Seminarausarbeitung zum Thema
Interprozesskommunikation
Effizienz und Sicherheitsrisiken

verfasst von Eyad Alkassar
betreut von Sebastian Bogan

7. Oktober 2004

Inhaltsverzeichnis

Teil I
Einleitung

Betriebssystem mit millionenzeiligem Code, unüberblickbarer Komplexität und explosionsartig steigendem Wartungsaufwand führen immer häufiger zu hohen Ausfallraten und kostspieligen Fehlern. Der Ruf nach Betriebssystemen mit hoher Modularität wird daher immer lauter.
So genannte Mikrokernel verfolgen genau dieses Ziel. Im Gegensatz zu bekannten Betriebssystemen wie Windows oder Linux, mit einer Monolith-Architektur wird versucht den Kern des Betriebssystems, den sogenannten Kernel möglichst dünn zu halten: viele Dienste außerhalb laufen zu lassen, um so höhere Stabilität und Konfigurierbarkeit durchzusetzen. Die ersten Mikrokernel-Implementierungen, der bedeutendste unter ihnen MACH, enttäuschten jedoch im Vergleich zu herkömmlichen Systemen mit ihrer schlechten Effizienz. Diese Schwäche ließ sich vor allem an der Interprozesskommunikation (IPC) festmachen, die sich in Mikrokernel als erheblich kostenaufwändiger erwies. Viele sahen daher IPC als prinzipielle Effizienzbarriere für den neuen Ansatz, die nicht zu überwinden sei. Geschwindigkeit von IPC musste also als wichtigstes Kriterium für die weitere Entwicklung gelten. Liedtke schaffte es mit L4 die Interprozesskommunikation so weit zu optimieren, dass sein Mikrokernel verglichen mit monolithischen Kernels in Effizienz konkurrenzfähig wurde [1].
Optimierung darf jedoch nicht auf Kosten der Sicherheit des Gesamtsystems gehen. J. Shapiro beschreibt mögliche Denial of Service (DOS) Attacken gegen Mikrokernels.[3]
Wie Liedtke in L4 diese Optimierungen schafft und welche Sicherheitsrisiken sich aus dieser Architektur ergeben bilden den Kern dieses Artikels. Er gliedert sich wie folgt auf.
Eine kurze Einführung und Begriffsbildung zu Interprozesskommunikation und Mikrokernel wird Kapitel 2 geben. In Kapitel 3 werde ich die interessantesten Optimierungen beschreiben, den Liedtkes L4-Kernel seine Geschwindigkeit unter anderem verdankt. Sicherheitsrisiken, die der Mikrokernelansatz im Allgemeinen und L4 im Speziellen birgt und mögliche Lösungen werden in Kapitel 4 diskutiert.

Teil II
Mikrokernel und IPC

Das hier beschriebene Modell ist nur eins von vielen möglichen und beschreibt die Umgebung von Liedtkes L4 Implemenierung. Die Abstraktion besteht im Wesentlichen aus drei Komponenten: Dem Kernel, den Adressräumen und IPC.

1 Kernel

Nebenläufigkeit ist ein wichtiger Aspekt des Kernels: Viele Prozesse laufen quasiparallel, und müssen sich in der Ausführung ständig gegenseitig abwechseln. Die Verwaltung von Ressourcen und die Kommunikation zwischen zwei Prozessen bzw. zwischen Prozess und Kernel sind also seine grundlegenden Aufgaben.
Liedtke bennent in seinem Paper [2] die notwendigen und hinreichenden sieben Funktionalitäten, die ein Kernel zur Verfügung stellen muss. Sie lassen sich in folgende Kategorien unterteilen:

- **Erzeugung neuer Tasks**
- **Speicherverwaltung**
- **IPC-Aufrufe**
- **Scheduler-Funktionalität**

2 Prozesse und Adressräume

Prozesse werden in Adressräumen modelliert. Ein Adressraum ist eine Abbildung von Virtuellen Adressen auf physikalische. Jeder Prozess verfügt über einen eignen Adressraum. Ein Prozess kann seinen Adressraum auch mit anderen teilen, in dem er die betreffenden Adressen in einen anderen Adressraum reinprojeziert. Dieser Vorgang wird als Mapping bezeichnet. Beim Mapping kann der ursprüngliche Besitzer den gemeinsamen Speicher auch wieder zurückverlangen. Diese Anfrage wird transitiv an alle beteiligten Prozesse weitergereicht. Das sogenannte Granten hingegen gibt Adressen unwiderruflich an einen anderen Prozess weiter.

3 Pager

Die Verwaltung der Adressräume übernehmen sogenannte Pager. Pager sind dem Mikrokernelansatz folgend ebenfalls Benutzerprozesse, welche anderen Prozessen von ihrem Speicherbereich Platz zuweisen. Benötigt zum Beispiel ein bestimmter Prozess mehr Speicherplatz, löst er einen Page-Fault-Interrupt aus. Der zugehörige

Pager reagiert darauf durch das Mappen weiterer Adressen. Der elementarste Pager, wird auch Systempager genannte und besitzt als Adressraum die Idendität auf die physikalischen Adressen.

4 IPC

Da verschiedene Prozesse in verschiedenen, in der Regel auch in disjunkten Adressräumen arbeiten, können sie nicht direkt über den Speicher miteinander kommunizieren, wie es der Fall bei monolithischen Betriebssystemen ist. Daher wird ein Kommunikationsprotokoll benötigt, welches der Kernel implementiert. Dieses verfügt in der Regel über drei verschiedene Primitive, also Funktionsaufrufe:

- **send** Senden einer IPC-Nachricht an einen spezifizierten Prozess. Dabei stehen mehrere Möglichkeiten der Datenübertragung zur Verfügung.
- **open-receive** Open-receive wartet auf eine Nachricht eines beliebigen Empfängers. Typischerweise implementieren Server (Prozesse, die Dienste anbieten) dieses Primitiv.
- **closed-receive** Closed-received wird aufgerufen, wenn der Prozess den Empfang einer Nachricht eines bestimmten Empfängers erwartet und alle anderen einkommenden Daten ignoriert. Typischerweise findet man dieses Primitiv in Client-Implementierungen.

Dabei unterscheidet man grob zwei gegensätzliche Konzepte: Synchrone und asynchrone Übertragung. Bei der Synchronen Übertragung blockiert der Sender solange, bis der Empfänger ein receive Primitiv ausführt. Analog blockiert der Empfänger solange bis ein send Primitiv an ihn gerichtet wird. Im asynchronen Modell wird hingegen beim Senden die Nachricht in einen Buffer geschrieben (eventuell selbst ein Prozess), der zu einem beliebigen Zeitpunkt vom Empfänger gelesen werden kann. Im folgenden Betrachten wir nur synchrones IPC (Ausnahme letztes Kapitel), da deren Anwendung bedeutend effizienter ist als die asynchrone Variante.// Synchrones IPC bietet drei verschiedene Möglichkeiten zur Datenübertragung:

- **Kopieren von Daten/ Call-By-Value** Dazu spezifizieren der Sender und der Empfänger Quell- bzw. Zieladressen in den jeweiligen Adressräumen.
- **Mapping von Adressräumen/ Call-By-Reference** Wie oben beschrieben. Das Send-Primitiv legt die zu veröffentichen, virtuellen Adressen fest und das Receive-Primitiv spezifiziert über welche eigenen, virtuellen Adressen gemappt werden soll.
- **Verwendung von Registern** Kurze Nachrichten können auch direkt über Register versendet werden.

5 Threads und TCBs

Threads eines Prozesses sind die unabhängig laufende Berechungen innerhalb eines Adressraums. Oftmals verfügt ein Prozess nur über einen einzigen Thread. Die Kommunikation zwischen Threads benötigt keine IPC, da sie direkt auf dem Speicher Nachrichten austauschen können. Wechselt der Kernel zwischen verschiendenen Threads (bzw. der Scheduler: ein Programm, dass diese Aufgabe übernimmt) muss der aktuelle Zustand des abgelösten Threads im Kernel gespeichert werden. Dazu wird ein sogenannter Thread Control Block (TCB) angelegt, der die folgenden Informationen unter anderem trägt:

- **Kernel Stack** Der Kernel legt für die Berechnung innerhalb eines Threads jeweils einen eigenen Thread an (in L4).
- **Register** Alle Register die beim Threadwechsel gelöscht oder überschrieben werden.
- **Status** Hier kann z.B. gespeichert werden, ob ein Prozess gerade im Empfangszustand (also blockierend) ist. Unter anderem greift darauf der Scheduler zu (siehe Kapitel III Queues).

Wie IPC wirklich schnell gemacht werden kann, werde ich im folgenden Kapitel beschreiben.

Teil III
Effiziente Implementierung

6 Einleitung

Alle beschriebenen Konzepte wurden in der dritten Version von L3 implementiert. Die gemessenen Zyklenzahlen beziehen sich dabei auf die Ausführung auf einer 386/ 486-Architektur.
Im Folgenden werde ich immer zuerst die naive Implementierung beschreiben, gefolgt von der Vorstellung einer möglichen Optimierung.
Wo aber liegt die prinzipielle Grenze für die Geschwindigkeit von IPC-Aufrufe? Liedtke versucht diese Frage mit der Analyse der einfachsten übertragenen Nachricht zwischen zwei Prozessen zu beantworten. Alle IPC Aufrufe sind mindestens so teuer. Dieser einfachste Vorgang, besteht lediglich im Versenden einer leeren Nachricht von Prozess A zu Prozess B. Folgende Operationen werden dabei mindestens auf der Seite von Prozess A ausgeführt:

- lade ID von B

- setze Nachrichtenlänge zu 0
- rufe Kernel auf

Der Kernel reagiert mit folgenden Operationen:

- greife auf tcb von B zu
- wechsle Adressraum
- lade ID von A
- rufe Benutzerprozess B auf

Wechsel in den Kernel bzw. in Benutzerprozessen sind die beiden teuersten Operationen. Die hohen Kosten entstehen z.B. durch das Abspeichern des Prozesszustandes im Speicher des Kernels oder durch den Wechsel des Adressraums.
Liedtke kommt in seiner Analyse auf 172 benötigte Zyklen, was in seiner Testumgebung 3.5 μs entspricht. Die genauen Zahlenwerte sind für uns nur zum Vergleich weiter relevant.

7 Kombinierte Systemaufrufe

Das Beispiel macht klar, dass Wechsel von Benutzerprozess zum Kernel bzw. umgekehrt die teuersten Operationen während der Interprozesskommunikation sind. Diese müssen daher so gut es geht vermieden werden. Dazu betrachten wir das am häufigsten vorkommende Nachrichtenschema bei der Kommunikation zweier Prozesse:

- **Aufruf ipcsend**
  Prozess A stellt Anfrage an Prozess B
- **Aufruf ipcclosereceive**
  Prozess A geht in Wartezustand
- **Aufruf ipcsend**
  Prozess B empfängt Nachricht
- **Aufruf ipcopenreceive**
  Prozess B sendet antwort an Prozess A

Insgesamt kommen wir in diesem Beispiel auf vier Kernel/ Benutzer Wechsel durch IPC-Aufrufe.
Die Optimierung, fällt sehr einfach aus. In diesem Aufrufschema folgt auf jedes Senden eines Prozesses ein direktes Empfangen desselbigen. Kombiniert man beides zu einem erhält man für Prozess A das Primitiv ipc-call bestehend aus ipc-send und ipc-closereceive und für Prozess B ipc-replywait, bestehend aus ipc-send und ipc-openreceive. Damit reduzieren wir die Kernelaufrufe von vier in der naiven Variante auf zwei mit unseren neuen Primitiven.

8 Kopieren großer Daten

Interprozesskommunikation dient, wie oben beschrieben nicht nur Kommunikation, sondern auch der Datenübertragung. Eine effiziente Möglichkeit wäre die Zusicherung von virtuellen Adressen über die Speicheroperationen Map oder Grant. Sollen jedoch tatsächlich Daten kopiert werden, ist eine effiziente Methode nicht mehr offensichtlich.

8.1 Naive Implementierung

Ein naiver Ansatz würde wie folgt verfahren: Wenn Prozess A Prozess B Daten schicken will wird zunächst ein gemeinsamer Bereich des Kernelspeichers in die beiden Adressräume von A und B geladen. Prozess A kopiert dann die entsprechenden Daten in diesen überlappenden Zwischenspeicher. Der Kernel liest nun diese aus und kopiert sie in den von B angegebenen Bereich in seinem Adressraum. Diese Methode funktioniert, benötigt jedoch ein zweifaches Kopieren: Adressraum A ▷ Kernelspeicher ▷ Adressraum B. Die gewünschte Fassung sollte mit einer einzigen Kopieroperation auskommen.

8.2 Gemeinsamer Adressraum

Mit nur einer Kopierinstruktion kommt die folgende Variante aus. Der Empfängerprozess teilt für die Dauer der Kommunikation mit dem Sender den Bereich in seinem Adressraum, in dem er die Daten empfangen will. Der Sender und nicht mehr der Kernel kopiert dann die entsprechenden Daten in den gemeinsamen Bereich. Dieser Ansatz birgt jedoch Sicherheitsrisiken:

- **Keine Integrität**
  Der Empfänger kann die Integrität der Daten nicht feststellen. Nach einem testen der Nachricht auf eine bestimmte Eigenschaft, könnte der Sender die versandten Daten wieder ändern.

- **Keine Richtlinien**
  Die Implementierung von Multilevel Sicherheitsrichtlinien (wie Bell/LaPadula) ist nicht mehr umsetzbar.

8.3 Optimierung

Liedtke präsentiert eine andere Lösung des Problems. Zum Kopieren von Daten aus dem Adressraum von Prozess A erzeugt der Kernel einen temporären, eigenen Adressraum und bildet den Zielbereich vom Adressraum des Prozesses B auf seinen ab. Jetzt kopiert er die Daten direkt in die den gemeinsamen Bereich. Dieser Ansatz kommt mit nur einer Kopierinstruktion aus. Gleichzeitig muss der Kernel durch einen eigenen Adressraum, die Umrechnung von virtuellen Zieladressen zu

physikalischen nicht kostenintensiv selber durchführen, sondern kann es der entsprechenden Hardware überlassen.

9 IPC Timeouts

In L4 können vor jedem IPC-Aufruf zwei Timeoutwerte t1, t2 festgelegt werden. Beide Werte werden in ms interpretiert. t1 definiert die Zeit, die ein aufrufender Thread bereit ist zu warten bis die Gegenseite reagiert (bei ipcsend oder ipccloserevceive). Beim Auftreten eines Pagefaults wartet ein Thread t2 ms auf die Antwort des entsprechenden Pagers. Beim Ablaufen einer der beiden Timeoutwerte wird die Kommunikation abgebrochen. Die beiden am häufigsten benutzten Werte sind null und unendlich. Deren Implementierung ist recht einfach.
Werden hingegen andere Werte benutzt, benötigt man eine Aufweckqueue. Da in den meisten Fällen der Timeout nicht abläuft, muss die Datenstrukturen auf das Löschen und Einfügen an beliebigen Positionen in der Queue optimiert sein.

9.1 Naive Implementierung

Löschen und Einfügen in konstanter Zeit kann z.B. durch ein Array realisiert werden. Die entsprechenden Timeoutwerte werden durch die ThreadId adressiert. Die Schwäche dieser Lösung ist offensichtlich: in jedem Zeitintervall muss das komplette Array nach abgelaufenen Werten durchsucht werden. Das führt zu einer nicht akzeptablen Laufzeit (auch bedingt durch das Einlesen des Arrays in den Cache).

9.2 $\frac{k}{n}$ Listen

Die folgende Datenstruktur reduziert den benötigten Aufwand um eine Konstante n. Dazu werden n unsortierte Aufwecklisten (doppelt verkettet) angelegt. Für einen neuen Timeoutwert t wird die Aufweckzeit x berechnet (aktuelle Zeit + t) und am Ende der Liste x $\mod$ n eingefügt. Jetzt wird in jedem Zeitintervall T nur noch die Liste T $\mod$ n überprüft.
Threads mit Timeoutwerten die noch weit in der Zukunft liegen, können als weitere Optimierung in eine gesonderte Winterschlafliste eingefügt werden, die nur in größeren Zeitabständen untersucht wird.

Sei k die Gesamtzahl aller Threads. Die erwartete Länge für eine der n Listen ergibt sich zu $\frac{k}{n}$. In jedem Zeitintervall werden nur noch $\frac{k}{n}$ Einträge betrachtet. Das Einfügen und Löschen von neuen Einträgen bleibt in konstanter Zeit möglich.

10 Andere Queues

Der Kernel führt Buch über den Status der Threads. Dazu benutzt er folgende drei Warteschlagen

- **Busy Queues** Threads, die blocken. Z.B. closesreceive aufgerufen haben ohne bisherige Reaktion.
- **Ready Queues** Threads, die zur weiteren Ausführung bereit sind.
- **Polling-me Queues** Für jeden Thread wird eine solche Queue geführt. In ihr stehen alle Threads, die Thread me senden wollen und daher blockieren.

IPC Aufrufe verändern die Zustände der aufgeführten Queues, für ipccall bzw. für ipcreplyandwait werden vier derartige Operationen benötigt (polling-me Queue hier außer acht gelassen)

- Lösche Sender aus der Ready Queue
- Füge ihn in die Busy Queue ein
- Lösche empfangenden Thread aus der Busy Queue
- Füge ihn in die Ready Queue ein

10.1 Lazy Scheduling

Bei Lazy Scheduling wird versucht möglichst alle Lösch- und Einfügeoperationen zu vermeiden, dafür nur die Statuseinträge in den Threadkontrollblöcken zu verändern. Queues sollen nur noch die beiden Invarianten erfüllen:

- Mindestens alle bereiten Threads bis auf den gerade ausgeführten müssen in der Ready Queue enthalten sein.
- Mindestens alle wartenden Threads müssen in der Busy Queue enthalten sein.

D.h. es können jetzt auch Threads in der Busy Queue auftauchen, die ready sind und umgekehrt busy Threads in der Ready Queue. Diese fehlerhaften Einträge werden immer beim Durchgehen der Queue durch den Scheduler nach dem Lesen des Status im Kontrollblock gelöscht.
Durch eine triviale Optimierung können Einfügeoperationen bei den Primitiven ipcCall und ipcReplyandWait vermieden werden. Offensichtlich ist der Zustand aller Queues nach der kompletten Ausführung einer der beiden genannten Primitve wieder wie vor der Ausführung.

10.2 Analyse von Lazy Scheduling

Der Vorteil dieser Konstruktion liegt auf der Hand: IPC führt nicht mehr zu direkten Löschoperationen auf der Queue. Einfügeoperationen werden nur noch durch das Primitiv ipcsend verursacht.
Ein Nachteil von Lazy Scheduling ist die Möglichkeit, dass ein Thread in mehreren Queues gleichzeitig auftreten kann. Diese Eigenschaft könnte die Verbesserungen, die wir durch $\frac{k}{n}$-Listen erhalten haben nutzlos machen, da im schlimmsten Fall alle Threads in allen Queues auftauchen.

11 Auswertung der Optimierung

Durch das hier beschriebene IPC Design und einigen weiteren Verbesserungen kommt Liedtke bei IPC von kurzen Nachrichten (acht Byte) auf eine Zeit von 5 μs verglichen mit der unteren Schranke von 3.5 μs. Dieser Zahlenwert, so Liedtke stellt eine Verbesserung um den Faktor 22 dar verglichen mit Mach IPC.
Bei kurzen Nachrichten (acht Byte) führen die aufgeführten Optimierungen zu folgenden Effizienzverbesserungen gegenüber Systemen mit naiver Implementierung: Lazy Scheduling 23 Prozent. Kombinierten Systemaufrufe ipccall und ipcreplyandwait 18 Prozent. Das einfache Kopieren von Datenblöcke (4096 byte IPC) schafft sogar nach Liedtke eine Zeitverkürzung um 157 Prozent.

Teil IV
Sicherheitsschwächen in IPC

Shapiro stellt in seiner Veröffentlichung drei Sicherheitskriterien auf, die jedes Kerneldesign gleichzeitig erfüllen müsse:

- **Assymetrisches Vertrauen** Das Modell darf nicht voraussetzen, dass ein Server einem Client trauen darf. Nur Server sind vertrauenswürdig.
- **Reproduzierbarkeit des Verhalten** Das Verhalten der IPC Primitive darf nicht vom Systemzustand (Rechnerauslastung, freier Speicher, etc.) abhängig sein, und muss in allen Fällen wiederholbar sein.
- **Dynamische Größe** Die Implementierung der Sicherheitsanforderungen darf nicht derartig restriktiv sein, dass die Funktionalität darunter leidet. Das Versenden von IPC Nachrichten dynamischer Größe muss also weiterhin unterstützt werden.

Dabei zeigt er auf, dass L4 nur zwei der drei Ziele erreicht und stellt als alternative Lösung EROS vor, ebenfalls ein Mikrokernel. Im Weiteren werde ich nicht genau

auf die EROS-Implementierung eingehen, sondern nur mögliche Lösungen anhand dieses Ansatzes beschreiben.

12 Assymetrisches Vertrauen

Viele Anewndungen, die über synchrones IPC realisiert werden, laufen im Client/ Server Modell ab. Ein Server ist hierbei ein Prozess, der bestimmte Dienste anbietet. Der Client kann diese über IPC anfordern. Dabei sind die Ressourcen des Servers beschränkt. Vertrauen bedeutet in diesem Kontext, dass ein Prozess erwarten kann, dass die Gegenseite das Kommunikationsprotokoll gemäß der Spezifikation erfüllt. Assymetrisch ist das Vertrauen, wenn voraussgesetzt wird, dass der Server vertraunswürdig ist, nicht aber der Client. Über diesen können bis auf das IPC-Design keine Annahmen betreffend seines Verhaltens getroffen werden.

12.1 Ein einfacher DOS-Angriff

Ein Client, darf also nicht in der Lage sein den Server zu blockieren. Folgendes Aufrufprotokoll beschreibt einen einfachen Denial-Of-Service (DOS) Angriff:
Der bösartige Client schickt eine Anfrage an den Server, der sich in einem open-wait Zustand befindet. Ein gutartiger Client würde jetzt in einen closed-wait Zustand übergehen um die Antwort des Servers zu empfangen. Folgt er dieser Spezifikation jedoch nicht, blockiert der Server, da seine Nachricht synchron übertragen werden muss. Damit kann er keine weiteren Anfragen mehr entgegennehmen.

Vier denkbare Anätze kommen als Lösung des Problems in Betracht:

12.1.1 Verwendung von Buffer

Der Kernel könnte die vom Server verschickte Antwort in einen eingenen Buffer speichern und so den Prozess wieder aufwecken. Offensichlich ist dies keine wirkliche Alternative. Durch Buffer wird zwar eine Denial-Of-Service Attacke auf den Server verhindert, dafür sind jetzt schon einige wenige bösartige Clients in der Lage einen Denial-Of-Ressource (DOR) Angriff auf den Kernel zu starten, in dem sie möglichst viel von seinem Speicher alloziieren. Buffer verlagern demnach eine lokale Sicherheitslücke auf ein potentielles, globales Ressourcen-Problem. Andererseits entspricht dieser Ansatz dem asynchronen IPC, wie es zum Beispiel in Mach implementiert wurde. D.h. alle gewonnen Optmimierungen im synchronen IPC beschrieben im vorherigen Kapitel wären nicht anwendbar.

12.1.2 Verwendung von Multithreading

Warum nicht einfach die Ressourcen des Servers erhöhen. Durch Einführung von Multithreading könnte der Prozess mehrere Anfragen gleichzeitig verwalten, auch

wenn ein Teil bösartiger Natur ist. Jene würden nur den betreffendne Thread blockieren und nicht mehr das gesamte System. Auch dies ist keine sinnvolle Lösung. Der einfachste Weg, den bösartige Prozesse gehen könnten, um diesen vermeintlichen Sicherheitsmechanismus auszuhebeln ist: auch sie erhöhen ihre Kapazizät, es werden soviele Prozesse gestartet, wie Threads für einen Server zur Verfügung stehen. Dadurch ergibt sich wieder eine ähnliche Problematik, wie wir sie schon bei Kernel Buffern gesehen haben: Wir wandeln eine lokale Schwäche in eine globale um. In L4 zum Beispiel wird die Konfiguration eines Threads in den TCBs im Kernelspeicher abgelegt. Dadurch wird ein Angriff auf den Kernel wieder möglich. Gleichzeitig wird der Scheduler in seiner Arbeit durch die Verwaltung der blockierendne Threads eventuell eingeschränkt. Die Implementierung von Multithreading in Servern führt des Weiteren hohe Komplexität mit sich, und damit Effizienznachteile.

12.1.3 Weitere Vorgehensweise

Anhand der oben beschriebenen Ansätze können wir zusammenfassen, welche Prinzipien beim Design einer effektiven Lösung unseres Problems befolgt werden müssen. Diese lassen sich bis auf den zweiten Punkt auch auf andere DOS-Attacken übertragen:

- **Unterschätze die Ressourcen bösartiger Benutzer nicht!** Das Design darf nicht auf eine reine Erhöhung der Ressourcen setzen, der Angreifer kann mit der gleichen Maßnahme antworten.
- **Lösche Feuer nicht mit Öl!** Die Sicherheitslücken eines Prozesses dürfen nicht auf Kosten der Stabilität des gesamten Systems geschlossen werden.
- **Effizienz ist (immer noch) alles!** Ein Ansatz der die Optimierungen von L4 zum Beisiel obsolet macht, würde dem Mikrokernel seine Berechtigung wieder absprechen, wie ich in der Einleitung erläutert habe.
- **Wer Dienste fordert, trägt die Beweislast!** Eine Lösung darf nicht eine erhöhte Komplexität auf Serverseite mit sich bringen. Im Gegenteil, der Client muss in der Lage sein sich als gutartiger Benutzer zu authentifizieren. Gleichzeitig muss der Server nur beweisbar korrekten Clients seine Dienste der Spezifikation folgend anbieten.

12.2 L4: Benutzung von Timeouts

Die Lösung in L4 respektiert die vier oben erwähnten Prinzipien durch die Verwendung von Timeoutmechanismen. Dabei kann ein bestimmtes Intervall bei jeder IPC-Kommunikation gesetzt werden. Typischerweise benutz der Client beim Senden und Empfangn vom vertraunswürdigen Server den Wert unendlich. Der Server hingegen mißtraut dem Benutzer und fordert eine direkte Antwort, er setzt

seinen Wert auf Null. Er geht also davon aus, dass bei einer Verzögerung der Client bösartig ist und beendet die Kommunikation. Je nach Anwendung können auch andere Werte spezifiziert werden. In diesem Fall, erfüllt L4 nicht mehr alle drei am Anfang beschriebenen und von Shapiro aufgestellten Sicherheitskriterien. Das Verhalten der einzelnen Primitive kann jetzt je nach Implementierung des Scheduler von der Prozessorauslastung abhängen. Ein weiterer praktischer Nachteil bei der Verwendung von Timeouts ist, dass Debugging wesentlich aufwendiger wird.

12.3 EROS: Prompte Antworten

Prompte Antworten in EROS entsprechen im Grunde Timeouts mit Wert null in L4. Darüber hinaus wird durch einen weiter unten beschriebenen Mechanismus sichergestellt, dass Clients immer direkt bereit sind die Antwort des Servers zu empfangen, ohne Fehler zu produzieren.

13 Der Pager Angriff

Obwohl beide IPC-Implementierungen in L4 und EROS einen akzeptablen Schutz gegen das Einführungsbeispiel geben, bleibt eine weitere Schwäche durch promptes Anworten ungelöst: Der Angriff mittels bösartigen Pagern, die als Benutzerprozesse implementiert werden. Folgendes Szenario: Ein bösartiger Benutzerprozes versendet Daten an einen Server mittels einer Kopieroperation, d.h. nur die virtuellen Adressen werden von ihm spezifiziert. Liegen diese jedoch nicht innerhalb des Adressraums des Senders, wird ein Page-Fault-Interrupt ausgelöst, der ebenfalls mittels IPC zur Behandlung an den Pager des Senders und nicht den des Empfängers gesendet wird. Da dieser ebenfalls ein Benutzerprogramm ist, könnte er die Anfrage böswillig ignorieren. Da die IPC-Nachricht nicht zu Ende ausgeführt werden kann, führt dieses Verhalten zum Blockieren des Senders und des Empfängers. Der Angriff war erfolgreich. Das gleiche Fehlverhalten, kann auch durch das Spezifizieren von ungültigen Empfangsadressen provoziert werden.

13.1 L4: Pager Timeouts

L4 setzt in diesem Fall wieder auf Timeouts. Unabhängig vom ersten Timeoutwert kann ein zweiter sogenannter Pager-Timeout bei jedem IPC spezifiziert werden. Dieser legt die Zeit in Millisekunden fest, die ein Prozess auf die Page-Fault Behandlung der Gegenseite bereit ist zu warten. Er beruht auf die erwartete Zeit zur Behandlung eines derartigen Interrupts, ist also nicht gleich null. Dieser Ansatz widerspricht unserem ersten Prinzip bei der Implementierung von Sicherheitsmaßnahmen: mehrere bösartige Benutzer könnten immer noch nacheinander einen Dienstausfall provozieren.

13.2 EROS: Probeläufe

Die von Shapiro beschriebene Lösung hingegen, ist wesentlich eleganter. Hier zwingt das IPC Design den Benutzer vor jedem Senden und Empfangen zu einem Probelauf auf seinen angegebenen Adressen. In dieser Phase können auftretende Interrupts von gutartigen Benutzern behandelt werden. Kommt es hingegen nach dem Probelauf und während der Ausführung des IPC-Primitivs zum Pagefault wird ein Angriff offensichtlich und die Kommunikation direkt abgebrochen. Damit wird gemäß unserer Desing-Prinzipien die Komplexität auf den Client ausgelagert. Im Gegensatz zur L4-Lösung geht also die Behandlungszeit nicht auf Kosten des Servers, DOS-Angriffe sind nich mehr möglich.

14 Dynamische Datengröße

EROS Ansatz birgt jedoch ein weiteres kritisches Problem: das Empfangen von Daten unbekannter Größe durch den Client. In diesem Fall kann kein Probelauf durchgeführt werden, da noch nicht bekannt ist welche und wieviele Adressen zum Speichern benötigt werden. Kommen die folgenden drei Bedingungen zusammen, benötigen wir einen neuen Mechanismus zur Übertragung:

- **Der Typ des IPC-Aufrufs erlaubt kein Blockieren** Das ist immer dann der Fall, wenn der vertrauenswürdige Sender dem potentiell bösartigen Client Daten schicken will.
- **Dynamische Datengrößes** Die Größe der versandten Daten kann nicht im Voraus bestimmt werden
- **Teure Wiederholung** Mögliche Seiteneffekte oder die Kosten für die dafür notwendigen Operationen machen eine mögliche Erweiterung des Speichers des Clients und die Wiederholung des IPC-Aufrufs zu teuer.

Offensichtlich haben wir dadurch diesen schwierigen Fall auf eine kleine Menge von möglichen IPC-Aufrufen beschränkt. Aufgrund des Timeoutmechanismus tritt dieses Problem in L4 nicht auf, es gewährt dem Pager des Clients je nach Größe der Daten eine gewisse Schonfrist.

14.1 EROS: Einführung von TBOs

EROS setzt auf sogenenannte vertrauenswürdige Buffer (Trusted Buffer Objects kurz TBO). Diese Objekte werden vor dem Empfangen durch den Client aufgerufen und kommunizieren mit dem Server. Dabei darf jetzt der Server blockieren, solange sich die TBOs als vertraunswürdig authentifizieren können. Dabei bezahlt im Gegensatz zum oben beschriebenen Buffer-Ansatz der Client für den benötigten Mehraufwand. Welche Bedingungen muss aber ein derartiges Objekt erfüllen, um als vertraunswürdig zu gelten? Im Grunde nur diese beiden:

- **Idendität** Der Server muss feststellen können, dass er dem TBO antwortet, d.h. auf einen Prozess wartet von dem er sicher sein kann, dass er direkt reagiert.
- **Integrität** Der Server muss überprüfen können, ob das TBO auch tatsächlich zur Ausführung seines Codes kommt und nicht durch einen bösartigen Benutzer daran gehindert werden kann. Möglichkeiten dazu wären ein Angriff auf den Scheduler oder auf den Speicherbereichs des TBOs.

Die Datenübermittlung wird dann in vier Schritten vollzogen:

- **Erzeugung des TBO** Will ein Client Daten vom Server kopieren ruft er zunächst einen bestimmten Code auf, genau das TBO. Dieser schickt die vom Client gewünschte IPC-Anfrage an den Server.
- **Annahme durch Server** Beim Empfangen der Nachricht verifiziert der Server die Vertraunswürdigkeit des Senders. Dadurch weiß er, dass er blockieren kann und, dass alle benötigten Kosten vom Client übernommen werden.
- **Übermittlung der Daten zum TBO** Der Server übermittelt dann die Daten an den TBO. Dieser akzeptiert solange er über Speicher verfügt. Im Falle von Platzmangel wird der Transfer abgebrochen und eine entsprechende Nachricht an den Benutzer versandt.
- **Übermittlung der Daten zum Client** Im letzten Schritt übermittelt der TBO dem Client seine empfangenen Daten.

Auf den genauen Authentifzierungsalgorithmus werde ich hier nicht im Detail eingehen. Näheres dazu findet man unter [3]. Im Grunde entspricht dieser Ansatz der schnellen asynchronen IPC Übermittlung.

15 Schlussbetrachtung

15.1 Zusammenfassung

Wir haben eine wichtige Sicherheitslücke von synchronen IPC gesehen und in L4 oder EROS vorgeschlagenen Lösungen anhand der Veröffentlichung von Shapiro besprochen. Dabei sollen stets alle drei am Anfang formulierten Sicherheitsziele gleichzeitig erfüllt werden. Die möglichen Lösungen haben uns zu einem weitern Problem dem Pager-Angriff geführt. Durch die Diskussion der verschiedenen Alternativen konnten vier grundlegende DOS-Abwehr Prinzipien formuliert werden.

15.2 Kritik

Shapiro schafft es nicht die Vollständigkeit und Notwendigkeit seiner drei Ziele plausibel zu begründen, vorallem ist die Reproduzierbarkeit problematisch. Der

Vorteil dieser Eigenschaft bei der Anwendung liegt zwar klar auf der Hand, nicht aber die Relevanz für die Sicherheit. Andererseits treten die beschriebenen Schwächen nicht in allen synchronen IPC-basierenden Mikrokernelarchitekturen auf. Werden zum Beispiel Kernel-Pager verwendet, die vor Manipulation geschützt sind, bleibt die zweite Angriffsvariante wirkungslos.

Viel wichtiger ist jedoch die Ineffizienz seiner Vorgehensweise. Er hat lediglich die Existenz von Sicherheitslücken gezeigt, nicht aber eine aus den vorgeschlangenen Methoden resultierende Abwesenheit derartiger Probleme. Hierzu benötigt es formaler Verifikation des verwendeten Modells und des Designs.

Literatur

[1] J. Liedtke. *Improving IPC by Kernel Design*. 14th ACM Symposium on Operating System Principles (SOSP), Asheville, North Caorlina, 1993.

[2] J. Liedtke. *μ-Kernels Must And Can Be Small*. 5th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS), Seattle, WA, 1996.

[3] Jonathan S. Shapiro. *Vulnerabilities in Synchronous IPC Desings*. IEEE Symposium on Security and Privacy, Oakland, CA, 2003.