Hauptseminar im WS97/98

Exokernel

Roland Schulz

28. 1. 1997

Inhaltsverzeichnis

1 Motivation der Exokernel-Idee

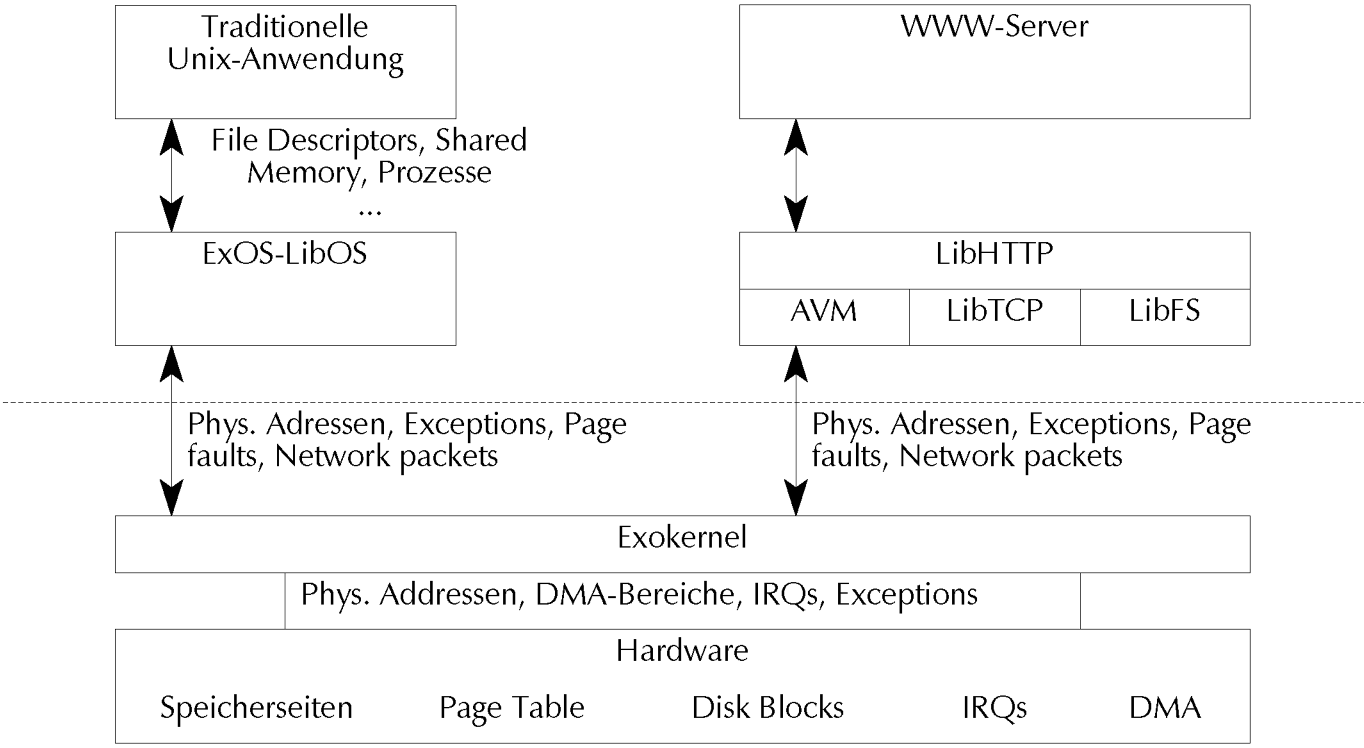
1.1 Nachteile traditioneller Betriebssysteme

Traditionelle Betriebssysteme haben die Aufgabe, die Ressourcen wie Speicher, Prozessor, Plattenplatz u.v.m. vor dem unerlaubten Zugriff zu schützen, sie zu verwalten, und den Anwendungsprogrammen eine festgelegte Abstraktionsschicht zur Verfügung zu stellen, auf der diese ihre Operationen ausführen können. Dabei treten folgende Probleme auf:

- Da es unmöglich ist durch eine Abstraktion allen Anforderungen der Anwendungen gerecht zu werden, geht Flexibilität und Performance verloren.
- Da die Implementierungen möglichst viele Anwendungsanforderungen befriedigen wollen (**Generalisierung**), sind sie unnötig komplex und damit fehleranfällig, teuer zu pflegen und zu ändern. Ebenso verschwenden sie Systemressourcen, die sonst für Anwendungen zur Verfügung stehen könnten.
- Durch die Generalisierung wird den Anwendungen ein Interface auf einer hohen Abstraktionsebene aufgezwungen, welches die effiziente Implementierung von neuen Abstraktionen verhindert.
- Die Ressourcenverteilung kann ohne Anwendungswissen ineffizient und mit hohem Aufwand verbunden sein. Die Abstraktionen verstecken Informationen der unteren Ebenen, die das Management erheblich verbessern können.
- Neue Features müssen im kernel space ablaufen, damit die nötigen Informationen und Zugriffsrechte gegeben sind. Hierdurch wird der Kern aufgebläht.

Schlußfolgerung: Abstraktionen sind zwar notwendig für die Anwendungsprogrammierung, aber diese sollten nicht im Kern erfolgen, sondern von den Anwendungen selbst gestellt werden. Auch wissen die Anwendungen selbst am besten, welche Mittel sie wann brauchen und sollten daher auch weitgehende Kontrolle über die Verteilung der Ressourcen haben.

1.2 Philosophie des Exokernel-Aufbaus



Im Exokernel-OS werden jegliche Hardware-Abstraktion aus dem Kernel in darüberliegende Schichten verdrängt. Der Kern exportiert nur die bloße Hardware wie den physikalischer Speicher (als Speicherseiten), Prozessoren (in Form von Zeitscheiben), Plattenspeicher (verteilt nach Diskblöcken), DMA-Kanäle, E/A-Geräte, TLB, Interrupts und Exceptions. Hierbei wird soweit möglich nicht von den von der Hardware vorgegebenen Bezeichnern abstrahiert, sondern diese werden ohne Umwandlung an die Anwendung weitergegeben (expose names).

Um das Management der Systemressourcen effizient gestalten zu können erhält die Anwendung weitgehende Einsicht in die Interna des Kerns (**expose information**). Dieser stellt nur sicher das Schutzbestimmungen beim Zugriff eingehalten werden (**separate protection and management**) und stellt die Mechanismen zur Verfügung, mit denen die vorhandenen Ressourcen einer Anwendung zugesprochen und wieder aberkannt werden können. Wenn möglich wird der Anwendung freigestellt, welche Instanz einer Ressource sie belegen/freigeben möchte (**expose allocation/revocation**).

Abstraktionen werden durch Libraries realisiert, die unprivilegiert im Anwendungsraum ablaufen und daher auch beliebig ausgetauscht, verändert oder aber neu implementiert werden können. Das System solcher Libraries, welche die geforderten Abstraktionen zur Verfügung stellen, heißt **libOS**. Im Folgenden bezeichne eine Anwendung immer das eigentliche Anwendungsprogramm im Verein mit dessen libOS.

Durch die Verschiebung von Abstraktion und Management in die Anwendungsebene ergeben sich folgende Vorteile:

- Nicht benötigte Features können vermieden werden, daher sinkt die Komplexität und steigt die Effizienz, denn es müssen keine Mittelwege gesucht werden, um möglichst vielen Anforderungen gerecht zu werden.
- Implementierung der Abstraktionen kann auf die Anwendung zugeschnitten werden. Dies muß nicht durch einen Kernel-Programmierer geschehen, sondern kann durch den Anwendungsprogrammierer selbst erfolgen.
- Mehrere Testimplementierungen können nebeneinander existieren und leicht gegeneinander ausgetauscht werden um optimale Effizienz zu erreichen.
- Einer schrittweisen Erweiterung der Umgebung ist möglich. Auch kann eine Spezialisierung ohne Aufwand nachträglich durchgeführt werden.
- Fehler in der Implementierung wirken sich höchstens auf eine Anwendung aus, daher ist die Fehlersuche einfacher und durch die Lokalitaet sind Änderungen ohne Anpassung aller anderen Anwendungen möglich.
- Ressourcenmanagement hat alle Anwendungsinformationen verfügbar. Zusätzlich kann auf die kerninternen Informationen wie Seitenkacheltabellen oder dirty-Bits der Speicherseiten zugegriffen werden. Hiermit kann eine optimale Ressourcenverteilung realisiert werden.
- Traditionellen Anwendungen kann die gewohnte Umgebung zur Verfügung gestellt werden. Demzufolge sind Änderungen der Anwendung die über das Linken hinausgehen nicht nötig.

1.3 Realisierung des Schutzkonzeptes

Die Sicherheit der verteilten Ressourcen wird vom Kern dadurch garantiert, daß jede Benutzung einer Ressource mit einem **guard** versehen wird, welcher die Zugriffsrechte überprüft. Zum Beispiel werden zum Schutz der Speicherseiten immer, wenn eine Anwendung eine Zuordnung von virtueller zu physikalischer Adresse in die Page Tables eintragen will, zuvor vom Kern überprüft ob sie das darf. Dies erfolgt durch selbstauthentifizierende **capabilities**, deren Aufbau ebenfalls der Anwendung überlassen bleibt.

Die Zuordnung von Ressourcen zu Anwendungen erfolgt auf Anwendungswunsch hin und kann explizit die gewünschte Instanz einer Ressource beschreiben. So kann eine Anwendung, wenn sie eine Seite aus einer Datei lesen möchte, explizit angeben an welche Stelle ihres Adressraums die Seite gemapped werden soll.

Möchte der Kern einer Anwendung eine Ressource wieder freigeben, so stellt er ihr einen **revocation hint** zu mit der Aufforderung, die Ressource freizustellen. Diese kann sich aussuchen, welche Instanz der Ressource sie freigeben möchte. Beachtet die Anwendung die Aufforderung nicht, so kann der Kern in der zweiten Stufe einen weiteren hint schicken, diesmal mit einer Zeitspanne, in dem die Ressource freigegeben werden muß. Ignoriert die Anwendung auch diesen request so wird ihr nach der festgelegten Zeit die Ressource entzogen. Der Kern lagert hierbei eine Instanz der Ressource aus und benachrichtigt die Anwendung über eine **exception**. Die Anwendung erhält eine capability mit der sie die Ressource wiederfinden kann. Auch kann sie schon vorher für jede Ressource einen Auslagerungsplatz reservieren und so die Speicherung steuern. Damit nicht wichtige Teile der Anwendung wie zum Beispiel der die Ein-/Auslagerung steuernde Programmcode ausgelagert werden, kann die Anwendung einer kleine Zahl von Ressourcen festlegen, welche dann nicht vom Kern freigegeben werden.

Prozessorzeit ist ebenfalls eine Ressource wie jede andere. Sie wird aufgeteilt in einen linearen Vektor dessen Elemente die Zeitscheiben sind. Jede Anwendung hat eine Position auf diesem Vektor zugeordnet, welche die Reihenfolge des Schedulings der Anwendungen angibt und die ungefähre obere Grenze darstellt, bis zu der diese an die Reihe kommen werden. Sie werden im round-robin Verfahren verteilt, wobei auch hier der Anwendung nicht einfach der Prozessor entzogen wird, sondern der Kern stellt genau wie bei jeder anderen Ressource einen revokation hint. Die Anwendung kann dann noch eine festgelegte Anzahl von Zeitscheiben weiterlaufen, bevor ihr der Prozessor endgültig entzogen wird. Im Gegenzug verfallen jedoch Zeitscheiben, auf die diese Anwendung in der Zukunft Anspruch gehabt hätte, wodurch jede Anwendung wieder genau ihren zugestandenen Anteil erhält. Die Anwendung kann bei der Allozierung Angaben machen, welche die Aufteilung der Zeitscheiben beeinflussen können. So kann zum Beispiel ein rechenintensiver Prozess seine Zeitscheiben am Stück anfordern, um den Zeitverlust durch Kontextwechsel möglichst gering zu halten.

2 Zusammenarbeit von Xok und libOS

2.1 Downloadable Code, write predicates, DPFs und ASHs

Aus dem Konzept der Trennung von Schutz und Management folgt das die Kommunikation zwischen dem Exokernel, welcher den Schutz sicherstellt, und dem libOS, welches die Ressourcen verwaltet, hochgradig effizient sein muß um die Performance nicht zu vermindern. Daher wurden Maßnahmen ergriffen um das Zusammenwirken zwischen dem **application level libOS** und dem Kern zu erleichtern. Zum einen kann die Anwendung Code in den Kerneladressraum laden der dann dort abläuft und so Übergänge vom Kern- in den Anwendungsadressraum überflüssig macht. Ein Beispiel hierfür sind **wakeup predicates**, Funktionen die aufgerufen werden wenn die Applikation gerade eine Zeitscheibe zugeteilt bekommen würde. Diese entscheiden ob die Applikation wirklich die Zeitscheibe braucht, oder ob sie noch auf das Eintreten eines Ereignisses wie zum Beispiel das Eintreffen eines Diskblocks warten muss.

Diese predicates werden als **object code** in den Kern geladen und beim Einbringen überprüft. Durch Analyse des Codes wird die Laufzeit beschränkt und nicht erlaubte Befehle zurückgewiesen. Ebenfalls sind **upcalls** von Kernel in die Anwendung möglich, wobei diese nicht gescheduled sein muß. So kann auf zeitkritische Ereignisse mit einem Minimum an Verzögerung reagiert werden. Das Einfügen von Interrupt- oder Exception handlern in den Kern wirkt sich dageben nicht positiv auf die Performance aus, da der Übergang vom Kern in den Anwendungsadressraum bei Xok sehr schnell geschieht.

Ein weiteres Einsatzgebiet von Downloadable Code sind **DPFs** (dynamic packet filters) für Netzwerkpackete. Diese sind in einer eigenen Sprache geschrieben (**vcode**), werden zur Laufzeit übersetzt, wobei verschiedene Optimierungen durchgeführt werden, um die Performance zu steigern. Dies reicht von Umstellen von Codesegmenten bis zum Mischen von verschiedenen Packet Filtern um gleiche Befehle zusammenzufassen. Damit diese jedoch nur die Packete ihrer eigenen Anwendung annehmen muß eine externe Kontrolle existieren, zum Beispiel kann die Installation einem privilegiertem Server vorbehalten bleiben.

Eine weitere Möglichkeit ist die Definition von **ASHs** (application secure handlers), welche weitestgehend frei gestaltet werden können und als Objektcode in den Kernadressraum geladen werden. Hierbei wird die Sicherung durch Codeanalyse bei der Installation und durch

Sandboxing garantiert. ASHs haben Zugriff auf den Adressraum der Anwendung und über hinzugelinkte Funktionen können die Daten der Netzwerkpackete eingesehen werden. Bei der Einbindung in den Kern wird auch hier aggressiv optimiert, bis zum Pipelining von mehreren Handlern um ganze Protokollstacks aufzubauen. ASHs haben auch die Möglichkeit Nachrichten zu versenden und so schnell auf eingehende Packete zu reagieren. Da durch sie die gesamte Protokollabwicklung implementiert werden kann sind keine Kopiervorgänge mehr erforderlich um eine Nachricht zuzustellen. Diese wird direkt vom Nachrichtenpuffer über die Protokollpipeline in den bereitgestellten Anwendungspuffer übertragen. Ein mehrfaches Durchlaufen der Nachricht in verschiedenen Protokollebenen entfällt.

2.2 Garantie der Filesystemkonsistenz - XN

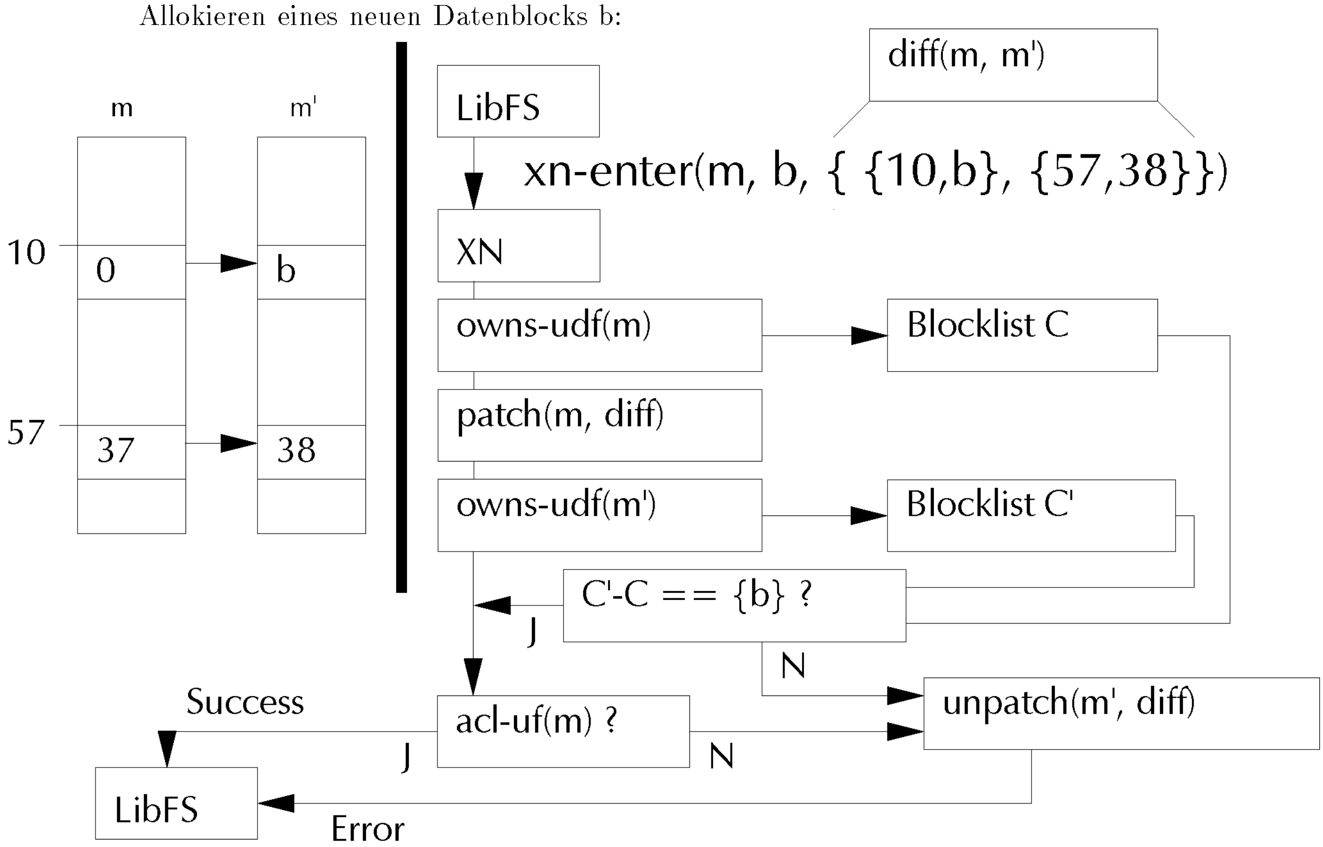
Ein Filesystem ohne globalen Controller konsistent zu halten, wenn einzelne Applikationen böswillig oder fehlerhaft sein können, ist nicht einfach. Der derzeitige Ansatz ist schon der vierte Versuch die zahlreichen Probleme eines **dezentralen Filesystems** in den Griff zu bekommen. Da die Resultate jedoch äußerst ermutigend sind, sollen hier der Aufbau und die Designentscheidungen etwas genauer betrachtet werden.

Da man ohne globalen Zustand wohl nicht auskommen wird stellt sich die Frage wie man den Zustand so im Kern ablegen kann, daß der eigentliche Aufbau des Filesystems immer noch der Anwendung überlassen bleibt, aber andererseit sämtliche **Metadaten** gesichert werden. Eine Leitentscheidung dabei war, daß die Einführung neuer Filesysteme keine privilegierte Operation sein darf. Es muß den Anwendungen freistehen eigene Strukturen zu verwenden. Trotzdem müssen Schutzrechte auf den Daten erzwungen werden. Auch müssen im Sinne einer akzeptablen Gesamtperformance Metadaten und Dateiblöcke gemeinsam und gleichzeitig genutzt werden können. Hierbei sollten Caches zwischen den Anwendungen kohärent gehalten werden. Auch soll das Speichersystem möglichst nahe der Hardware operieren.

All diese Bedingungen werden durch kernelinterne Tabellen gerade residenter Diskblöcke und durch eine **template**-ähnliche Beschreibung der Metadaten ermöglicht. Zu jeder Metadatenstruktur werden in einer RISC-ähnlichen Sprache Funktionen definiert, die **XN**, das System der kernel-level Zugriffskontrollen, benutzen kann um gerade die Informationen über die Metadaten zu ermitteln, die es braucht, um die Sicherung zu ermöglichen. Dies geschieht durch **UFs** (untrusted functions). Diese operieren nur lesend auf den Metadaten und haben keine besonderen Kernrechte. Drei solche Funktionen bilden das template für jeden Typ von Metadaten. Dies sind im einzelnen die **acl-uf**, welche die Zugriffsbestimmungen regelt, die **size-uf**, welche nur die Größe der Metadaten ermittelt, und die **UDF** (untrusted deterministic function) **owns-udf**, welche die Zustandsübergänge auf den Metadaten beschreibt, die für XN interessant sind. Diese liefert angewendet auf die Metadaten eine Liste aller Blöcke mit ihrem Typ, welche in der Metadatenstruktur gespeichert sind. Letztere Funktion muss deterministisch sein, also nur von den eigentlichen Metadaten abhängen. Diese Eigenschaft wird auch per Codeanalyse zur binding time überprüft.

Neue Metadaten werden definiert nur indem dem Kern diese drei Funktionen bekannt gemacht werden. Sobald diese in den globalen root-catalog eingetragen sind sind sie unveränderbar und können auch nicht wieder entfernt werden. Um nun einen Block zu lesen muß die Anwendung einen registry-Eintrag für den Block und alle seine parents eintragen lassen, wobei sie eine Adresse aus ihrem eigenen Adressraum angibt, an die der Block gemapped werden soll. Daraufhin kann sie eine Leseanforderung starten. Bei dieser werden zuerst die acl-uf des Blocks und die seines Parents überprüft. Ergeben diese das Leserechte vorliegen, so wird der Block umgehend an die gewünschte Stelle übertragen. Wichtig ist hier das der Eintrag des Blocks und seiner Parents erfolgen kann, bevor die erste Leseoperation angestossen wird. So können dann alle Blöcke auf einmal eingelesen werden, was bei entsprechender Anordnung nur einen Plattenzugriff erfordert.

Allokieren eines neuen Datenblocks b:



Soll ein neuer Block alloziert werden, so muß der Zeiger b auf diesen Block zwangsläufig in eine schon vorhandene Metadaten-struktur m eingetragen werden.

- LibFS ruft XN mit m, b und diff(m, m')
- XN ruft owns-udf(m) und merkt sich die Ausgabe
- XN erzeugt aus m und diff die neuen Metadaten m'
- XN ruft owns-udf(m') und vergleicht mit Ausgabe von Schritt 2. Die beiden Resultate dürfen sich nur durch den neu hinzugekommenen Block b unterscheiden.
- Durch acl können nicht nur Zugriffsrechte ermittelt werden sondern auch sichergestellt werden das z.B. die inode modification time aktualisiert wird
- Wenn ja, fertig
- Wenn nein, mache Änderung an m' rückgängig, Fehler

XN liefert noch weitergehende Garantien:

- Zurückschreiben erfolgt in einer Reihenfolge, so das Metadaten nicht geschrieben werden, wenn sie noch Zeiger auf uninitialisierte Metadaten enthalten (tainted blocks). Ausnahme: temporary filesystem, unattached subtree.
- Die Wiederbenutzung eines Blocks wird aufgeschoben bis alle persistenten Zeiger auf diesen entfernt wurden (reference count).
- Der Disk-Block cache ist konsistent (globale, geschützte cache map/status-Tabelle im Kern). Cache-Seiten selbst liegen im Anwendungsraum
- Eintragung in die cache-registry kann erfolgen ohne das der Block selbst im Speicher ist und ohne das dessen Typ bekannt ist (ermöglicht prefetch)
- Cache-locks für atomare Updates
- Jeder Prozess kann beliebige cache-Seiten zurückschreiben (flushd)
- Die Zugriffsberechtigung wird beim mapping kontrolliert (secure binding). Der eigentliche Zugriff kann dann mit den normalen Speicherschutzmechanismen gesichert werden. Rohdaten werden bei entsprechenden Rechten read/write gemapped, Metadata immer readonly, denn die Modifikation muß über XN erfolgen.

3 Vergleich mit anderen Systemen

3.1 Vorteile

Benchmark	Aegis (μs)	Ultrix (μs)
System Call	4.7	32.2
Exception dispatch	3.0	238.0
IPC - Pipes	30.9	326.0
IPC - Shared Memory	12.4	187.0
Matrix Multiplication 150x150	7.0	7.1
VM - protect 1	32.5	51.6
VM - protect 100	213.0	175.0
VM - trap	13.9	240.0
Roundtrip latency Ethernet	259	3400

Die Testergebnisse sprechen sowohl bei der Kombination Aegix/ExOS auf DEC Workstations als auch bei Xok/ExOS auf Intel-PPro-Basis für sich.

- Unveränderte Unix-Anwendungen zeigen vergleichbare oder bessere Laufeigenschaften gegenüber traditionellen Betriebssystemen.
- Durch Anpassung des libOS ist auch ohne Modifikation der Anwendung erhebliches Tuning möglich.
- Hochspezialisierte Anwendungen können eine Performancesteigerung um bis zu 800% erreichen, wie zum Beispiel der implementierte zero touch cp (Faktor 3) welcher die Daten kopiert ohne die Ein-/Ausgabepuffer kopieren zu müssen, und der Cheetah http

server (Faktor 8), welcher die Dateien direkt in den Ausgabepuffer liest und die einkommenden Protokollheader weiterverwendet. Eine weitere Optimierung sieht für kleine Dateien eine Äenderung des TCP/IP-Stacks vor, wobei die ACK-Antwort auf das Request-Packet für kleine Seiten verzögert wird, bis die Seite gelesen ist und dann mit der Antwort verschickt wird. Dies soll Netzkapazität in der Höhe von bis zu 20% freisetzen. Hierbei zeigt sich wie wie einfach neue Konzepte in die Struktur des Exokernel-Systems eingebracht werden können.

- Weitere Experimente ergaben, daß eine Emulation des Unix-Kerns mit minimalen Geschwindigkeitseinbußen möglich ist. Ein erneutes Linken gegen das libOS ist dabei nicht notwenig.
- Auch zeigen die Vergleiche zwischen Aegis und Xok, daß nicht das Tuning des Systems wichtig ist, sondern die Kontrolle von zeitaufwendige Operationen durch die Anwendung.

3.2 Nachteile

- Das Exokernel-System erfordert flexible Abstraktionen. Diese sind in der Regel nicht einfach zu finden. Auch sind high level abstractions schwer zu schützen, ohne die Implementierung vorzugeben und ohne der Anwendung die Kontrolle zu entziehen.
- Trennung von Schutz und Management erfordert viel Überlegung und Bedacht bei der Wahl der Schnittstellen.
- Eine dezentrale Ressourcenverwaltung kann potentiell Einbrüche bei der Gesamtperformance hervorrufen. Auch hier ist überlegtes Design notwenig um Engpässe zu vermeiden.
- Böswillige Applikationen können Systemperformance erheblich drücken. Dies ist aber auch bei den meisten traditionellen Betriebssystemen nicht anders.
- Globale Systeminformationen können nicht immer vermieden werden. Trotzdem ist die durch Synchronisation eingebüßte Performance bei typischen Anwendungen gering gegenüber den Einsparungen.
- Portabilität kann problematisch werden, wenn man nicht im libOS entsprechend früh abstrahiert.
- Ohne shared libraries steigt die Größe der Executables ins Uferlose.
- Paging der libOS ist kompliziert zu realisieren, wird aber spätestens, wenn mehrere Versionen gleichzeitig ablaufen sollen, ratsam.

Literatur

[1] Dawson R. Engler, M. Frans Kaashoek, James W. O'Toole Jr. *The Operating System Kernel as a Secure Programmable Machine*
http://www.pdos.lcs.mit.edu/papers/osr-exo.ps

[2] Dawson R. Engler, M. Frans Kaashoek. *Exterminate All Operating System Abstractions*
http://www.pdos.lcs.mit.edu/papers/hotos-jeremiad.ps

[3] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, Kenneth Mackenzie. *Application performance and flexibility on exokernel systems*
http://www.pdos.lcs.mit.edu/papers/exo-sosp97.html

[4] Deborah A. Wallach, Dawson R. Engler, M. Frans Kaashoek. *ASHs: Application-specific handlers for high-performance messaging*
http://www.pdos.lcs.mit.edu/papers/ash-sigcomm96.ps

[5] Dawson R. Engler, M. Frans Kaashoek. *DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation*
http://www.pdos.lcs.mit.edu/papers/dpf.ps

[6] Dawson R. Engler, Sandeep Guptar, M. Frans Kaashoek. *AVM: Application-level Virtual Memory*
http://www.pdos.lcs.mit.edu/ engler/avm.ps

[7] Dawson R. Engler. *vcode: a retargetable, extensible, very fast dynamic code generation system*
http://www.pdos.lcs.mit.edu/ engler/vcode-pldi.ps

[8] Dawson R. Engler. *The Design and Implementation of a Prototype Exokernel Operating System*
http://www.pdos.lcs.mit.edu/ engler/msthesis.ps