkernel 15.000 Bugs aufweist. Bugs werden auch verwendet, um absichtlich Würmer oder Viren in ein Betriebssystem einzuschleusen und es zu schädigen.

Fehlerfreier Code ist nicht realisierbar und schon gar nicht in Betriebssystemen, welche in C oder C++ geschrieben sind. Programme, die in diesen Programmiersprachen geschrieben werden, machen reichlich Gebrauch von Zeigern und sind somit Quellen zahlreicher Bugs.

Die Zuverlässigkeit eines Betriebssystems bedingt dessen Sicherheit. Das bedeutet, eine Anwendung muss sich darauf verlassen können, dass ihre Daten, die sich im flüchtigen oder nicht flüchtigen Speicher befinden, nicht durch andere Prozesse unberechtigterweise gelesen oder verändert werden

Um die Zuverlässigkeit und damit die Sicherheit von Betriebssystemen zu erhöhen, ist fehlerhafter Code zu isolieren. Hierbei wird angenommen, dass jeder Code fehlerhaft sein kann, ob beabsichtigt oder unbeabsichtigt. Deshalb sollen so wenig Programme wie möglich oder zumindest die Programme von Drittanbietern nicht mit den Privilegien des Betriebssystemkerns in dessen Adressraum ausgeführt werden. Wenn Kernelcode privilegierte CPU-Instruktionen ausführen kann, können diese auch von Gerätetreibern ausgeführt werden, da sie Bestandteile des Kernels sind. Mögliche Ursachen sind ungültige Zeiger oder Pufferüberläufe, die durch jeden Gerätetreiber ausgelöst werden können.

Stattdessen dürfen Gerätetreiber und andere Programme nur in ihren eigenen Adressbereichen mit minimalen Privilegien laufen. Auf diese Art wird fehlerhafter Code isoliert. Programmfehler können nicht mehr auf den Adressraum des Kernels zugreifen und Datenstrukturen des Kernels oder anderer Prozesse schädigen. In der Folge bleiben Fehler auf den Adressraum des weniger privilegierten Prozesses beschränkt und verhindern dadurch Abstürze des Systems oder anderer Prozesse.

Ferner muss die Komplexität bestehender Betriebssysteme reduziert werden, weil einzelne kleinere Module von einzelnen Menschen leichter zu verstehen sind.

### 1.2 Unzuverlässigkeit durch Gerätetreiber

Ca. 85 % der Systemabstürze [2] unter Windows XP können auf Bugs der Gerätetreiber zurückgeführt werden. Nicht berücksichtigt werden Programmfehler, die absichtlich in Form von Viren oder Würmern in ein System eingebracht werden, um es zu schädigen. Dennoch wird fremder Programmcode, z. B. als Gerätetreiber, in das System eingebracht. Die Qualität dieses Codes kann stark variieren, je nachdem, ob er vom Gerätehersteller selbst oder einem weniger versierten Programmierer stammt.

Oft sind Dokumentationen für Betriebssysteme und Hardware lückenhaft, unvollständig oder einfach falsch, so dass Treiberprogrammierer nicht in der Lage sind fehlerfreien Code zu erzeugen.

Unter Linux ist die Fehlerrate der Gerätetreiber drei bis siebenmal [2] größer als die des restlichen Kernels.

## 2 DESIGN DER BETRIEBSSYSTEME

Um die Zuverlässigkeit von Betriebssystemen zu beurteilen, werden diese anhand ihres Designs unterschieden. Im Rahmen dieser Seminararbeit werden nur monolithische Systeme und Systeme mit einem minimalen Kern (Mikrokernel) vorgestellt.

Die Betriebssystemarchitekturen unterschieden sich dadurch, ob die Funktionen, die sie ihren Benutzern und seinen Anwendungsprogrammen anbieten, im Betriebssystemkern enthalten sind oder nicht.

### 2.1 Monolithische Systeme

Wie schon in Abschnitt 1.1 beschrieben, beinhaltet der Betriebssystemkern neben den Gerätetreibern auch alle Funktionen zur Hauptspeicher-, Prozessverwaltung und zur Interprozesskommunikation (IPC). Das gesamte Betriebssystem ist ein einziger Prozess in einem Adressraum, welches im Kernelmodus läuft. Das Betriebssystem ist in Komponenten oder Module aufgeteilt, die als abgegrenzte Bereiche im Adressraum des Kernels existieren.

Monolithische Betriebssysteme verwenden Hardwareschutz, wie virtuellen Speicher und Schutzringe, siehe Abb. 1.

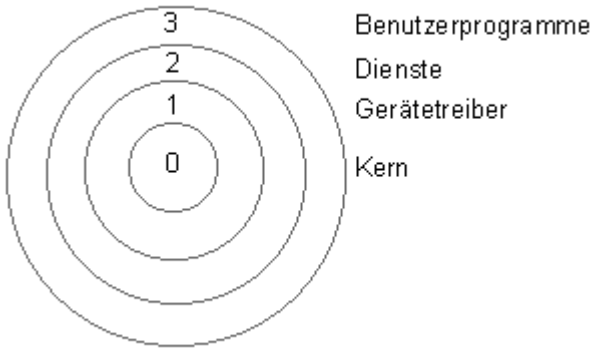



**Abbildung 1. Die Abstraktions- bzw. Schutzebenen des Pentium. [3]**

Innerhalb dieser Schichten gibt es verschiedene Zugriffsrechte. Damit soll von der Hardware abstrahiert und die Prozesse sollen voneinander getrennt werden. Allerdings wurde hier die ursprüngliche Aufteilung der verschiedenen Privilegienebenen aufgegeben, so dass lediglich nur noch die Benutzerprogramme im Ring 3

laufen, während das Betriebssystem, die Gerätetreiber und andere Dienste im Ring 0 verwendet werden.

Das Betriebssystem dient der Verwaltung der Hardware und es stellt für alle Programme eine Schnittstelle der Systemaufrufe bereit. Mit Hilfe der Systemaufrufe können Benutzerprogramme Dienste des Betriebssystems in Anspruch nehmen, z. B. das Kopieren von Daten.

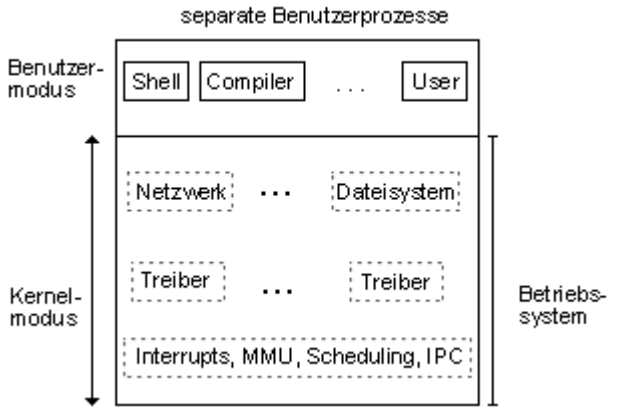
Um in einem monolithischen System den Kern des Betriebssystems zu schützen, muss ein Benutzerprozess einen Systemaufruf ausführen, wenn es eine bestimmte Funktion im Kern braucht. Durch den Systemaufruf wird ein Supervisor Call (TRAP) veranlasst, welcher eine Unterbrechung auslöst. Die Unterbrechung erzeugt einen Kontextwechsel vom Benutzermodus in den Kernelmodus.

Erst jetzt kann das Betriebssystem die Kontrolle übernehmen und die gewünschte Betriebssystemroutine des Benutzerprogramms ausführen. Nach der Beendigung der Betriebssystemroutine wird durch einen privilegierten Befehl vom Kernelmodus zurück in den Benutzermodus geschaltet. Dabei muss der Speicher jedes Prozesses geschützt werden. Dafür werden Grenzregister verwendet und ein unzulässiger Zugriff auf eine Adresse außerhalb des Grenzregisters löst einen TRAP aus, der den laufenden Prozess unterbricht (Segmentation Violation Interrupt).

Wenn ein Kernelprozess einen Fehler erzeugt, und dazu gehören in monolithischen Systemen auch Prozesse der Gerätetreiber, ist kein Kontextwechsel in einen anderen Kernelmodus möglich. Der Kernel muss den Fehler irgendwie in seinem Adressraum beheben oder auch nicht. Dadurch sind nahezu beliebige Fehler auf Kernelebene möglich. Absichtlich manipulierter Treibercode in Form von Viren und Würmern kann zur Laufzeit seinen Maschinencode verändern und die Schutzmechanismen des Betriebssystems aushebeln, weil alle Prozesse privilegiert sind. Das Betriebssystem kann abstürzen oder ein Datenverlust ist möglich. Lediglich Fehler von Benutzerprozessen, wie zuvor beschrieben, können behandelt werden.

I/O-Operationen, die durch Prozesse angestoßen werden, führen über einen Systemaufruf zur Umschaltung vom Benutzermodus in den Kernelmodus. Nach dem Kontextwechsel informiert die CPU durch den zugehörigen Gerätetreiber den betroffenen Gerätecontroller über die auszuführende Operation. Eine Prüfung auf die Zweckmäßigkeit der Operation erfolgt nicht. Dies führt jedoch auch zu Geschwindigkeitsvorteilen. Nachdem die Operation ausgeführt wurde, sendet der Controller der CPU ein Signal, indem er ein Bit im Unterbrechungsregister setzt. Hierdurch unterbricht die CPU den laufenden Prozess. Sie sichert den Programmzähler und den Zustand der Register des unterbrochenen Prozesses auf dem Systemstapel. Anhand der Position des gesetzten Bits im Unterbrechungsregister wir die Position der Anfangsadresse der Unterbrechungsroutine für das Gerät im Unterbrechungsvektor lokalisiert und dann ausgeführt.

Ein Gerätetreiber kann im Kernelmodus, aufgrund eines Codefragments, in eine unendliche Schleife eintreten und nicht mehr terminieren, wodurch CPU-Zeit verbraucht wird und das gesamte System beeinträchtigt wird. Würde der Treiberprozess im Benutzermodus laufen, wäre er unter der vollständigen Kontrolle des Schedulers und die CPU könnte dem Prozess entzogen werden.




**„Abbildung 2. Aufbau eines monolithischen Systems. Das gesamte Betriebssystem läuft im Kernelmodus ohne klare Fehlerisolierung.“ [3]**

Dennoch wird in monolithischen Systemen weiterhin fremder Code, in Form von Gerätetreibern, in den Betriebssystemkern eingebracht. Diese Ausführungen verdeutlichen, dass nahezu alle Aktionen auf einem Computersystem in einem privilegierten Modus und „quasi“ in einem Adressraum, dem des Kerns, ausgeführt werden.

Zwar werden unerlaubte Speicherzugriffe der Prozesse durch die CPU und die MMU (Memory Management Unit) vermieden, aber dennoch laufen diese im Kernelmodus.

Unter Linux besteht die Möglichkeit der Kernel zu verkleinern, indem Gerätetreiber als so genannte Module aus dem Kernel ausgelagert werden. Allerdings müssen die Gerätetreiber zu ihrer Verwendung wieder in den Kernel geladen werden und laufen dann wieder privilegiert im Adressraum des Kernels.

Mögliche Änderungen der Gerätetreiber oder andere Teile können eine vollständige Kompilierung des Kernels erforderlich machen. Teilweise können Module jedoch einzeln übersetzt werden.

Zusammenfassend werden die Nachteile monolithischer Systeme nochmals aufgeführt:

- Keine klare Fehlerisolierung. Fehler treten im Kernelmodus auf, wodurch auch der Kern des Betriebssystems geschädigt werden kann.
- Der meiste Code läuft im Kernelmodus und kann Strukturen anderer Prozesse und des Betriebssystems schädigen (Zeiger).
- Durch die Größe des Betriebssystems existieren, wie unter Abschnitt 1.1 beschrieben, sehr viele Bugs im Code.
- Teilweise unsicherer Code von Drittanbietern wird innerhalb des Betriebssystemkerns ausgeführt. Viren und Würmer können das Betriebssystem völlig unbrauchbar machen.

- Die Komplexität monolithischer Systeme ist selbst für große Teams versierter Programmierer nicht mehr beherrschbar.

Trotz aller Nachteile monolithischer Systeme sollen deren Vorteile nicht unerwähnt bleiben. Diese beschränken sich überwiegend auf deren gute Performance, die zum Zeitpunkt ihrer Entstehung ein wichtiges Leistungsmerkmal war. So ist die Performance monolithischer Systeme etwa 5 % bis 10 % [3] besser, als die von Systemen mit Mikrokernel. Gründe hierfür sind:

- Alle Funktionen des Betriebssystems laufen im Kernelmodus. Zeit- und rechenaufwändige Kontextwechsel werden vermieden. Vor allem wird die Zahl der auszutauschenden Botschaften reduziert, da in diesen Systemen jegliche Kommunikation zwischen Prozessen erlaubt ist.
- Die aufwändige Kommunikation zwischen Teilen des Betriebssystems entfällt und es werden Probleme vermieden, die durch ein weiteres Aufteilen der Betriebssystemfunktionalitäten entstehen würden.

### 2.2 Mikrokernel

Die Idee der Mikrokernel lässt sich bis Mitte der 70er Jahre zurückverfolgen. Zu dieser Zeit wurden sehr kleine Betriebssystemkerne erzeugt, z. B. Amoeba, Chorus, Mach, V und Minix. Ziel war es die Zahl schwerer Kernelfehler zu reduzieren, in dem möglichst viele Module aus dem Kernel entfernt wurden. Somit war es jetzt einfacher, aufgrund der geringeren Komplexität des Kernels, Bugs zu finden und zu eliminieren.

Der Mikrokernel ist nur verantwortlich für das Interrupt-Handling und die IPC. Er enthält den Scheduler und Mechanismen zum Starten und Beenden von Prozessen. Alle anderen Bestandteile werden in den Benutzeradressraum verlegt und laufen nicht mehr im privilegierten Kernelmodus. Ein solches Betriebssystem reduziert zwar die Fülle möglicher Probleme monolithischer Betriebssysteme bezüglich Bugs und nicht vertrauenswürdigem Code. Aber es muss noch festgelegt werden, welcher Nachrichtenaustausch zwischen welchen Kommunikationspartnern erlaubt ist, siehe Abschnitt 3.1.4 und Abschnitt 3.2.3 – Prozess Manager.

#### *2.2.1 Single Server*

Es existiert ein einziger Server, welcher alle höheren Funktionen des Betriebssystems bereitstellt. Der Server läuft im Benutzeradressraum. Ein Absturz des Servers kann das gesamte System beeinträchtigen. Die Architektur ähnelt sehr einem monolithischen System.

#### *2.2.2 Multi Server*

In diesem Architekturmodell werden nicht vertrauenswürdige Module als streng voneinander getrennte Prozesse im Benutzeradressraum ausgeführt. Bis auf den Kernel laufen die übrigen Bestandteile des Betriebssystems als Server und die Treiberprozesse im Benutzermodus. Die in Abschnitt 2.1 vorgestellten Schutzringe der CPU werden auf Ebene der Software wieder eingeführt.

Durch die strenge Trennung der Server und deren Ausführung im unprivilegierten Modus hat der Absturz eines Servers keine Auswirkungen auf andere Prozesse.

## 3 MINIX

Das Betriebssystem MINIX wurde 1987 von Andrew S. Tanenbaum zu Lehrzwecken entwickelt. MINIX ist heute in der Version 3.2 verfügbar und basiert auf der in Abschnitt 2.2.2 beschriebenen Multi Server Architektur.

Aus dem Betriebssystemkern wurden alle Gerätetreiber und Dienste (Server) entfernt und liegen jetzt oberhalb des Mikrokernels, im Benutzeradressraum des Kernels. Die Gerätetreiber und Server sind in ihren eigenen Adressbereichen gekapselt, die durch die MMU geschützt werden.

Die unter Abschnitt 2. erwähnten Schutzringe haben unter MINIX ihre ursprüngliche Bedeutung erhalten. Im Ring 0 ist der Betriebssystemkern, in Ring 1 sind die Gerätetreiber, in Ring 2 die Dienste und in Ring 3 sind die übrigen Anwendungsprogramme angesiedelt.

Der Mikrokernel übernimmt die Programmierung der CPU und MMU. Er ist verantwortlich für das Interrupt-Handling, IPC und einfache Speicherverwaltung. Er enthält den Scheduler und Mechanismen zum Starten und Beenden von Prozessen. Alle anderen Bestandteile wurden in den Benutzeradressraum verlegt und laufen nicht mehr im privilegierten Kernelmodus.

### 3.1 Designziele von MINIX

Wesentliches Ziel des MINIX-Systems ist seine Zuverlässigkeit.

#### *3.1.1 Einfachheit*

Das MINIX-System soll so einfach wie möglich und damit auch leicht verständlich sein. Der Kernel von MINIX 3.2 umfasst 3800 Zeilen Code [3]. Damit ist er wesentlich kleiner als die Kernel monolithischer Betriebssysteme. Ebenso soll der Betriebssystemkern keinen fremden, nicht vertrauenswürdigen Code enthalten.

#### *3.1.2 Modularität*

Das gesamte System ist aufgeteilt in eine Ansammlung kleiner unabhängiger Module. Diese Modularisierung dient der Fehlereindämmung, und zwar in dem Sinne, dass Abhängigkeiten zwischen Modulen soweit wie möglich vermieden werden. Damit haben Fehler eines Moduls keine Auswirkungen auf andere Module. Einzelne Module können ausgetauscht oder entfernt werden, ohne dass das Betriebssystem, insbesondere der Kern, davon betroffen ist.

Weitere Vorteile der Modularität:

- Kurze Entwicklungszyklen, weil Server eines Multi Server Systems wie andere Anwendungen geschrieben, kompiliert, getestet und Fehler behoben werden können.
- Normales Programmiermodell, weil Gerätetreiber im Benutzermodus laufen, können sie wie normale Anwendungsprogramme, Systembibliotheken nutzen.
- Leichte Fehlerbehebung, weil Tests ohne Neustarts möglich sind.

Weil solche Abhängigkeiten nicht immer vermeidbar sind, werden weitere Sicherheitsmechanismen notwendig. Beispielsweise kann das Dateisystemmodul so entworfen werden, dass es auf Treiberfehler reagieren kann.

### *3.1.3 Minimum an Befugnissen*

Fehler in privilegierten Modulen dürfen sich nicht in unprivilegierten Modulen ausbreiten, wodurch sich größere Schäden ausbreiten können. Natürlich breiten sich Fehler unprivilegierter Module auch nicht aus.

Eine Fehlerisolierung ist nur möglich, wenn alle Server und Gerätetreiber unprivilegiert sind. Denn nur dann kann der Fehler tatsächlich auf den Adressraum des Prozesses begrenzt werden, der für den Fehler verantwortlich ist. Lediglich der Kernel unterhält Bitmaps, um Nachrichten zur Fehlerbehandlung zu versenden.

### *3.1.4 Fehlertoleranz*

MINIX wurde so entworfen, dass es den meisten Fehlern widerstehen kann. Dabei werden alle Server und Gerätetreiber vom Reincarnation Server (siehe Absch. 3.2.3) verwaltet und beobachtet.

Die Entwickler von MINIX verwenden den Begriff Failure Resilience [4] (Fehlertoleranz). Die Fehlertoleranz beruht auf der Fehlerisolierung, der Fehlererkennung und Recoverymaßnahmen zur Laufzeit.

Fehlertoleranz meint, dass kein System frei von Fehlern ist. Deshalb können Fehler immer und überall auftreten. MINIX soll dann geeignet auf diese Fehler reagieren, ohne dass das System dadurch in Mitleidenschaft gezogen wird. Betroffene Module sollen wieder in das laufende System integriert werden. Das alles soll ohne Eingriff des Benutzers erfolgen.

Fault Isolation meint, dass ein Bug in einem Prozess sich nicht über Prozessgrenzen ausbreiten und vermehren kann. Denn die Adressbereiche der Prozesse sind durch den Kernel und die MMU geschützt. Wann immer Prozesse auf gemeinsame Daten zugreifen, kann ein einzelner Prozess Zugriff auf dezidierte Speicherbereiche gewähren, indem er dem Kernel eine Zugriffsliste für diesen Speicherbereich sendet. Bei jedem Lese- und Schreibversuch auf den freigegebenen Speicherbereich, werden dann die Zugriffe vom Kernel kontrolliert. Außerdem haben Fehler in Benutzerprozessen keine Möglichkeit auf Kernelprozesse oder die Hardware zuzugreifen, weil die Modularisierung des Systems harte Grenzen für jedes Modul vorsieht, die alle im Benutzeradressraum angesiedelt sind und dafür zu wenig Privilegien besitzen.

Es kann kein Benutzerprozess auf den Adressraum eines anderen Prozesses zugreifen. Alle Prozesse können nur über einen genau festgelegten Weg untereinander Informationen austauschen (siehe Absch. 3.2.3 - Prozess Manager und File Server). Dadurch und aufgrund minimaler Befugnisse der Prozesse werden bösartige Veränderungen an Prozessdaten verhindert.

In MINIX existieren unprivilegierte Benutzer- und Gruppen-IDs, um den Zugriff auf die POSIX-konformen Kernelaufrufe zu beschränken. Es existieren ferner genau definierte Zugriffsbeschränkungen für jeden Benutzerprozess, Server und Treiberprozess. Operationen werden nur mit möglichst wenigen Befugnissen ausgeführt. Die Privilegien werden durch den Reincarnation Server gesetzt und zur Laufzeit vom Kernel überprüft.

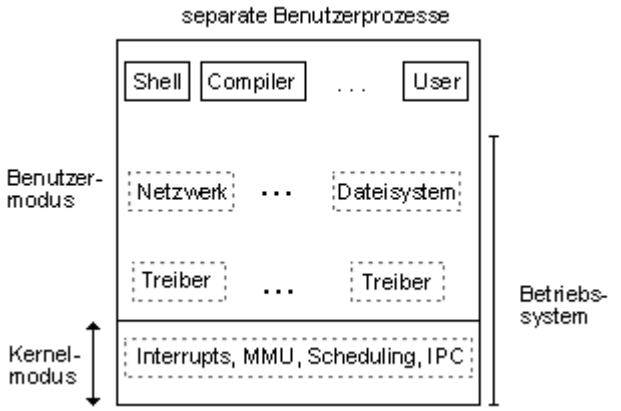
Welcher Kernelaufruf einem Prozess möglich ist, wird durch ein Bitmap in der Prozesstabelle des Kernels angezeigt. Ferner ist die Menge möglicher Kernelaufrufe und die IPC für jeden Prozess limitiert und definiert. Beispielsweise wird jeder Gerätetreiber mit einer Schutzdatei geladen. Diese Datei enthält Angaben der benötigten Treiberressourcen, zu verwendende I/O-Ports, Interrupts und Berechtigungen für die Interprozesskommunikation (IPC). Der Reincarnation Server überwacht alle Gerätetreiber und Server im System. Er dient ferner der Fehlererkennung, Fehlerbehebung und wird in Abschnitt 3.2.3 beschrieben.

Für Recoverymaßnahmen im laufenden Betrieb, als letzter Punkt der Fehlertoleranz, wird der unter Abschnitt 3.2.3 beschriebene Data Store verwendet.

Sollte beispielsweise das Versagen eines Festplattentreibers vom Reincarnation Server festgestellt werden, kann der File Server schnell anhand des Data Store, die fehlgeschlagene Operation des Gerätetreibers rekonstruieren. Die zugrunde liegende Annahme ist, dass solche Fehler vorübergehend sind und durch einen Neustart des Prozesses behoben werden.

## 3.2 Multi Server-Archiktur von MINIX

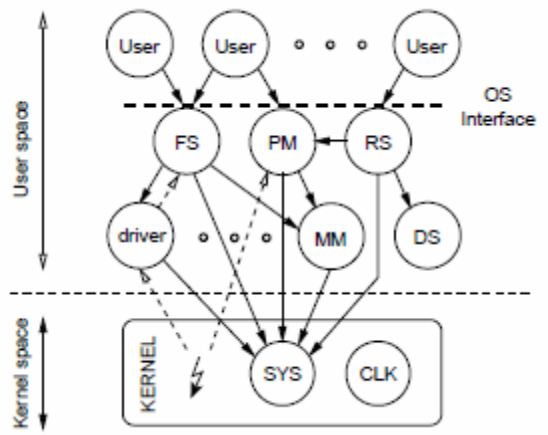
Entsprechend der unter Abschnitt 3.1 aufgeführten Designziele, werden unter MINIX 3.2 fast alle Funktionen des Betriebssystems, wie beispielsweise die Gerätetreiber oder das Dateisystem, als voneinander abgegrenzte Benutzerprozesse in ihrem eigenen privaten Adressbereich ausgeführt.




**„Abbildung 3. Architektur von MINIX. Das Betriebssystem läuft als eine Sammlung isolierter Benutzerprozesse auf dem Mikrokernel.“ [3]**

Jeder Server und Gerätetreiber verwendet einen separaten Benutzerprozess, dessen Adressbereich vollständig von den Adressbreichen des Kernels, anderer Server, Gerätetreiber und anderen Benutzerprozessen getrennt ist. Unprivilegierte Prozesse erhalten, aufgrund fehlender Befugnisse, keine Möglichkeit andere Prozesse zu stören oder auf die Adressräume anderer Prozesse zuzugreifen. In diesem Modell teilen sich die Prozesse also nicht irgendwelche verteilten Adressbereiche in einem gemeinsamen Adressbereich und sie können nur über eine genau definierte IPC miteinander kommunizieren, die im Kernel angesiedelt ist.

Die Architektur des MINIX-Systems beschreibt vier Ebenen, wobei Ebene vier den Anwendungsprogrammen vorbehalten ist. Ebene 3 ist für die Server und Ebene 2 ist für die Gerätetreiber reserviert. Die Ebenen 2 und 3 bilden das POSIX-konforme Multiserver-Betriebssystem, dass in Abbildung 3 dargestellt wird.




**„Abbildung 4. Kernkomponenten des Multi Server-Betriebssystems, und einige typische IPC-Wege. Top-down IPC wird blockiert, während button-up IPC zugelassen wird. " [2]**

In Ebene 1 läuft lediglich der Mikrokernel im Kernelmodus.

Im Gegensatz zu UNIX verwendet MINIX verschiedene Restriktionen, um die Zuverlässigkeit und damit die Sicherheit des Betriebssystems zu gewährleisten:

- Kein unbefugter Kernelzugriff. Der SYSTEM-Task kontrolliert alles, siehe Abschnitt 3.2.1.
- Schutz der Adressbereiche. Bedingt durch minimale Befugnisse wird der Adressraum des Kernels und der anderer Prozesse geschützt, siehe Abschnitt 3.2.2 und Abschnitt 3.2.3.
- Restriktiver Zugriff auf I/O-Ports. Der Zugriff wird ebenfalls durch den SYSTEM-Task kontrolliert, siehe Abschnitt 3.2.2 und Abschnitt 4.3.
- Restriktive und zuverlässige IPC. Die Kommunikation zwischen Prozessen unterliegt bestimmten Regeln, siehe Abschnitt 3.2.1 – Prozess Manager und Abschnitt 4.5.

### *3.2.1 Der Mikrokernel von MINIX 3.2*

Charakteristisch für den Kernel ist seine geringe Größe und die Tatsache, dass er nur grundlegende Funktionen zur Speicher- und Prozessverwaltung, sowie Grundfunktionen zur Synchronisation und Kommunikation anbietet, die nicht durch unprivilegierte Benutzerprozesse ausgeführt werden können.

Der Kernel ist verantwortlich für die Kommunikation mit der Hardware. Das beinhaltet Interrupt-Handling, die Programmierung der CPU und MMU, das Schreiben auf autorisierte I/O-Ports und das Prozess-Scheduling.

Weiterhin bietet der Kernel eine zuverlässige und restriktive IPC an. Unter Verwendung getypter Nachrichten mit fester Länge, können Prozesse untereinander und mit dem Kernel kommunizieren (siehe Absch. 3.2.3 - PM und Absch. 4.5). Eine direkte Kommunikation zwischen Prozessen der Gerätetreiber ist ausgeschlossen, ebenso wie die direkte Kommunikation zwischen Prozessen der vierten Ebene und Treiberprozessen.

Der Kernel enthält zur Versorgung von Teilen des Betriebssystems im Benutzeradressraum zwei weitere Prozesse, den CLOCK-Task und den SYSTEM-Task. Diese Prozesse werden als Tasks bezeichnet, um sie von den Servern und Treiberprozessen des Benutzeradressraums zu unterscheiden.

Obwohl der SYSTEM-Task und CLOCK-Task im Adressraum des Kernels liegen und in den Kernel einkompiliert sind, werden sie als separate Prozesse mit eigenem Aufrufstapel ausgeführt. Der SYSTEM-Task und CLOCK-Task bieten für die Betriebssystemfunktionen des Benutzeradressraums Schnittstellen zu den Kerneldiensten an, wie I/O und Zeitmessungen.

Der SYSTEM-Task (SYS) ist eine Kernelschnittstelle für die Server und die Gerätetreiber, in dem er diesen 35 Kernelaufrufe [1] anbietet. Alle Prozesse, außerhalb des Kernels, sind voneinander unabhängig und können keine Operationen des Betriebssystems ausführen. Sie können lediglich den SYSTEM-Task bitten bestimmte Aufgaben wahrzunehmen. Nur Prozesse der Gerätetreiber und Server können Kernelaufrufe an den SYSTEM-Task richten.

Alle Kernelaufrufe, die der SYSTEM-Task erhält, werden in Anfragen umgewandelt. Ist der anfragende Prozess zur Anfrage berechtigt, antwortet der SYSTEM-Task entsprechend und leitet die Anfrage an den Kernel weiter oder führt sie selbst aus. Ebenso werden Antworten des Kernels nur über den SYSTEM-Task dem anfragenden Benutzerprozess übermittelt. Kernelaufrufe existieren für das Prozessmanagement, das Kopieren von Daten zwischen Prozessen, I/O, Interrupt-Management, Zugang zu Datenstrukturen des Kernels und den CLOCK-Task. Der SYSTEM-Task selbst läuft in einer Schleife und bearbeitet entsprechende Kernelaufrufe.

Führt beispielsweise ein Benutzerprozess den Systemaufruf fork aus, sendet er eine Nachricht (Systemaufruf) an den Prozess Manager. Weil fork aber Änderungen an der Prozesstabelle des Kernels erfordert, muss der Prozess Manager einen Kernelaufruf an den SYSTEM-Task senden. Denn nur SYSTEM-Task kann diese Änderungen an der Prozesstabelle des Kernels vornehmen.

Der CLOCK-Task (CLOCK) ist verantwortlich für die Berechnung der CPU-Benutzung, also das Scheduling der Prozesse, er interagiert mit der Hardwareuhr und verwaltet andere Zeitgeber. Da der CLOCK-Task sehr eng mit dem Scheduling verknüpft ist, wäre es schwierig und ineffizient den CLOCK-Task im Benutzermodus auszuführen. Deshalb ist dieser Task im Kernel angesiedelt.

Nach dem Systemstart programmiert der CLOCK-Task die Hardware-Uhr und registriert einen Interrupt-Handler, der auf der Uhr läuft. Dieser erhöht für Prozesse deren CPU-Verbrauch und dekrementiert deren Zeitscheibe.

Wenn ein neuer Prozess zur Bearbeitung ansteht, wird eine NOTIFICATION an den CLOCK-Task geschickt, damit der CLOCK-Task die Zeitscheibe des neuen Prozesses dekrementiert.

Der Benutzeradressraum des Betriebssystems hat keine direkte Schnittstelle zum CLOCK-Task. Die Dienste des CLOCK-Tasks können aus dem Benutzeradressraum nur durch Kernelaufrufe über den SYSTEM-Task in Anspruch genommen werden.

Die drei Ebenen oberhalb des Kernels können auch als eine Ebene angesehen werden, da der Kernel sie in der gleichen Art und Weise behandelt. Jedes Element dieser Ebene ist auf Instruktionen des Benutzermodus beschränkt und jedes Element dieser Ebene darf nur die Aufgaben ausführen, die ihm vom Kernel zugestanden werden. Somit hat kein Prozess des Benutzeradressraums direkten Zugriff auf I/O-Ports oder auf Bereiche außerhalb seines Speicherbereichs. Dennoch haben diese Prozesse unterschiedliche Privilegien, beispielsweise das Privileg eines Kernelaufrufs. Diese Privilegien hängen ab von der Ebene auf der sie angesiedelt sind und werden in den Abschnitten 3.2.2 und 3.2.3 erläutert.

#### *3.2.2 Treiberschicht*

Die erste Schicht im Benutzeradressraum, über der Kernelebene, ist die Schicht der Gerätetreiber. In dieser Ebene sind die Benutzerprozesse der Gerätetreiber angesiedelt. Die Treiberschicht kann als nicht vertrauenswürdig betrachtet werden, weil hier Softwareprodukte von Drittanbietern verwendet wird. Diese Schicht enthält Treiber für Tastatur, Maus, Monitor, Testplatten, Ethernet usw.

Jeder Gerätetreiber ist ein separater Prozess, der bei seiner Erzeugung ausgeführt wird.

Er kann Nachrichten an andere Prozesse schicken und Kernelaufrufe durchführen, beispielsweise um I/O-Ports lesen oder schreiben zu lassen, so wie alle anderen Benutzerprozesse der zweiten und dritten Ebene. Die Kernelaufrufe kann nur der SYSTEM-Task entgegen nehmen und nach einer Prüfung bearbeiten. Prozesse der zweiten Ebene haben die meisten Privilegien.

Die Adressbereiche der Prozesse werden durch die CPU und MMU vor unbefugtem Zugriff geschützt, so wie alle anderen Benutzerprozesse. Die Treiberprozesse dürfen einige Kernelaufrufe erzeugen, um Ergebnisse einiger privilegierter Operationen zu erhalten, beispielsweise um von I/O-Ports zu lesen oder um diese zu schreiben oder um Daten zwischen verschiedenen Adressbereichen kopieren zu lassen. Der Kernel plant aber diese erlaubten Anfragen. Hierfür benutzt er eine Bitmap in seiner Prozesstabelle, welche kontrolliert welche Aufrufe für welchen Treiber oder Server erlaubt sind. Der Kernel enthält auch Datenstrukturen mit Zugriffsberechtigungen, die definieren welche I/O-Geräte ein Treiber benutzen darf oder ob beispielsweise das Kopieren von Daten erlaubt ist.

#### *3.2.3 Serverschicht*

Das Betriebssystem stellt den Benutzerprozessen der vierten Ebene, mit den Servern der dritten Ebene und in Form von Systemaufrufen, die erforderlichen Schnittstellen zur Kommunikation bereit. Diese Ebene stellt eine Erweiterung des MINIX-Systems im Benutzeradressraum dar und ist vertrauenswürdig, weil sie nicht von Drittanbietern stammt. Benutzerprozesse der vierten Ebene verwenden POSIX-konforme Systemaufrufe, in dem sie Nachrichten an die entsprechenden Server senden. Die Serverschicht liegt über der Treiberschicht. Wie die Gerätetreiber, sind die Server gewöhnliche Prozesse, besitzen aber eine niedrigere Priorität als die Treiberprozesse.

An dieser Stelle soll die Unterscheidung zwischen Kernelaufrufen und Systemaufrufen betrachtet werden, die so in monolithischen Systemen nicht anzutreffen ist. Unter MINIX sind Kernelaufrufe Low-Level-Operationen, die vom SYSTEM-Task entgegen genommen werden. Sie ermöglichen den Treiber- und Serverprozessen ihre Arbeit, da sie auf Ergebnisse oder Meldungen des SYSTEM-Tasks angewiesen sind. Das Lesen eines I/O-Ports ist ein solcher Kernelaufruf.

Im Gegensatz dazu sind Systemaufrufe, wie beispielsweise fork High-Level-Operationen, die durch den POSIX-Standard definiert sind. Dies sind die einzigen Aufrufe, die von Benutzerprozessen der vierten Ebene durchgeführt werden können. Kernelaufrufe sind Benutzerprozessen der vierten Ebene nicht gestattet.

In monolithischen Betriebssystemen ist ein Systemaufruf immer das Anfordern eines Kerneldienstes und kann von jedem Prozess aufgerufen werden. Als Konsequenz eines Systemaufrufs erfolgt immer ein Kernelaufruf und damit ein Moduswechsel vom Benutzermodus in den Kernelmodus. Im Kernelmodus erhält der aufrufende Prozess weitergehende Privilegien, eine etwaige Fehlerbehandlung obliegt jetzt dem Betriebssystemkern in seinem Adressraum, wodurch eine Fehlerisolierung erschwert ist.

Der Prozessmanager (PM) bildet mit dem File Server (FS) die POSIX-kompatible Schnittstelle für die Anwendungsprogramme.

Zu den Aufgaben des PM gehören:

- Neue Prozesse erzeugen.
- Prozesse entfernen.
- Vergeben der Prozess-IDs.
- Vergabe der Prozessprioritäten.

Die POSIX-konforme Signalverwaltung wird vom Prozess Manager übernommen. Benutzerprozesse können hier Signalhandler registrieren um vom Prozess Manager Signale zu erhalten. Dadurch unterbricht der Prozess Manager Systemaufrufe und setzt einen Signalrahmen auf den Prozessstack, damit der Benutzerprozess den Signalhandler benutzen kann. Allerdings ist diese Vorgehensweise für Systemprozesse (zweite und dritte Ebene) unpassend. Systemprozesse benutzen eine Erweiterung des POSIX-Aufrufs sigaction() [2], welche unter MINIX implementiert ist.

Mit dieser Erweiterung können Systemprozesse den Prozess Manager anweisen Signale in Notification-Nachrichten umzuwandeln. Dieser Nachrichtentyp hat die höchste Priorität aller Nachrichten und wird deshalb sofort behandelt.

Auf Kernelebene werden zur Behandlung der Prozesse CPU-Register verwendet, während der Prozess Manager Policies des Prozessmanagements implementiert, also die Festlegung der Prioritäten von Benutzerprozessen. Der Kernel führt Listen über den erlaubten Nachrichtenaustausch zwischen Prozessen. Diese Informationen sind auch in den Prozesstabellen der einzelnen Prozesse hinterlegt und werden vom Kernel kontrolliert. Der Kernel behandelt also eigentlich alle Prozesse gleich und ist dafür verantwortlich, dass der Prozess mit der höchsten Priorität die CPU erhält.

Die synchrone IPC zwischen Prozessen folgt dem Rendezvous-Prinzip. D. h., wenn ein Prozess versucht eine Nachricht zu erhalten, obwohl ihm keine gesendet wurde, blockiert er. Ebenfalls blockiert auch ein Nachrichtensender, wenn kein Empfänger für seine Nachricht existiert. Nur wenn Sender und Empfänger für den Nachrichtenaustausch bereit sind, kopiert der Kernel die Nachricht aus dem Adressraum des Senders in den Adressraum des Empfängers. Diese Vorgehensweise benötigt keine Pufferverwaltung. Es ist weniger komplex und verbraucht weniger Ressourcen.

Wie in Abbildung 3. ersichtlich, wird auch ein Versand unsinniger Nachrichten zwischen Gerätetreibern ausgeschlossen. Beispielsweise kann ein Sound-Treiber keine Nachrichten an einen Drucker-Treiber senden, denn es werden die in der Abbildung 3, vorgestellten Kommunikationswege eingehalten.

Der Memory Manager (MM) ist im Prozess Manager integriert und verwaltet freie Speicherbereiche, indem er eine nach Speicheradressen sortierte Liste von freien Speicherbereichen unterhält. Der Memory Manager entscheidet anhand dieser Liste wo im Speicher ein Prozess abgelegt wird.

Die eigentliche Bearbeitung der Liste erfolgt jedoch im SYSTEM-Task des Kernels.

Jeder Prozess besteht aus einem Text-, Stack- und Datensegment. Das schreibgeschützte Textsegment ist ausführbar und kann auch von mehreren Prozessen verwendet werden um dasselbe Programm auszuführen. Dadurch werden zwar keine Pufferüberläufe verhindert, aber sie werden wesentlich erschwert, da die Inhalte des Daten-, Stacksegments und des Heaps nicht ausführbar sind und in Adressräumen unprivilegierter Prozesse erfolgen. Lediglich Systemprozesse können Zugriff auf weitere Speichersegmente erhalten. Etwaige Pufferüberläufe in Benutzerprozessen werden verkraftet, da Server und Gerätetreiber in ihren Handlungen und Adressbereichen beschränkt sind.

Der File Server (FS) erhält von Benutzerprozessen Botschaften zum Lesen und Schreiben von Dateien. Hierfür werden POSIX-konforme Aufrufe, wie open(), read(), oder write() verwendet. Der File Server unterstützt auch symbolische Links.

Aus Performancegründen werden Daten im File Server-Cache gespeichert. Falls eine Anfrage eines Benutzerprozesses nicht aus dem File Server-Cache beantwortet werden kann, sendet der File Server eine Anfrage fester Länge an den entsprechenden Treiberprozess. Auf die Antwort des Treibers lädt er dann die Daten in seinen Cache. Dann führt der File Server einen Kernelaufruf durch und fragt den SYSTEM-Task, ob er die Daten in den Adressraum des Benutzerprozesses kopiert. Für den vorgestellten Fall, dass die nachgefragten Daten nicht im File Server-Cache vorhanden sind, ist ein Mehraufwand von 4 Botschaften und 2 Kontextwechsel notwendig. In Abschnitt 5 wird dieser Vorgang detailliert beschrieben.

Aus Gründen der Zuverlässigkeit wird der File Server-Cache periodisch auf die Festplatte zurück geschrieben.

Der Reincarnation Server (RS) ist die wichtigste Komponente für das Zusammenspiel und die Verwaltung der Server und Gerätetreiber. Er beobachtet deren Verhalten. Da der Reincarnation Server der Elternprozess aller Server- und Treiberprozesse ist, erkennt er deren Prozessabstürze. Der Reincarnation Server überprüft u. a. auch die in Abschnitt 3.1.4 erwähnten Schutzdateien der Gerätetreiber, vor allem auch beim Start neuer Treiberprozesse.

Der Reincarnation Server sendet seinen Kindprozessen periodische Statusanfragen, die die Kindprozesse innerhalb bestimmter Fristen beantworten müssen, anderenfalls tritt der Reincarnation Server in Aktion.

Die Antworten der Statusanfragen werden auch für etwaige Recoverymaßnahmen abgestürzter und neu zu startender Prozesse verwendet. Sie werden im Data Store, der als nächster Punkt erläutert wird, gespeichert.

Bei der Entdeckung eines Fehlers durch den Reincarnation Server schaut dieser in einem speziellen Skript (malfunctioning component's policy script) [1] nach, welche Fehlerbehandlung vorzunehmen ist. Fehlerhafte Prozesse, auch Systemprozesse, werden beendet und unmittelbar neu gestartet. Es werden Recoverymaßnahmen für den betroffenen Prozess durchgeführt und andere vom Prozessneustart betroffene Prozesse über etwaige Änderungen des wieder hergestellten Prozesses informiert. Ein neu gestarteter Prozess muss wieder in das System integriert werden. Dafür muss der Reincarnation Server wieder den Prozessnamen mit seiner ID im Data Store publizieren, um andere prozessabhängige Komponenten von der neuen Systemkonfiguration zu unterrichten.

Beispielsweise muss im Fall eines Neustarts des Festplattentreibers dem File Server bekannt gemacht werden, dass der Gerätetreiber neu gestartet wurde.

Ferner muss dem File Server mitgeteilt werden, dass für den Festplattentreiber ein neuer IPC-Endpunkt (ID) im Data Store existiert. Erst jetzt kann der File Server seine eigene Tabelle aktualisieren und den Festplattentreiber bitten sich mit den Recoveryinformationen aus dem Data Store zu reinitialisieren.

Als Wächter des Betriebssystems schützt der Reincarnation Server dessen Funktionalität. Das Programm service [2] versorgt Benutzerprozesse mit einer passenden Schnittstelle zum Reincarnation Server. service erlaubt es dem Administrator Systemprozesse zu starten und zu beenden.

Mit service kann der Administrator das oben erwähnte Policy Skript aktualisieren.

Wenn der Reincarnation Server kein Recovery durchführen kann, wird der Fehler zur Anwendungsebene transportiert und der Benutzer darüber informiert, dass die gewünschte Operation nicht ausgeführt werden kann.

Wesentlich ist, dass das System und alle anderen Prozesse in der Zwischenzeit normal weiterlaufen.

Der Information Server (IS) führt Debugging- und Statusinformationen für Gerätetreiber und Server zur Auswertung mit.

Der Data Store (DS) dient u. a. den zuvor beschriebenen Recoverymaßnahmen, er wird aber auch von Systemprozessen verwendet, die hier zusätzlich einen Teil ihrer privaten Information ablegen. Diese redundante Speicherung von Informationen erhöht die Zuverlässigkeit des Systems, auch für Recoverymaßnahmen bei einem Prozessabsturz.

Der Data Store ist aber auch eine einfache und flexible Möglichkeit der Interaktion zwischen Prozessen des Betriebssystems nach dem Erzeuger-Verbraucher-Prinzip.

Der Erzeuger kann mit seiner ID Daten im Data Store publizieren. Der Verbraucher kann diese Daten selektieren und abonnieren, nach der Erzeuger-ID oder bestimmten Ereignissen, an denen er interessiert ist. Alle beteiligten Komponenten werden vom Data Store automatisch über Updates, mittels „broadcast notifications“ [2], informiert.

Vergleichbar dem DNS (Domain Name System), bietet der Data Store einen Namensdienst für Prozesse an. Denn jeder Prozess hat einen eindeutigen IPC-Endpunkt, der vom Kernel automatisch generiert wird und es den Systemprozessen erschwert sich gegenseitig zu finden. Der Namensdienst versieht die IPC-Endpunkte mit Identifizieren, die global gültig sind und den Systemprozessen das gegenseitige Auffinden erleichtern. Jedes Mal, wenn ein Prozess gestartet oder neu gestartet wird, aktualisiert der Reincarnation Server diese Namensauflösung.

Wie Gerätetreiber können auch Server hinzugefügt oder entfernt werden, ohne den Kernel neu zu übersetzen. Sie werden beim Systemstart automatisch oder später gestartet, wenn sie gebraucht werden.

Allgemein bezeichnet man die Prozesse der zweiten und dritten Ebene als Systemprozesse. Lediglich Prozesse der vierten Ebene sind reine Benutzerprozesse. Benutzerprozesse haben die niedrigste Priorität und gelten wiederum als nicht vertrauenswürdig. Allerdings gibt es auch Ausnahmen, die aber vorwiegend Prozesse der zweiten und dritten Ebene betreffen. Obwohl ein Prozess der zweiten Ebene i. d. R. eine höhere Priorität als ein Prozess der dritten Ebene hat, kann ein Treiberprozess seine hohe Priorität an einen Serverprozess abgeben, wenn dieser sehr schnell seine Arbeit verrichten muss.

## 4 DIE ZUVERLÄSSIGKEIT VON MINIX

### 4.1 Mikrokernel

Der Kernelcode, inklusive SYSTEM- und CLOCK-Task, der im Kernelmodus ausgeführt wird, enthält nur 3800 Codezeilen [3]. Infolgedessen enthält er wesentlich weniger Bugs, ist weniger komplex als ein monolithischer Kernel und kann von einem einzelnen Programmierer verstanden werden.

Der Quellcode monolithischer Systeme ist zu komplex, als dass er von einer einzelnen Person verstanden werden könnte. Infolgedessen ist die Möglichkeit die Anzahl der Bugs in einem solchen Programm erheblich zu reduzieren, nur sehr gering.

Die Auslagerung von Treibern und Servern in den Benutzeradressraum kann helfen die Anzahl der Bugs zu reduzieren. In jedem Fall minimiert sie die Anzahl möglicher Fehler im Adressraum des Kernels. Monolithische Kernel nehmen das Risiko in Kauf, dass Fehler in Gerätetreibern oder Servern das gesamte System beeinträchtigen, auch weil sie die gleichen Privilegien wie der Kernel besitzen.

### 4.2 Streng abgegrenzte Adressbereiche

Die MMU und der Kernel sorgen dafür, dass jeder Prozess vollständig isoliert ist, d. h. er ist auf seinen privaten Adressbereich beschränkt und dieser wird vor anderen Prozessen geschützt. Deshalb können Fehler eines Prozesses nicht auf Adressbereiche anderer Prozesse schädigen.

Der Kernel von MINIX 3.2 hat 50 [1] Bugs, während die Kernel von Linux und Windows XP bei einer weniger optimistischen Annahme mehr als 25.000 [1] bzw. 50.000 [1] Bugs haben.

So besteht beispielsweise der Betriebssystemkern von 4.4BSD aus mehreren Schichten. Die Schichten enthalten u. a. Treiber für Netzwerkgeräte, Festplatten, Sockets, Netzwerkprotokolle, Routing, Dateisysteme, virtuellen Speicher, Prozesszuteilung und vieles mehr.

Auch hier beschränken die MMU und der Kernel die Adressbereiche der einzelnen Prozesse. Da aber all diese Prozesse privilegiert und im Adressraum des Kernels laufen, ist an einen wirksamen Schutz einzelner Adressbereiche nicht zu denken. Die Zuverlässigkeit dieses UNIX-Systems ist fraglich, weil alle Komponenten des Betriebssystemkerns im privilegierten Kernelmodus laufen und Fehler auf andere Adressbereiche zugreifen können. Fraglich in dem Zusammenhang ist auch wie eine Fehlerisolierung aussehen muss. So kann jeder Gerätetreiber beispielsweise den Adressraum der Prozesszuteilung lesen und schreiben, und die Sicherheit und Zuverlässigkeit des 4.4BSD-Kern zerstören.

Das folgende Beispiel soll das verdeutlichen:

„Nehmen Sie an, dass ein aktuelles Betriebssystem (Windows oder Linux, egal) so sicher programmiert ist, dass es bei den Änderungen am GDT, dem Page Directory und den Page Tables keine Fehler macht. Ist die Speicherverwaltung dann sicher?

Nein, wäre sie nicht. Immer wenn Sie auch spezielle Hardware benutzen wollen und dafür einen Treiber

installieren müssen, läuft dieser Treiber-Code auch im Ring 0 und könnte die Datenstrukturen manipulieren.“[7]

### 4.3 Gerätetreiber im Benutzermodus

Kein Gerätetreiber läuft unter MINIX im Kernmodus und kann deshalb keine privilegierten CPU-Instruktionen ausführen. Jeder Treiber muss einen Kernelaufruf durchführen und angeben welchen I/O-Port er lesen bzw. schreiben möchte. Somit verhindert der Kernel falsche I/O-Zugriffe.

Auch hier gelten die unter Abschnitt 4.2 beschriebenen Gefährdungen des Betriebssystemkerns. Denn kein Gerätetreiber besitzt unter UNIX nur minimale Befugnisse, sondern kann bei der Erledigung seiner Aufgaben privilegierten Zugriff auf die CPU erhalten, womit es ihm möglich wäre alle Register nach Belieben zu überschreiben.

Beispiel für schädlichen Treibercode in einem monolithischen Betriebssystem:

„.. Ein Betriebssystem kann dadurch verhindern, dass ein Prozess beispielsweise zur Laufzeit seinen eigenen Maschinencode verändert. Ein Virus versucht typischerweise genau das, nämlich seinen Maschinencode in einen anderen Prozess einzufügen.“ [7]

Zur Erinnerung, in monolithischen Systemen laufen Treiberprozesse im Kernelmodus und damit privilegiert.

### 4.4 Selbstheilung von MINIX

Wie in Abschnitt.3.2.3 beschrieben, sind alle Prozesse Kindprozesse des Reincarnation Servers. Fehlerhafte Prozesse oder abgestürzte Prozesse werden neu gestartet. Dafür werden Recoverymaßnahmen durchgeführt ohne, dass das Betriebssystem und andere Prozesse davon betroffen sind.

Unter UNIX sind etwaige Recoverymaßnahmen abgestürzter Prozesse, durch Vorhalten redundanter Informationen, wie in einem Data Store, nicht möglich. Wenn Prozessdaten abgestürzter Prozesse nicht mittels Register herstellbar sind, sind sie verloren. UNIX bietet in diesem Fall keine explizite redundante Archivierung dieser Daten an und ist hier weniger zuverlässig als MINIX. Es werden lediglich fehlerhafte Komponenten durch ihren Neustart entfernt.

### 4.5 IPC

MINIX unterstützt synchrone und asynchrone IPC. Die synchrone IPC folgt dem Rendezvous-Prinzip. D. h., wenn ein Prozess versucht eine Nachricht zu erhalten, obwohl ihm keine gesendet wurde, blockiert er. Ebenfalls blockiert auch ein Nachrichtensender, wenn kein Empfänger für seine Nachricht existiert. Nur wenn Sender und Empfänger für den Nachrichtenaustausch bereit sind, kopiert der Kernel die Nachricht aus dem Adressraum des Senders in den Adressraum des Empfängers. Diese Vorgehensweise benötigt keine Pufferverwaltung, da erstens keine Puffer existieren und zweitens alle Nachrichten eine feste Länge haben. Das Verfahren ist weniger komplex und spart Ressourcen der fehlenden Pufferverwaltung.

Außerdem existiert unter MINIX der IPC-Aufruf SENDREC [3], der die IPC-Aufrufe SEND und RECEIVE zusammenfasst. Mit SENDREC wird der Sender solange blockiert bis er vom Empfänger eine Antwort auf seine Anfrage erhalten hat. SENDREC erlaubt es den normalen Benutzerprozessen mit den Servern zu kommunizieren, da diese eine POSIX-Schnittstelle anbieten. Nur über diese Schnittstelle dürfen normale Benutzerprozesse mit den Servern und somit mit dem Betriebssystem kommunizieren. Das Verfahren schützt MINIX vor Überlastung falls der Empfänger eine asynchrone Antwort senden sollte.

Normalerweise verwendet MINIX den synchronen Nachrichtenaustausch. Hierbei sind aber Deadlocks (Verklemmungen) möglich, wenn zwei oder mehr Prozesse gleichzeitig versuchen miteinander zu kommunizieren, sich die Prozesse aber blockieren, weil sie gegenseitig aufeinander warten.

Falls der zuvor beschriebene synchrone Nachrichten-austausch nicht möglich ist, wird der zuvor erwähnte Notificationmechanismus verwendet. Der Mechanismus ist asynchron und erlaubt es dem Senderprozess einem Empfängerprozess eine Nachricht zu senden, auf die der Empfängerprozess nicht augenblicklich antworten muss, ohne dass der Senderprozess dadurch blockiert wird. Notification-Nachrichten sind getypt und für jeden Prozess wird ein Bit pro Typ gespeichert. Alle Notification-Nachrichten eines Prozesses werden in einer Bitmap zusammengefasst und sind Bestandteil seiner Prozesstabelle. Wenn der Empfängerprozess bereit ist die Nachricht zu empfangen, erzeugt der Kernel eine Nachricht vom Typ NOTIFICATION und die ursprüngliche Nachricht wird zugestellt.

Interrupt-Handler benutzen auch diese Verfahren um eigene Blockierungen zu vermeiden, wenn sie Nachrichten an Gerätetreiber schicken, die ihrerseits gerade belegt sind.

MINIX kontrolliert die IPC, da der Kernel ein Bitmap pro Prozesse besitzt, dass die erlaubte IPC für den Prozess festlegt. Es wird festgelegt, ob ein Prozess einem anderen Prozess direkt ein „send“ oder „sendrec“ senden darf und ob der andere Prozess antworten darf. Diese Entscheidungen werden unter MINIX also nicht durch die Prozesse selbst getroffen, sondern durch den Kernel. Der Kernel enthält ferner Bitmaps, die festlegen welche Treiber und Server miteinander kommunizieren dürfen. Das verhindert, dass Benutzerprozesse direkt Nachrichten an die Gerätetreiber schicken.

Ebenso werden die direkte Kommunikation zwischen Gerätetreibern, Anwendungsprozessen und ein Austausch unsinniger Nachrichten ausgeschlossen. Mögliche Pufferüberläufe des Kernels beim Nachrichtenversand werden verhindert, da MINIX nur Nachrichten fester Länge zulässt.

Unter UNIX wird IPC auch kontrolliert, allerdings können Benutzerprozesse beim synchronen IPC Nachrichten versenden können, ohne die Antwort abzuwarten. Auf diese Weise können in monolithischen Systemen Empfängerprozesse blockiert werden.

Alle Prozesse unter UNIX können direkt miteinander kommunizieren. Beim synchronen Nachrichtenaustausch verwenden Prozesse Pipes (Kanäle), über die ein Strom

von Bytes übertragen wird. Die Synchronisation bedeutet, falls ein Prozess versucht aus einer leeren Pipe zu lesen, wird er blockiert. Ist die Pipe voll, wird die Übertragung der Daten gestoppt.

Auch beim asynchronen Nachrichtenaustausch können Prozesse direkt miteinander kommunizieren, mittels eines Software-Interrupts. Ein Prozess sendet einem anderen Prozess ein Signal und teilt so dem System mit, was passieren soll, wenn ein Signal eintrifft. Dafür muss jeder Prozess, der ein Signal empfangen möchte eine Signalbehandlungsroutine definieren. Dieser so genannte Signal-Handler übernimmt dann die Kontrolle, wenn ein Signal an einen Prozess gesendet wird. Im Gegensatz zu MINIX wird der Nachrichtenversand zwischen Prozessen nicht in dem Sinne kontrolliert, als nur zuvor definierte Nachrichten zwischen Prozessen zulässig sind. Deshalb können unter UNIX auch unsinnige Nachrichten versendet werden. Eine Pufferverwaltung und die dafür notwendige Bereitstellung von Ressourcen müssen von UNIX übernommen werden, auch weil unter UNIX Nachrichten beliebig lang sein können. Für den eigentlichen Nachrichtenversand ist ein Kontextwechsel in den Kernelmodus nötig. Mögliche Pufferüberläufe beim Nachrichtenversand sind dann privilegiert und können wiederum Adressbereiche anderer Prozesse zerstören, was die Zuverlässigkeit von UNIX einschränkt.

### 4.6 Interrupts

Ein Interrupt wird sofort als Nachricht dem entsprechenden Gerätetreiber gemeldet. Sollte der Treiber zu dem Zeitpunkt beschäftigt sein, erhält er NOTIFICATION-Nachricht. Diese wird der Treiber sobald wie möglich bearbeiten, ohne dass hierfür komplexe Blockierungsmaßnahmen nötig wären.

Hier wird also die in Abschnitt 4.5 beschriebene asynchrone IPC verwendet, weil der Interrupt-Handler nicht durch synchrone Nachrichten blockiert werden kann.

Jeder Gerätetreiber kann einen Kernelaufruf auf dem SYSTEM-Task durchführen. Allerdings bietet der Kernel nur wenige Kernelaufrufe und er enthält eine Bitmap für jeden Gerätetreiber. In dieser Bitmap ist hinterlegt welche Kernelaufrufe der Gerätetreiber ausführen darf. Dabei wird jeder erlaubter Zugriff durch ein eigenes Bit in der Bitmap geschützt. Der Treiberprozess kann also nur bestimmte I/O-Ports durch SYSTEM-Task lesen oder schreiben lassen. Somit ist ein Druckertreiber unter MINIX auf seine I/O-Ports beschränkt und kann keine anderen I/O-Ports benutzen

Beispielsweise können mehrere Gerätetreiber Adressräume von Benutzerprozessen lesen und schreiben lassen, um Daten zu exportieren oder einzufügen. Dabei wird aber ein vielfaches Lesen und Schreiben, mittels desselben Aufrufs, der sich nur bezüglich der Parameter unterscheidet, verhindert. Folglich kann ein Druckertreiber nur Daten eines Benutzerprozesses lesen, aber nicht schreiben, denn dann würde er den Benutzerprozess zerstören. Ein unberechtigter Zugriff des Treiberprozesses wird somit durch das oben erwähnte Bit vermieden.

In monolithischen Systemen kann ein Druckertreiber auch Adressbereiche eines Benutzerprozesses schreiben. Wie unter Abschnitt 4.5 beschrieben, können Prozesse nicht direkt miteinander kommunizieren. Das gilt auch für Treiberprozesse, um den Austausch falscher und unsinniger Informationen zu unterbinden. Unter UNIX kann der Prozess des Druckertreibers den gesamten Kernelprozess zerstören, da er selbst im Kernelmodus läuft und privilegiert ist. Treiber- und damit Kernelprozesse können beliebige I/O-Ports lesen und schreiben.

### 4.7 Zeiger

Unter MINIX können die Zeiger eines Gerätetreibers oder eines anderen Benutzerprogramms nicht auf Bereiche außerhalb des eigenen abgegrenzten Adressraums ihres Prozesses zugreifen, weil ihnen die notwendigen Befugnisse fehlen. Andererseits obliegen dem Programmierer die Überprüfungen von Zeigerreferenzen oder Arraygrenzen. Sie sind aber wegen der geringeren Komplexität, wie unter Abschnitt 1.1 beschrieben, leichter durchführbar. Die MMU kontrolliert die Adressbereiche.

### 4.8 Pufferüberläufe

Ausführbare Programme sind als Binärdateien auf dem Datenträger abgelegt. Zur Ausführung eines solchen Programms wird der dazugehörende Programmcode in den Hauptspeicher geladen und hierin ausgeführt. Ein in der Ausführung befindliches Programm wird als Prozess bezeichnet. Einem Prozess wird ein virtueller Adressraum zur Verfügung gestellt, siehe Abbildung 5.

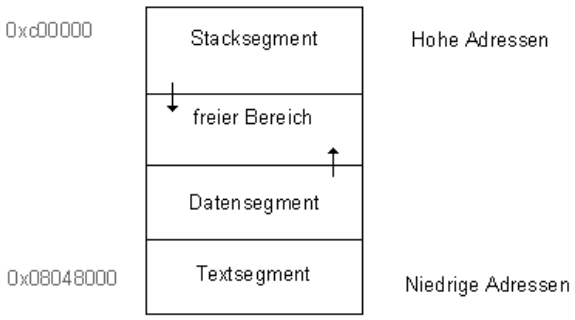



**Abbildung 5. Prozesssegmente**

Das Textsegment eines Prozesses enthält die Maschineninstruktionen, die den ausführbaren Code eines Programms darstellen. Dieses Segment ist schreibgeschützt um zu verhindern, dass ein Prozess versehentlich oder absichtlich (Würmer, Viren, etc.) seine Instruktionen überschreibt. Schreibversuche in diesem Bereich führen zu Speicherzugriffsfehlern (Segmentation Violation). Ein solcher Prozess kann in einem monolithischen Betriebssystem zu beliebigen Schäden führen, da er im Kernelmodus ausgeführt wird.

Im Datensegment sind die Variablen, Zeichenketten und andere Daten des Programms hinterlegt. Das Datensegment wächst von niedrigen zu hohen Adressen, bzw. schrumpft von hohen zu niedrigen Adressen. Es ist veränderbar, da sich Daten und Variablen eines Programms zur Laufzeit verändern können. UNIX und MINIX erlaubt das Wachsen oder Schrumpfen des Datensegments, je nachdem ob Speicher benötigt oder

freigegeben wird. Dafür kann der Systemaufruf brk [6] von Programmen benutzt werden. Durch Zeigerfehler oder falsche Datentypen in Variablen können u. a. Pufferüberläufe erzeugt werden, was unter UNIX, aufgrund der privilegierten Prozessausführung, zur Speicherverletzung führen kann.

Das Stacksegment beginnt am oberen Ende des Adressraum eines Prozesses und kann nach unten wachsen. Der Stack ist unterteilt in den User-Stack und einen Bereich für Umgebungsinformationen des Prozesses. Auf dem Stack werden mittels push- und pop-Operationen automatische Variablen (lokale Variablen einer Funktion) und Verwaltungsinformationen (Rücksprungadresse und Framepointer, bei einem Funktionsaufruf) geschrieben. Beispielsweise können Verwaltungsinformationen auf dem Stack manipuliert werden, wenn ein statischer Array mit mehr Werten gefüllt wird, als er aufnehmen kann. Der Prozess stürzt ab, weil die Rücksprungadresse überschrieben wurde und der Versuch an die überschriebene Adresse zu springen, einen Speicherzugriffsfehler erzeugt. Im Kernelmodus, wie unter UNIX, hat das verheerende Folgen.

Zur Laufzeit eines Prozesses sind unter UNIX etwaige Änderungen des Datensegments und Stacksegments möglich, da sie im Kernelmodus laufen, privilegiert. So können beispielsweise durch Viren und Würmer Änderungen der Umgebungsvariablen, falsche Zeiger, falsche Wertzuweisungen an Variablen etc. zum Prozessabsturz führen oder auf fremde Adressbereiche zugreifen.

In MINIX 3.2 können die zuvor beschriebenen Fehler auch auftreten. Anders als in monolithischen Systemen ist die Ausführung von Benutzerprozessen nicht privilegiert.

Ein solcher Prozessabsturz schädigt also nicht die Speicherbereiche anderer Prozesse oder gar die des Kernels. Ein solcher Prozess kann durch den Reincarnation Server im besten Fall neu gestartet und wiederhergestellt werden. Im schlimmsten Fall wird lediglich eine Meldung an den Benutzer zur vierten Ebene weitergereicht und die anderen Teile des Systems arbeiten weiter.

### 4.9 Unendliche Schleifen in Gerätetreibern

Der Reincarnation Server erkennt, wenn ein Treiberprozess viel CPU-Zeit verbraucht und reduziert dann seine Priorität. Wenn ein solcher Prozess nicht mehr in der vorgegebenen Zeit die Anfragen des Reincarnation Servers beantwortet, wird er beendet und neu gestartet.

Unter UNIX kann bei Schleifen in Treiberprozessen deren Priorität gesenkt werden, wodurch sie weniger CPU-Zeit erhalten. Das ist aber bei diesen privilegierten Treiberprozessen nicht immer möglich, weil der Treiberprozess mit denselben Privilegien läuft wie der Scheduler, der den Prozess beenden soll. Auch kann das Beenden dieser Prozesse zu Datenverlusten führen.

Unter MINIX hingegen hat der Scheduler immer eine höhere Priorität als der Treiberprozess und kann ihn deshalb beenden.

## 5 BEISPIEL EINES KERNELAUFRUFS IN MINIX

Die folgende Abbildung stellt die Kommunikation, den Datenaustausch und die Adressbereiche der beteiligten Prozesse im Falle eines Festplattenzugriffs dar.

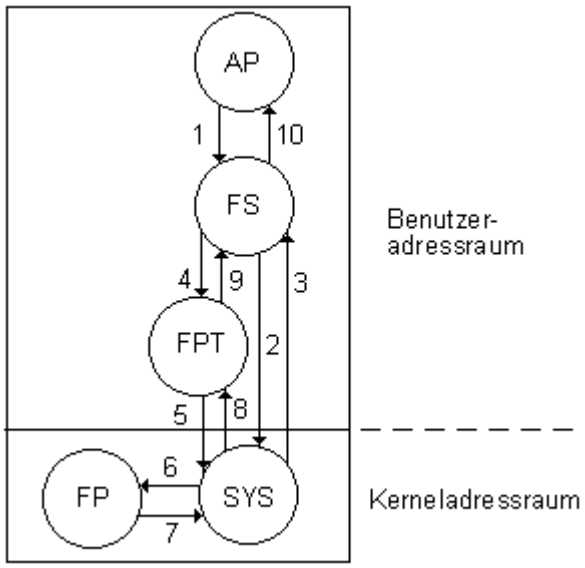



**Abbildung 6. Kommunikation und Speicherbereiche bei einem Festplattenzugriff unter MINIX.**

Der Prozess eines Anwendungsprogramms (AP) möchte einen Datenbereich der Festplatte (FP) lesen. Es sendet eine entsprechende Anfrage (1) an den File Server (FS). Bei dieser Anfrage handelt es sich nicht um einen Kernelaufruf, sondern um einen Systemaufruf (siehe Absch. 3.2.3).

Der File Server (FS) prüft zunächst die Zulässigkeit der Anfrage durch den Anwendungsprozess und ob er die nachgefragten Daten in seinem Cache hat. Ist das der Fall, richtet er einen Kernelaufruf (2) an den SYSTEM-Task. Der File Server (FS) bittet damit den SYSTEM-Task die Daten aus seinem Cache in den Puffer des Anwendungsprozesses zu kopieren. Im Erfolgsfall meldet der SYSTEM-Task dem File Server die erfolgreiche Datenübertragung (3) und der File Server gibt dem Anwendungsprozess eine Bestätigung (10), dass die nachgefragten Daten übertragen wurden.

Hat der File Server (FS) die Daten nicht in seinem Cache, sendet er dem Plattentreiber (FPT) einen Systemaufruf (4), mit der Bitte die Daten von der Festplatte (FP) zu lesen. Der Plattentreiber (FPT) seinerseits richtet einen Kernelaufruf (5) an den SYSTEM-Task (SYS). Der SYSTEM-Task prüft die Zulässigkeit der Anfrage durch den Festplattentreiber und führt die notwendigen I/O-Operationen (6, 7) durch. Der SYSTEM-Task kopiert die gelesenen Daten in den Cache des File Servers und informiert (8) den Plattentreiber (FPT). Der Plattentreiber (FPT) gibt (9) die Informationen an den File Server (FS).

Da der File Server nun die Daten in seinem Cache hat, richtet er jetzt einen Kernelaufruf (2) an den SYSTEM-Task damit dieser, wie oben beschrieben, die Daten in den Puffer des Anwendungsprozesses kopiert. Nach dem Kopiervorgang erhält der File Server eine Nachricht (3) vom SYSTEM-Task und kann daraufhin den Anwendungsprozess (AP) darüber informieren (10), dass

die angeforderten Daten vorliegen. Während dieser Kommunikation und bis zum Ende der Datenübertragung wird der anfragende Benutzerprozess vom Kernel blockiert.

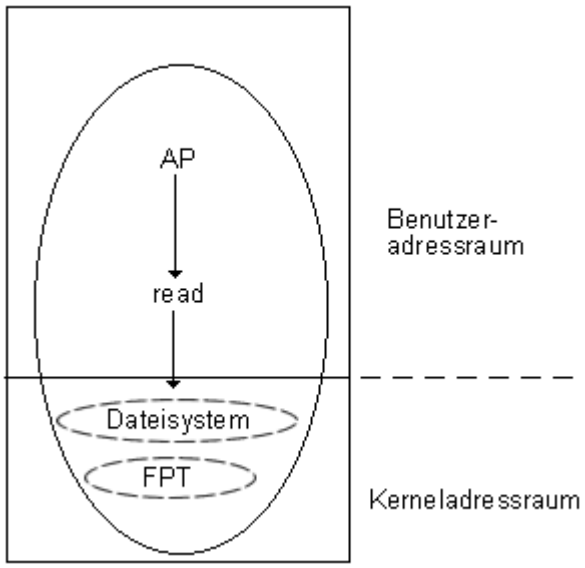



**Abbildung 7. Kommunikation und Speicherbereiche bei einem Festplattenzugriff unter UNIX.**

Alle Aktionen, bis auf den I/O-Zugriff durch den SYSTEM-Task und die Übertragung der Daten von der Festplatte zum SYSTEM-Task, finden im Benutzeradressraum statt und sind somit unprivilegiert.

Anders ist das in monolithischen Systemen, in denen alle Aktionen, bis auf den Systemaufruf des Benutzerprozesses, privilegiert im Adressraum des Kernels erfolgen.

Jeder Anwendungsprozess kann mittels eines Systemaufrufs einen Moduswechsel vom Benutzermodus in den Kernelmodus erzwingen. Das geschieht auch ohne Überprüfung, ob der Anwendungsprozess dazu berechtigt ist, oder ob eine gewünschte Operation überhaupt sinnvoll ist. So ist es beispielsweise fraglich, ob ein Prozess eines Grafikprogramms auf einen Soundtreiber zugreifen darf. Insgesamt wird der Adressbereich eines Anwendungsprozesses auf die Adressbereiche anderer Prozesse ausgedehnt, vor allen in Richtung des Kerneladressraums. Das führt zu einem Gewinn an Privilegien, die von einem normalen Anwendungsprozess missbraucht werden können.

## 6 ZUSAMMENFASSUNG

Seit Anfang der 90er Jahre steigt die Zahl der Computerbenutzer stark an. Entsprechend hat sich auch das Profil der Computerbenutzer, weg vom versierten Enthusiasten, hin zum durchschnittlichen Anwender, verändert. Der durchschnittliche Anwender erwartet keine Systemabstürze oder Fehlermeldungen seiner Anwendungsprogramme. Für ihn muss die Zuverlässigkeit der Computer verbessert werden, und diese beginnt bei der Zuverlässigkeit der Betriebssysteme.

Die Unzuverlässigkeit heutiger Betriebssysteme ist vor allem darin begründet, dass zu viele Programme im Kern des Betriebssystems enthalten sind und mit Privilegien des Betriebssystemkerns ausgeführt werden. Ferner ist der Code heutiger Betriebssysteme so umfangreich und komplex, dass er selbst für große Teams von Programmierern zu komplex ist. Dadurch bedingt existieren im Code dieser Betriebssysteme sehr viele Programmierfehler, die wiederum die Sicherheit und Zuverlässigkeit heutiger Betriebssysteme beeinträchtigen, weil sie mit zu vielen Rechten ausgeführt werden.

Erschwert wird die Situation durch das Einbringen von Fremdsoftware. Hier sind vor allem Gerätetreiber zu nennen, die von Drittanbietern bereitgestellt und installiert werden und ebenfalls mit den Privilegien des Betriebssystemkerns ausgeführt werden.

Diese monolithischen Systeme beinhalten neben dem Betriebssystemkern, den Gerätetreibern, auch alle Funktionen zur Hauptspeicher-, Prozessverwaltung und zur Interprozesskommunikation (IPC). Das gesamte Betriebssystem ist ein einziger Prozess in einem Adressraum, welcher im Kernelmodus läuft. Zwar schützen solche Systeme die Adressräume ihrer Prozesse, dennoch können sie letztlich nicht verhindern, dass privilegierte Prozesse auf fremde Adressräume zugreifen. Das gilt besonders für die eingebrachte Software von Drittanbietern, wodurch in einem System Software unterschiedlicher Qualität eingesetzt wird.

Monolithische Systeme bieten deshalb keine klare Fehlerisolierung, weil Fehler im Kernelmodus auftreten und nahezu beliebige Strukturen, auch des Betriebssystemkern schädigen können. Infolge dessen kann ein weiteres Arbeiten unmöglich gemacht und ein Neustart des Systems notwendig werden, was mit einem erheblichen Datenverlust verbunden sein kann.

Die Probleme monolithischer Systeme wurden in den 70er und 80er Jahren erkannt und es wurden Betriebssysteme mit unterschiedlichen Ansätzen entwickelt, um die oben beschriebenen Schwächen monolithischer Betriebssysteme zu lösen. Eine auf einem Mikrokernel basierende Lösung ist das von Andrew S. Tanenbaum entwickelte Betriebssystem MINIX.

MINIX 3.2 ist ein POSIX-kompatibles, UNIX-ähnliches Betriebssystem bei dem Funktionalitäten, die für den Kern nicht zwingend notwendig sind, in höhere Schichten ausgelagert wurden.

Damit ist der Kern zuverlässiger, weil er ungleich weniger Fehler enthält als monolithische Kerne. Er ist weniger komplex, wodurch Programmierfehler leichter zu entdecken und zu beheben sind. Der Mikrokernel bietet nur noch rudimentäre Mechanismen für Interrupt-Handling und IPC an. Seine Arbeit wird unterstützt durch den CLOCK-Task und SYSTEM-Task, die auch im Kernelmodus laufen. Diese beiden Tasks unterstützen Operationen des Betriebssystems im Benutzermodus.

Alle anderen Module befinden sich in höheren Schichten und laufen im Benutzeradressraum des Betriebssystems. Deshalb besitzen sie keine Privilegien der Kernelprozesse, was es sicherer macht, weil die Benutzerprozesse aufgrund ihrer geringen Befugnisse

nicht auf Strukturen des Kerneladressraums zugreifen können.

Durch diese Fehlerisolierung ist es nicht mehr möglich, dass einzelne Prozesse das gesamte System oder weite Teile davon beeinträchtigen. Prozessabstürze werden mit Hilfe des Reincarnation Servers entdeckt und behoben.

MINIX bietet mit dem Data Store ein Werkzeug für Fehlerbehandlung und Recovery an. Der Data Store enthält u. a. redundante Prozessinformationen, um Nachrichten oder systemrelevante Informationen zu publizieren.

Der Performanceverlust gegenüber monolithischen Systemen beträgt 5 % bis 10 % [2]. Die Performanceverluste von MINIX sind u. a. begründet im umfangreicheren Botschaftenaustausch der Prozesse, der Überprüfung von Prozessberechtigungen und dem Fortschreiben teils redundanter Informationen, die wiederum für Recoverymaßnahmen verwendet werden.

Aktuell steht MINIX in der Version 3.2 zur Verfügung. MINIX ist ein zuverlässigeres UNIX, dessen Zuverlässigkeit ausreichend dargestellt wurde.

## 7 LITERATUR


1. Bos, H., Herder, J.N., Homburg, P., Tanenbaum, A.S. MINIX 3: A Highly Reliable, Self-Repairing Operating System. Dept. of Computer Science, Vrije Universiteit Amsterdam.
2. Bos, H., Gras, B., Herder, J.N., Homburg, P., Tanenbaum, A.S. Reorganizing UNIX for Reliability. Dept. of Computer Science, Vrije Universiteit Amsterdam.
3. Bos, H., Herder, J.N., Tanenbaum, A.S. A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers. Technical Report IR-CS-018, January 2006.
4. Herder, J.N. Building A Dependable Operating System: Fault Tolerance in MINIX 3. Dept. of Computer Science, Vrije Universiteit Amsterdam, September 2010, 1-73.
5. Tanenbaum, A.S. Betriebssysteme – Entwurf und Realisierung, Teil 1 Lehrbuch. Wolf-Decker Haass – München; Wien: Hanser; London: (1990), 202-268.
6. Tanenbaum, A.S. Moderne Betriebssysteme, 2. überarbeitete Auflage. Pearson Studium (2003), 721-808.
7. Wohlfeil, S. Sicherheit im Internet – Ergänzungen, Kurseinheit 2: Zugriffskontrolle und Benutzer-authentisierung, Fernuniversität in Hagen, 46, 83.