Großer Beleg

Konzepte zur Vermeidung von Denial-of-Service-Angriffen in L4/DROPS

Stefan Lebelt
lebelt@os.inf.tu-dresden.de

7. Dezember 2005

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Ronald Aigner

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 7. Dezember 2005

Stefan Lebelt

Danke

Ich bedanke mich an dieser Stelle bei Prof. Dr. Hermann Härtig für die Möglichkeit, in der Betriebssystemgruppe arbeiten zu dürfen. Besonderer Dank gilt ebenfalls meinem Betreuer Ronald Aigner für seine umfangreiche Beratung und gute Zusammenarbeit, des weiteren Christian Helmut, Normen Feske, Alexander Warg, Udo Steinberg, Jean Wolter und Bernhardt Kauer für die anregenden Gespräche und hilfreichen Kommentare.

Inhaltsverzeichnis

1 Einführung **7**

1.1 Grundlagen . . . 7
1.1.1 Mikrokernkonzept . . . 7
1.1.2 L4 . . . 8
1.1.3 DROPS . . . 9
1.1.4 Denial-of-Service . . . 9
1.2 Angriffsszenarien . . . 10
1.2.1 Angreifermodell . . . 10
1.2.2 Angriffe via IPC . . . 10
1.2.3 Angriffe auf die Speicherverwaltung . . . 11
1.2.4 Angriffe auf die Task-/Threadverwaltung . . . 12

2 Lösungsansätze und relevante Arbeiten **15**

2.1 Grundlegende Ansätze . . . 15
2.1.1 Replikation . . . 15
2.1.2 Kontrollierte Bereitstellung beschränkter Ressourcen . . . 15
2.2 Konkrete Beispiele . . . 16
2.2.1 Propagation . . . 16
2.2.2 Ressourcenmanager . . . 16
2.2.3 Ökonomische Modelle . . . 16
2.2.4 Clans & Chiefs zur Eingrenzung von DoS-Angriffen . . . 17
2.2.5 Endpunkte in L4.sec . . . 17
2.2.6 Kernspeicher in L4.sec . . . 18
2.2.7 Weitere Arbeiten . . . 18

3 Vermeidung von Angriffen via IPC **19**

3.1 Angriff und Abwehr in der Theorie . . . 19
3.2 Abwehr in der Praxis . . . 19
3.2.1 Auswahlstrategie . . . 20
3.2.2 Dienstbeschränkung . . . 20
3.2.3 Time-Slice-Donation . . . 20
3.2.4 Multiple Delegation . . . 22
3.2.5 Priority-Remembrance . . . 24
3.2.6 Zwischenbetrachtung . . . 24
3.2.7 Duale Delegation als Ersatz für Clans & Chiefs . . . 28
3.2.8 Endpunkte . . . 28
3.2.9 Zusammenfassung . . . 29

4 Entwicklung des „Dealman“ am Beispiel Speicher **31**
4.1 Vermeidung von Angriffen auf den Nutzerspeicher . . . 31
4.1.1 Sichere Pager durch bekannte Policies . . . 31
4.1.2 Sichere Pager durch Task-bounded-Swapping . . . 32
4.1.3 Sichere Server durch Memory-Donation . . . 33
4.1.4 Memory-Accounting . . . 33
4.1.5 Memory-Accounting am Beispiel DOpE . . . 44
4.1.6 Zusammenfassung . . . 45
4.2 Der γ-Bank-Manager „Dealman“ . . . 46
4.2.1 Motivation . . . 46
4.2.2 Das erweiterte ökonomische System . . . 46
4.2.3 Neue Möglichkeiten . . . 47
4.2.4 Notwendige Erweiterungen und Optimierungen . . . 48
4.2.5 Integration in DROPS . . . 48
4.2.6 Zusammenfassung . . . 48

5 Nutzung des „Dealman“ für andere Ressourcen **49**
5.1 Vermeidung von Angriffen auf den Kernspeicher . . . 49
5.1.1 Grundlegende Lösung . . . 49
5.1.2 Problem des begrenzten Kernspeichers . . . 49
5.1.3 Zusammenfassung . . . 49
5.2 Task- und Threadverwaltung . . . 50
5.2.1 Der Taskmanager $\tau 0$ . . . 50
5.2.2 Scheduling . . . 51
5.2.3 Verhinderung von DDoS-Angriffen . . . 53
5.2.4 Zusammenfassung . . . 53

6 Zusammenfassung und Ausblick **55**
6.1 Zusammenfassung . . . 55
6.1.1 Richtlinien zur Konstruktion DoS-sicherer Server . . . 55
6.1.2 Ungelöste Probleme . . . 56
6.2 Ausblick . . . 56
6.2.1 Delegationsframework . . . 56
6.2.2 Implementation . . . 56
6.2.3 Anpassung der Basisdienste . . . 57
6.2.4 Untersuchung der Folgen von ökonomischen Systemen . . . 57

A Anhang **59**
A.1 Durchsetzung von Time-Slice-Donation . . . 59

Abbildungsverzeichnis **60**

Tabellenverzeichnis **61**

Glossar **65**

Literaturverzeichnis **69**

1 Einführung

Überall wo Dienste von Servern bereitgestellt und von Clients genutzt werden spielt der Begriff Verfügbarkeit eine Rolle. Verfügbarkeit bedeutet, dass Informationen dort und dann zugänglich sind, wo und wann sie von Berechtigten gebraucht werden [Pfi].

Denial-of-Service-Attacken haben das Ziel die Verfügbarkeit von Dienstanbietern zu beeinträchtigen. Einer der ältesten DoS-Angriffe ist das Mail-Bombing [mai]. Dabei wurde die Adresse eines Empfängers zusätzlich mehrfach im BCC-Feld der Email eingetragen. Der Mailserver, der diese Email zustellen sollte, musste nun eine Vielzahl an Emails an ein und den selben Empfänger generieren und verschicken, war damit überlastet und für eine gewisse Zeit nicht verfügbar.

Mikrokernbasierte Betriebssysteme implementieren grundlegende Dienste wie Speicherverwaltung und Treiber durch Serverthreads im Userland. So ist also hier das klassische Client-Server-Muster zu finden. Die Anforderungen an die Verfügbarkeit dieser Server sind hoch. Ist ein Pager nicht bereit, wenn einer seiner Clients einen Pagefault generiert, so verzögert sich die Ausführungszeit des Clients, was möglicherweise weitere unschöne Konsequenzen nach sich zieht.
Dieser Beleg befasst sich ausführlich mit der Entwicklung von Konzepten, die Denial-of-Service-Attacken in L4-basierten Betriebssystemen unterbinden sollen. Es werden Angriffsmöglichkeiten aufgezeigt und grundlegende Maßnahmen besprochen. Diese Maßnahmen werden aufgegriffen und Schritt für Schritt umgesetzt. Am Ende steht ein System zur Verwaltung beschränkter Ressourcen das weit mehr Möglichkeiten bietet als die Begrenzung von DoS.

1.1 Grundlagen

Dieses Kapitel gibt einen kurzen Überblick über L4, DROPS und Denial-of-Service, beschränkt sich dabei aber auf die für diesen Beleg wesentlichen Konzepte und Problematiken. Bei der Konstruktion von Methoden zur DoS-Abwehr wird von der L4V2-Spezifikation und Ein-Prozessor-Systemen ausgegangen.

1.1.1 Mikrokernkonzept

Mikrokerne stellen im Gegensatz zu monolitischen Betriebssystemkernen nur grundlegende Dienste wie Adressräume, Aktivitäten und Kommunikation zur Verfügung. Treiber, Speicherverwaltung und Ähnliches können als Server im Userland implementiert werden.
Eine solche Architektur hat folgende Vorteile gegenüber monolitischen Systemen:

- Die Trusted Computing Base kann sehr viel kleiner gehalten werden.
- Es ist eine stärkere Trennung zwischen den verschiedenen Systemkomponenten möglich.

- Durch die Kompaktheit des Kerns ist es einfacher ihn zu verifizieren

Dem Mikrokernansatz folgend, sollte bei Erweiterungen und der Entwicklung neuer Dienste prinzipiell zunächst versucht werden, diese im Userland und ohne Erweiterung des Kerns zu realisieren.

1.1.2 L4

L4 ist ein Mikrokern der zweiten Generation, das bedeutet, dass er nicht, wie z. B. Mach, durch den Umbau eines ehemals monolitischen Kerns, sondern von Grund auf als Mikrokern konstruiert wurde. L4 ist ursprünglich das Werk von Jochen Liedtke [Lie98].
Der Kern wird in Form von Michael Hohmuths Fiasco [Hoh] an der TU-Dresden eingesetzt und weiterentwickelt.

Tasks

Ein fundamentales Sicherheitskonzept sind Adressräume, also voneinander isolierte Speicherbereiche. Diese werden in L4 durch Tasks repräsentiert.
Tasks können aktiv oder inaktiv erzeugt werden [AH99]. Inaktive Tasks sind lediglich TaskIDs und repräsentieren das Recht, die Task aktiv zu erzeugen. Das geschieht mittels des Systemrufs `task_new`. Aktiv Erzeugen bedeutet, dass ein Adressraum und 128 zugehörige Threads angelegt werden, von denen genau einer zunächst aktiv ist.

Threads

Threads sind die Aktivitäten in L4. 128 Threads gehören zu einer Task und können mit dem Systemruf `lthread_ex_regs` aktiviert werden.

IPC

Während die Kommunikation zwischen Threads der selben Task über deren gemeinsam genutzten Speicher trivial ist, trifft das auf Inter-Prozess-Kommunikation (IPC) naturgemäß nicht zu. Taskübergreifende Kommunikation ist nur mit Hilfe des Mikrokerns möglich. IPC ist in L4 synchron. Das bedeutet, dass Sender und Empfänger zum Zeitpunkt der IPC dafür bereit sein müssen. Es werden im Kern keine Nachrichten gepuffert. Neben Short- und Long-IPC, bei denen jeweils tatsächlich Daten direkt oder indirekt von einem Adressraum in einen anderen kopiert werden, gibt es mit dem Flexpage-IPC auch die Möglichkeit, Teile des physischen Speichers anderen Tasks zu mappen und so gemeinsam genutzte Speicherbereiche zu erzeugen. Diese Technik bildet die Grundlage für Paging auf Nutzerebene.

Clans & Chiefs

Um die Kommunikation zwischen Tasks kontrollieren zu können gibt es das Konzept der Clans & Chiefs [LJI97]. Dabei gehört jede Task zu einem Clan. Jeder Clan besitzt genau einen Chief. Die Tasks eines Clans können beliebig via IPC miteinander kommunizieren, während clanübergreifende IPC nur über die jeweiligen Chiefs möglich ist.

L4.sec

L4.sec ist eine experimentelle L4-Schnittstelle, die neue Sicherheitskonzepte einführt. Insbesondere Endpunkte (Kapitel 2.2.5) und die Verwaltung des Kernspeichers (Kapitel 2.2.6) sind wichtige Neuerungen.

1.1.3 DROPS

Das „Dresden Realtime OPerating System“ ist ein echtzeitfähiges Betriebssystem, das auf dem Mikrokern Fiasco basiert. Ressourcen werden in DROPS von Ressourcenmanagern verwaltet [HRW+99]. Diese Ressourcenmanager ermöglichen es, Echtzeit- und Nicht-Echtzeit-Tasks parallel auszuführen.

1.1.4 Denial-of-Service

„Als DoS-Angriff (Denial of Service attack, etwa: Dienstverweigerungs-Angriff) bezeichnet man einen Angriff auf einen Host (Server) mit dem Ziel, einen oder mehrere seiner Dienste arbeitsunfähig zu machen. In der Regel geschieht das durch Überlastung. Erfolgt der Angriff koordiniert von einer größeren Anzahl anderer Systeme aus, so spricht man von einem DDoS (Distributed Denial of Service).“ (wikipedia - [dos])

Denial-of-Service-Attacken sind also Angriffe auf die Verfügbarkeit von Diensten.

Man unterscheidet zwischen solchen Angriffen, die Sicherheitslücken, Programmfehler, u. s. w. ausnutzen und solchen, die durch übermäßige Inanspruchnahmen des Dienstes dessen beschränkte Ressourcen auslasten. In diesem Beleg wird nur die zweit genannte Variante betrachtet.

DDoS

Distributed-Denial-of-Service-Angriffe bilden eine Spezialform von DoS. Dabei wird eine Infrastruktur vieler (verteilter) Angreifer benötigt, die zu einem bestimmten Zeitpunkt gemeinsam koordiniert angreifen können. Das größte Problem an DDoS-Attacken ist, dass es schwer ist diese als solche zu erkennen und damit fast unmöglich darauf zu reagieren.

Beschränkte Ressourcen

Beschränkte Ressourcen sind Betriebsmittel, von denen nicht unbegrenzt viele Exemplare existieren. Sie entstehen durch bestimmte Rahmenbedingungen wie CPU-Rechenzeit, Speichergröße, maximale Anzahl der Tasks im System, u. s. w. Daraus resultieren Größen wie z. B. die maximale Anzahl von Anfragen, die pro Zeiteinheit bearbeitet werden können.

1.2 Angriffsszenarien

In diesem Kapitel werden die festgestellten grundlegenden Probleme in Form von Angriffsszenarien vorgestellt.

1.2.1 Angreifermodell

Von folgenden Annahmen über die Stärke des Angreifers wird ausgegangen:

- Es ist möglich jegliche Programme im Userspace unter den von L4/DROPS vorgegebenen Rahmenbedingungen laufen zu lassen.
- Es können keine Sicherheitslücken oder Programmfehler des Systems ausgenutzt werden.
- Es ist keine Manipulation an der Hardware möglich.

1.2.2 Angriffe via IPC

IPC erfolgt in L4 synchron. Dazu gibt es zwei mögliche Empfangsstatus: `receive` und `wait`. `Receive`, auch geschlossenes Warten genannt, bedeutet, dass der Empfänger auf eine Nachricht eines bestimmten Threads wartet. `Wait`, das offene Warten, erlaubt es Nachrichten von jedem beliebigen Thread zu empfangen. Der Status `reveive` lässt sich im Allgemeinen nicht für DoS-Angriffe missbrauchen, da ein Thread A_1 nur dann in diesen Zustand übergeht, wenn er auf die Nachricht eines konkreten, ihm bekannten Threads B_1 wartet, dem er i. d. R. vertraut.

Allgemeiner Floddingangriff

Im Gegensatz zu `reveive` ist der Zustand `wait` problematisch. Dabei kann jeder beliebige Thread X_i dem Wartenden A_1 eine Nachricht schicken, auf die A_1 reagieren muss. Dieser Umstand kann missbraucht werden, indem ein Angreifer den Wartenden A_1 mit Nachrichten überflutet. Der Angriff kann sogar zu einem DDoS-Angriff erweitert werden, indem X_i zunächst eine Menge von Threads erzeugt, die den Wartenden A_1 attackieren. Benötigt A_1 für die Bearbeitung dieser Nachrichten mehr als eine Zeitscheibe, so ergeben sich Totzeiten, Zeiten, in denen der Thread nicht empfangsbereit ist und so von anderen nicht erreicht werden kann. Ist A_1 nicht empfangsbereit, so werden anfragende Threads vom Kern in die Senderwarteschlange von A_1 eingereiht und blockieren, bis der Sendetimeout abgelaufen ist oder der Empfänger A_1 empfangsbereit wird. Genau genommen ist der allgemeinen Floddingangriff somit ein Angriff auf die IPC-Warteschlange (Kapitel 1.2.2).

DDoS Angriff auf die IPC-Warteschlange

In L4V2 ist die Warteschlange als dynamisch wachsende und schrumpfende Liste implementiert. Damit ist es nicht möglich, sie mit einem allgemeinen Floddingangriff komplett zu füllen. Da ein Element dieser Liste nur wenig Speicher belegt und die Anzahl der Threads pro Task auf 128 begrenzt ist (Kapitel 1.1), kann auch die Möglichkeit, auf diese Weise den Speicher auszulasten, vernachlässigt werden. Außerdem ist das ein Problem der Speicherverwaltung (Kapitel 4.1).

Wartende Threads werden vom Kern nach dem FIFO-Prinzip (First In First Out) für die IPC ausgewählt, sobald der Empfänger wieder empfangsbereit ist. Vorausgesetzt, der Angreifer ist in der Lage Threads zu starten, ist damit folgender Angriff möglich:

- Der Angreifer startet eine Menge von Threads, die jeweils Anfragen an den Server stellen.
- Die Bearbeitung der ersten Anfrage benötigt eine gewisse Zeit (mehr als eine Zeitscheibe).
- Die nächsten Threads werden in die Warteschlange eingereiht und schieben deren Ende immer weiter nach hinten.
- Ein legaler Dienstnutzer muss nun warten, bis alle Anfragen der Angreifer, die vor ihm in der Warteschlange stehen, bedient wurden.
- Diese Situation wird permanent, wenn die Anfragen der Angreiferthreads fortlaufend wiederholt werden.
- Je mehr Angreifer es gibt, desto länger wird die Warteschlange, desto länger werden die Totzeiten.

„Stehlen“ von Prioritäten

Ein weiteres Problem besteht, wenn ein Server mit einer anderen Priorität als der Client läuft. Die Diensterbringung erfolgt dann mit dieser anderen Priorität. Davon kann ein Angreifer profitieren, indem er viele Anfragen an einen höherprioren Server schickt, der so häufig aktiv ist und niederpriore Threads verdrängt.

1.2.3 Angriffe auf die Speicherverwaltung

Den nächsten hier betrachteten Angriffspunkt bildet der Speicher. Schwerpunkt soll dabei auf drei wesentliche Angriffe gelegt werden.

Angriff auf die Mapping-Datenbank

Zur Verwaltung von Speicher-Mappings verwaltet der L4V2-Kern pro physischer Speicherkachel einen Mapping-Baum, der bei verschiedenen Operationen traversiert werden muss. Die beschränkte Ressource Kernspeicher führt damit zu zwei Problemen:

1. Wird der Baum größer, dann steigt der Aufwand des Traversierens.
2. Wird der Baum zu groß, dann belegt er viel Speicher.

Es stellt sich die Frage, ob es mit vertretbarem Aufwand möglich ist, einen Mapping-Baum übermäßig zu füllen. Die Antwort darauf lautet: Ja.
Für den Angriff werden mehrere kollaborierende Threads unterschiedlicher Tasks benötigt, was voraussetzt, dass der Angreifer neue Tasks erzeugen und Threads starten kann.

Ablauf:

- Der Angreifer A_1 erzeugt zunächst eine neue Task B
- A_1 mappt nun eine Flexpage an verschiedene Stellen in den Adressraum von B
- Ist der virtuelle Speicher von B „verbraucht“ erzeugt A_1 eine neue Task.

Dieses „Spiel“ spielt A_1 solange, bis der Kernspeicher vollständig mit Mapping-Einträgen gefüllt ist und der Kern anhält.

Angriff auf Pager

Die Verwaltung des virtuellen Speichers erfolgt im Userland mit Hilfe von hierarchischen Pagern. Bei einem Pagefault aktiviert der Kern den Pager des auslösenden Threads. Dieser läuft dann zunächst auf der Zeitscheibe dieses auslösenden Threads und behandelt den Pagefault. Da Pagefaults zu jedem beliebigen Zeitpunkt auftreten können, bleibt dem Pager im Mittel die halbe Zeitscheibe des auslösenden Threads. Reicht diese Zeit nicht aus, so wird bei der nächsten Aktivierung des Pagers auf dessen Scheduling-Kontext gewechselt, da der auslösende Thread blockiert ist.

Es gibt zwei grundlegende Angriffsmöglichkeiten auf Pager.

IPC-Angriff:
Da jeder Pager ein Server ist, kann der im Kapitel 1.2.2 beschriebene Angriff auf die IPC-Warteschlange durchgeführt werden, sofern die Behandlung von Pagefaults im Mittel länger als eine halbe Zeitscheibe dauert. Der Angreifer kann in diesem Fall immer wieder Pagefaults auslösen, die der Pager behandeln muss, und so den beschriebenen negativen Effekt erzielen. Läuft der Pager mit erhöhter Priorität wird zudem das „Stehlen von Prioritäten“ (Kapitel 1.2.2) relevant.

Speicherauslastung:
Jedem Pager steht ein Teil des physischen Speichers zu Verfügung, den er bei Bedarf den entsprechenden Threads mappt. Ein Angreifer, der künstlich einen großen Bedarf erzeugt, kann so den kompletten Speicher seines Pagers erhalten. Andere Threads, die den gleichen Pager nutzen, können so beschränkt werden.

Angriff auf den Speicher von Servern

Ein weiteres grundlegendes Problem besteht, wenn Server für jede Diensterbringung neuen Speicher benötigen und diesen dynamisch allokieren. So steigt, z. B., der Speicherbedarf eines Namensservers, der seine Daten mit Hilfe von verketteten Listen verwaltet, mit jedem neuen Eintrag. Ein Angreifer kann durch ständiges Beanspruchen der Serverdienstleistung dessen Speicher füllen.

1.2.4 Angriffe auf die Task-/Threadverwaltung

Jeder Thread gehört genau zu einer Task. Diese kann mehrere Threads enthalten. Alle Threads einer Task laufen im gleichen Adressraum und können so über den gemeinsamen Speicher Daten austauschen.

Aktivitätsbomben

Aktivitätsbomben sind das Äquivalent zu den aus der Unixwelt bekannten Forkbomben. Forkbomben können ein System durch die massenhafte Erzeugung von neuen Prozessen lahmlegen. Die einfachste Variante `while(1){fork();}` ist dabei zugleich die gefährlichste, da alle neu erzeugten Prozesse ihrerseits ebenfalls neue Prozesse erzeugen. Innerhalb kürzester Zeit ist so die maximale Anzahl an Prozessen erreicht, sodass keine weiteren erzeugt werden können. Außerdem wird das System durch den Verwaltungsaufwand stark belastet.
Eine Aktivitätsbombe unter L4 kann folgendermaßen aussehen:

- Ein böswilliger Thread erzeugt alle möglichen Tasks (siehe Kapitel 1.1.2).
- Nun starten alle Threads dieser neuen Tasks (im Falle von L4V2 sind das 128/Task) und führen komplexe Berechnungen durch.

Das Ergebnis ist das gleiche wie bei der beschriebenen Forkbombe. Da die maximale Anzahl an Tasks und Threads erreicht ist, können keine weiteren gestartet werden. Durch die „komplexen Berechnungen“ ist die CPU stark belastet. Auch dieser Angriff ist nur dann möglich, wenn der Angreifer Tasks und Threads erzeugen b. z. w. starten kann.

Distributed Denial-of-Service (DDoS)

Nach dem gleichen Prinzip wie Aktivitätsbomben lassen sich DDoS-Attacken durchführen, indem die erzeugten Threads gemeinsam koordiniert angreifen (siehe Kapitel 1.2.2).

Missbrauch von Time-Slice-Donation

Ein etwas komplexerer Angriff ergibt sich durch die Möglichkeit Rechenzeit an andere Threads abzugeben. Die entsprechenden Mechanismen werden im Kapitel 3.2.3 näher erläutert und zum Teil zur Verhinderung/Eingrenzung anderer Angriffe genutzt. Mit Hilfe von Time-Slice-Donation kann Einfluss auf das Scheduling genommen werden, indem der Angreifer viele Threads erzeugt, die nichts anderes tun, als diesem ständig ihre Zeitscheibe zu donieren. Damit bekommt der Angreifer mehr Rechenzeit, da er vom Scheduler öfter aktiviert wird.

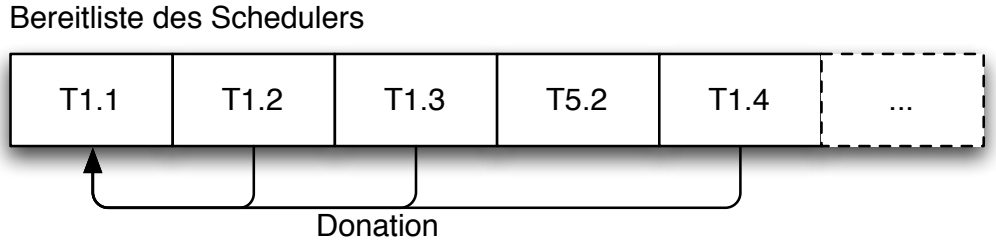

Abbildung 1.1: Beispiel für einen möglichen Missbrauch von Time-Slice-Donation

Das Beispiel in Abbildung 1.1 zeigt, dass der Thread T1.1 dreimal mehr Rechenzeit bekommt als der Thread T5.2, da dieser immer dann aktiviert wird, wenn der Scheduler die CPU T1.1, T1.2, T1.3 oder T1.4 zuteilt. Dabei wird davon ausgegangen, das jeder Thread bei Aktivierung konstant die gleiche Zeitscheibe zugeteilt bekommt.

2 Lösungsansätze und relevante Arbeiten

In diesem Kapitel werden grundlegende Lösungsansätze und Arbeiten anderer Autoren vorgestellt, die für die entwickelten Konzepte und Mechanismen herangezogen wurden.

2.1 Grundlegende Ansätze

Zwei grundlegende Möglichkeiten, Denial-of-Service-Problemen zu begegnen, sind die Replikation von Ressourcen und deren kontrollierte Bereitstellung. Distributed-Denial-of-Service-Attacken lassen sich nur vermeiden, indem es unmöglich gemacht wird, eine entsprechende Infrastruktur zur Koordination mehrerer Angreifer zu nutzen.

2.1.1 Replikation

„If an object gets to many requests, replicate it“ sagte Andrew Tannenbaum in seinem Vortrag „The design of a billion-user worldwide distributed system“ den er am 23.05.2005 vor ca. 500 Studenten und Mitarbeitern der TU-Dresden hielt.

Replikation ist eines der Hauptarchitekturmerkmale von Globe [glo, PCT03] und wird zum Beispiel in [SPvS04] zur Konstruktion von skallierbaren Webanwendungen herangezogen.

Einschätzung

Replikation ist ein guter und offenbar funktionierender Ansatz. Bei aller Euphorie muss aber gesagt werden, dass sie nur in ganz bestimmten Fällen eingesetzt werden kann, und zwar genau dann, wenn die betreffende Ressource replizierbar ist. Das ist bei verschiedenen Informationsdiensten (z. B. Namensdienste), die von Servern angeboten werden, der Fall, nicht aber bei Ressourcen wie Hauptspeicher oder Rechten. Auch die Ressource CPU kann in bestimmten Fällen als replizierbar angesehen werden. Beispielsweise dann, wenn Prozesse von einem Knoten zu einem anderen migrieren können.

2.1.2 Kontrollierte Bereitstellung beschränkter Ressourcen

Klassische, nicht replizierbare Ressourcen wie Haupt- und Hintergrundspeicher oder Zugriffsrechte müssen auf andere Art und Weise vor Denial-of-Service-Angriffen geschützt werden. Hier bleibt nur die Möglichkeit, die Vergabe und den Entzug selbiger intelligent zu verwalten. In der Regel wird dazu eine kontrollierende Instanz benötigt. Eine solche Instanz existiert zum Beispiel in der Theorie in Form des Konzeptes Clans & Chiefs (Kapitel 2.2.4) für die Ressource IPC und in Form von Pagern für die Ressource Hauptspeicher.

2.2 Konkrete Beispiele

2.2.1 Propagation

L4V2 bietet keine direkte Unterstützung für Multi-threaded-Server. In L4 Version X [Lie99] wurde daher ein neuer Mechanismus, der Propagation genannt wird, eingeführt. Dabei „propagiert“ ein Multi-threaded-Server Anfragen an einen „Arbeiterthread“, ändert den Empfangsstatus des anfragenden Clients derart, dass dieser nun auf die Antwort des „Arbeiterthreads“ wartet.

Einschätzung

Propagation erscheint geeignet, um die Ressource Server replizierbar zu machen, und es stellt damit eine Möglichkeit dar, DoS Angriffen via IPC (Kapitel 1.2.2) zu begegnen. Allerdings bringt Propagation einen serverseitigen Mehraufwand mit sich. Im Kapitel 3.2.4 dieses Belegs wird deshalb ein ähnlicher Mechanismus, ohne diesen Mehraufwand, entwickelt.

2.2.2 Ressourcenmanager

In [Not02] wird ein einheitliches System zur Ressourcenreservierung und -freigabe für DROPS entworfen und implementiert. Kernkomponente sind dabei dessen Ressourcenmanager. In der Arbeit werden unter anderem ökonomische Modelle angesprochen, aber auf Grund deren Komplexität nicht angewendet.

Einschätzung

Ressourcenmanager werden sich später (Kapitel 4.2.5) als Schnittpunkt zur Integration des, in diesem Beleg vorgestellten, Ressourcen-Accounting, in DROPS herausstellen. Dabei sind eine einheitliche Ressourcenmanagerschnittstelle und ein einheitliches Ressourcenbeschreibungsformat [Not02] von großem Vorteil.
In Bezug auf DoS stellt das Ausklammern der ökonomischen Modelle ein Manko dar. Daher werden sie im Rahmen dieses Belegs näher betrachtet und angewendet.

2.2.3 Ökonomische Modelle

Ökonomische Modelle ordnen Ressourcen einen Preis und jeder Anwendung ein Budget zu [Not02, MD88c, MD88b, MD88a]. Im Falle des Speichers heißt das beispielsweise, Tasks bekommen periodisch einen bestimmten Betrag auf ihr Konto gutgeschrieben und müssen davon Miete für den belegten Speicher zahlen. Diese Kosten können sich ändern, je nachdem, wieviel Speicher zur Verfügung steht. Einem Angreifer wird also ziemlich schnell „das Geld ausgehen“, wenn er ohne Freizugeben mehr und mehr Speicher anfordert. In [MD88b] wird dieses System als „Rent-based storage management“ bezeichnet.

Einschätzung

Die marktbasierte Vergabe von Ressourcen erscheint zunächst geeignet, zur „kontrollierten Bereitstellung beschränkter Ressourcen“, hat aber einen entscheidenden Nachteil - sie ist aufwendig. Trotzdem wird diese Idee zu einem späteren Zeitpunkt aufgegriffen und bildet, in

abgewandelter Form, die Grundlage für das im Kapitel 4.1.4 konstruierte und im Kapitel 4.2 erweiterte ökonomische System.

2.2.4 Clans & Chiefs zur Eingrenzung von DoS-Angriffen

Die Autoren von [LJI97] sind der Meinung dass das Konzept Clans & Chiefs letztendlich die Lösung aller DoS-Probleme in L4 sei. Es wird gesagt, die besprochenen Angriffe ließen sich unterbinden oder zumindest abschwächen, wenn die Kommunikation mit gefährdeten Servern ausschließlich über die Chiefs der jeweiligen Clans abgewickelt würden. Diese seien in der Lage entsprechende Sicherheits-Policies durchzusetzen und Anfragen von Client, die diesen widersprechen auszusortieren und nicht zuzustellen.

Einschätzung

Ein grundlegendes Problem ist die Policy selbst. Die Autoren gehen davon aus, dass Angriffe als solche erkennbar sind, da der Angreifer gegen die Policy verstößt. Genau das ist aber möglicherweise nicht der Fall, besonders dann, wenn der Angriff nicht von einem einzelnen, sondern von mehreren Clients erfolgt (DDoS). Dieses grundsätzliche Problem bei DDoS wurde bereits im Kapitel 1.1 erwähnt. Ist die Policy zu schwach, wird DoS nur ungenügend eingedämmt, ist sie zu stark, gehen auch legale Anfragen verloren.
Schwerwiegender sind die neuen Angriffsmöglichkeiten, die sich durch den Einsatz von Clans & Chiefs ergeben. Der Chief stellt wiederum einen Server dar, der angegriffen werden kann. Einem Angreifer wird dadurch möglich, alle Threads seines Clans vom Rest des Systems zu isolieren. Dem zu begegnen, indem jedem einzelnen Thread bei seiner Erzeugung ein eigener Clan & Chief zugeordnet wird, scheitert in der Praxis an den damit verbundenen massiven Leistungseinbußen.

Fazit:
Das Konzept der Clans & Chiefs ist zur Verhinderung/Abschwächung von DoS-Angriffen nur bedingt geeignet.

2.2.5 Endpunkte in L4.sec

Endpunkte ist die Bezeichnung für ein neues Konzept, das mit L4.sec [Völ, Kau05] eingeführt wurde. Es sind Kernobjekte, die man sich als Enden von Kommunikationskanälen vorstellen kann. Ein Server, der vor DoS-Attacken via IPC geschützt werden soll, startet von vornherein mehrere Instanzen von sich selbst, die alle „hinter“ dem selben Endpunkt auf Anfragen warten. Clients stellen ihre Anfragen nun über diesen Endpunkt. Dabei wählt der Kern eine der wartenden Serverinstanzen (Serverthreads) aus und leitet die Anfrage an diesen.

Einschätzung

Endpunkte lösen das betrachtete DoS-Problem nicht. Da der Server von Anfang an festlegt, wie viele seiner Instanzen auf Anfragen warten, besteht weiterhin die beschränkte Ressource. Im Kapitel 3.2.8 werden Endpunkte zur Verbesserung eines zuvor vorgestellten Konzeptes zur Vermeidung von DoS via IPC herangezogen.

2.2.6 Kernspeicher in L4.sec

In L4.sec [Völ, Kau05] gibt es außerdem ein Objekt namens Kernel-Memory. Es ist damit möglich, eine Flexpage vom User-Adressraum in eine Kernseite des Kernadressraum zu konvertieren. Alle Zugriffsrechte des ursprünglichen Eigentümers werden danach zeitweise entzogen, um eine Manipulation von Kernspeicher aus dem Userspace heraus zu verhindern.

Einschätzung

Die Möglichkeit, Nutzerspeicher in Kernspeicher zu konvertieren, ist essentiell für die Vermeidung von bestimmten DoS-Attacken, wie den im Kapitel 1.2.3 beschriebenen Angriff auf die Mapping-Datenbank.

2.2.7 Weitere Arbeiten

Einige weitere Arbeiten werden später, an passender Stelle, angesprochen.

3 Vermeidung von Angriffen via IPC

Hier sollen die im Kapitel 1.2.2 beschriebenen Angriffe genauer betrachtet und mögliche Lösungen entwickelt werden.

3.1 Angriff und Abwehr in der Theorie

Wie in [LJI97] aufgezeigt, verweilt ein Sender B_1 maximal $n\varepsilon$ in der Senderwarteschlange von A_1, wobei ε die Zeit ist, die A_1 im Mittel benötigt um Anfragen zu bearbeiten und n die Länge der Warteschlange darstellt.
Wird A_1 wie im Kapitel 1.2.2 dargelegt mit Anfragen überflutet, so füllt sich seine Senderwarteschlange. Damit werden weitere Anfragen entweder stark verzögert, da der Dienstnutzer ggf. die maximale Zeit $n\varepsilon$ warten muss, oder sogar abgewiesen, wenn die Länge der Warteschlange begrenzt ist.
[LJI97] bietet 3 Möglichkeiten zur Abwehr an:

1. Da diese Attacke nur gegen Server möglich ist, die im Status `wait` auf Anfragen von beliebigen Threads „offen warten“, sollte der Server wenn möglich pro anfragendem Client einen Thread bereitstellen.

2. Die Länge der Warteschlange soll reduziert werden, um die maximale Wartezeit zu reduzieren.

3. Clients sollen in Clans gekapselt werden.

Möglichkeit 1 repräsentiert den beschriebenen Ansatz der Replikation (Kapitel 2.1.1) und wird mit Einführung der „multiplen Delegation“ im Kapitel 3.2.4 umgesetzt.
Möglichkeit 2 stellt einen Kompromiss dar. Indem die maximale Wartezeit $n\varepsilon$ reduziert wird, steigt gleichzeitig die Zahl der abgewiesenen legalen Clients. Zudem werden abgewiesene Clients ihre Anfrage wiederholen, sodass das IPC-Aufkommen im System steigt. Durch Begrenzen der Warteschlange wird eine beschränke Ressource und damit eine neue DoS-Angriffsmöglichkeit geschaffen (Kapitel 1.1).
Möglichkeit 3 eignet sich nur bedingt, wie im Kapitel 2.2.4 bereits beschrieben wurde.

3.2 Abwehr in der Praxis

Da die Senderwarteschlange in L4V2 als verkette Liste implementiert ist, kann sie nicht komplett gefüllt werden. Ihre Länge ist nur durch den Kernspeicher begrenzt. Durch geeignete Kernspeicher-, Task-, und Threadverwaltungsmechanismen und -policies (Kapitel 5.1 und 5.2) kann auch das übermäßige Anwachsen dieser Liste und somit ein Angriff auf den Kernspeicher vermieden werden.

3.2.1 Auswahlstrategie

Ein Lösungsansatz besteht in der Änderung der Auswahlstrategie. Die Anwendung von LIFO (Last In First Out), einer Strategie, die ebenso effizient wie FIFO implementiert werden kann, scheidet aber aus, weil dabei ein legaler Dienstnutzer im ungünstigsten Fall durch die Angreifer in der Warteschlange immer weiter nach hinten geschoben und somit ausgehungert wird.

Es ist denkbar, die Scheduling-Prioritäten auch für IPC zu nutzen und jeweils den Thread mit der höchsten Priorität aus der Warteschlange zu wählen. Hierzu erscheint die Verwendung einer nach Prioritäten sortierten Warteliste sinnvoll. Dabei erhöht sich der Aufwand von $O(1)$ auf $O(\log prio)$, bei Verwendung eines Tries als Warteschlange [Reu05]. Da die Anzahl der Prioritätsstufen in L4V2 auf 256 begrenzt ist, bleibt der Aufwand linear. Dieser Ansatz löst das Problem für Threads gleicher Priorität aber nicht und verringert zudem die IPC-Performance.

3.2.2 Dienstbeschränkung

Weiterhin besteht die Möglichkeit, die maximale Zahl der Anfragen pro Client pro Zeiteinheit zu beschränken. Da das eine Policy ist, sollte es außerhalb des Kerns durchgesetzt werden. Dafür wird eine kontrollierende Instanz benötigt. Clans & Chiefs bietet eine solche Instanz, kann aber, wie im Kapitel 2.2.4 betrachtet, nur bedingt eingesetzt werden. Zudem erscheint es sinnvoll, die Entscheidung über die entsprechende Policy dem Server zu überlassen. Dazu wird im Kapitel 3.2.4 der Mechanismus multiple Delegation, bzw. im Kapitel 3.2.7 die duale Delegation eingeführt.

3.2.3 Time-Slice-Donation

Totzeiten, die den beschriebenen Angriff erst möglich machen, entstehen nur dann, wenn ein Server für die Bearbeitung einer Anfrage mehr als eine Zeitscheibe benötigt. Da Server im Allgemeinen Dienste für andere Threads (Dienstempfänger) erbringen, liegt es nahe, das diese Dienstempfänger dem Server für die Zeit der Diensterbringung ihre Zeitscheibe zur Verfügung stellen.
Der grundlegende Ablauf sieht dabei folgendermaßen aus:

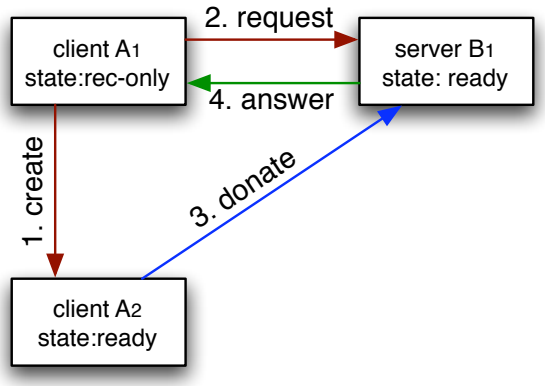

Abbildung 3.1: Time-Slice-Donation

- Client A_1 möchte eine Anfrage an den Server B stellen
- A_1 erzeugt einen neuen Thread A_2 (1. in Abb. 3.1)
- A_1 stellt die Anfrage an B (2. in Abb. 3.1)
- A_2 wechselt nun immer, sobald er vom Scheduler aktiviert wird, mittels `l4_switch_to` zum Server B und übergibt diesem damit seine Zeitscheibe (falls er rechenbereit ist). (3. in Abb. 3.1)
- Hat B die Bearbeitung der Anfrage abgeschlossen, antwortet er dem Client A_1 und wechselt wieder in den Zustand `wait`. (4. in Abb. 3.1)
- A_1 zerstört A_2 und setzt seine Arbeit fort.

Je mehr Clients Anfragen an den Server stellen, umso mehr Rechenzeit bekommt dieser, umso schneller kann er die Anfragen bearbeiten.
Dieser Donation-Mechanismus kommt vollständig ohne Änderungen im Kern aus, hat aber einen entscheidenden Nachteil: Bei transitiver Anwendung, bei der die Zeitscheibe des ersten Clients möglicherweise mehrfach weitergegeben wird, kann es passieren, dass diese durch den Aufwand des Weitergebens abläuft, bevor der letzte in der Kette mit der Bearbeitung seiner Anfrage beginnen kann.

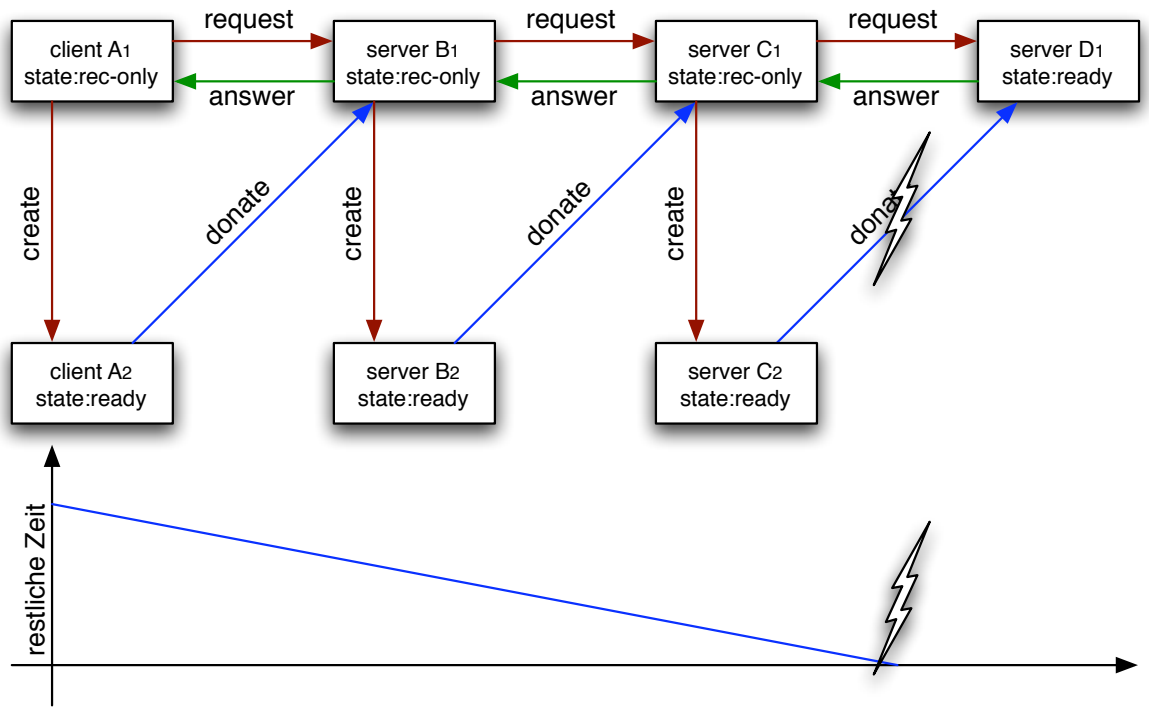

Abbildung 3.2: Donation-Time-Expiration

Diese Situation wird **„Donation-Time-Expiration“** genannt und ist in Abbildung 3.2 dargestellt.

Das kann verhindert werden, indem ein in [Ste04] vorgestellter Mechanismus angewendet wird: „Donating Call“.
Dabei wird bei dem aufrufenden Thread A_1 das `send-donation-flag` gesetzt, was den Scheduler dazu veranlasst beim nächsten Scheduling-Durchlauf die Zeitscheibe des aufrufenden Threads A_1 dem Server B zukommen zu lassen. Dabei geht keine Zeit dieser Scheibe verloren. Der Mechanismus ist transitiv anwendbar.
Der Server kann beim Empfang der IPC selbst entscheiden, ob er die Donation in Anspruch nehmen möchte, oder nicht. Das ist notwendig, da die Funktionalität mancher Dienste durch Time-Slice-Donation gestört wird (z. B. bei Semaphorediensten).
Um Time-Slice-Donation, egal in welcher Form, zur Vermeidung von DoS zu nutzen ist es zwingend notwendig, dass dieser Mechanismus auch wirklich von jedem Client angewendet wird. Dazu muss der Server feststellen können, auf wessen Zeitscheibe er arbeitet. Wird festgestellt, dass es die eigene ist, muss die CPU sofort abgegeben werden. Der Ablauf wird im Anhang A.1 genauer beschrieben.
Die Weitergabe des Scheduling-Kontextes hat aber auch eine Schattenseite. Durch die Möglichkeit anderen Threads die eigene Zeitscheibe zu überlassen sind zwei weiterer Angriffe möglich.
Einerseits kann einem böswilligen Server durch kollaborierende Threads mehr Rechenzeit verschafft werden (näheres dazu im Kapitel 5.2) andererseits ist es einem Angreifer möglich, den Ansatz aus [Ste04] zu nutzen um sehr lange Donationketten zu bauen. So kann, die Effizienz des Schedulers stark beeinträchtigt werden, weil dieser immer dann, wenn der Kopf der Kette ausgewählt werden soll, die gesamte Kette durchlaufen muss. Das Problem lässt sich durch Festlegen einer maximalen Donationkettenlänge eingrenzen. Die Begrenzung der Kette schafft keine begrenzte Ressource im definierten Sinne (Kapitel 1.1) dar, da sie für jede einzelne Kette gilt. Ein anderer Ansatz, der in [SWH05] verfolgt wird, besteht darin, sich den jeweils Letzten einer Donationkette zu merken.

3.2.4 Multiple Delegation

Im Folgenden wird ein grundlegender, an Propagation (Kapitel 2.2.1) angelehnter, Mechanismus, zur Verhinderung/Eingrenzung von DoS-Angriffen via IPC vorgestellt. Die Idee ist eine Umsetzung der in [LJI97] vorgeschlagenen ersten Möglichkeit zur DoS-Abwehr (Kapitel 3.1), pro anfragenden Client einen Serverthread bereit zu stellen, bzw. des allgemeinen Replikationsansatzes (Kapitel 2.1.1).
Dazu wird ein weiterer Syscall benötigt: `send_and_receive_from_task`. Er setzt eine IPC an einen Thread T_i einer Task T ab und wartet auf die Antwort eines beliebigen Threads der Task T.
Ablauf:

- Der Client A sendet via `send_and_receive_from_task` seine Anfrage an den Serverthread B_1 und geht in den Status `receive_from_task` über.
- Der Serverthread B_1 erzeugt einen weiteren Thread B_n und delegiert die Anfrage an diesen.
- Der neue Thread B_n bearbeitet die Anfrage und sendet das Ergebnis an den wartenden Client A.

Der ursprüngliche Serverthread übernimmt damit nur noch die Arbeit eines Vermittlers. Diese grundlegende Lösung ist nur dann sinnvoll, wenn die Beschränkung auf 128 Threads aufgehoben wird, weil die durch den Server repräsentierte Ressource sonst beschränkt bleibt.

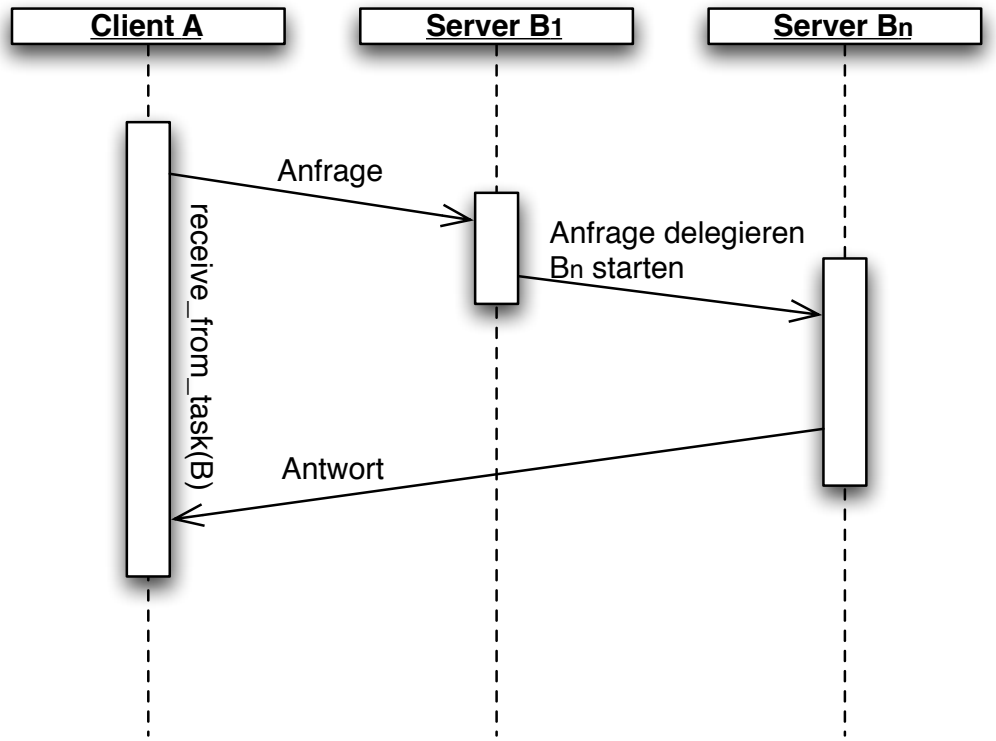

Abbildung 3.3: Delegation

Delegationszeitproblem

Des weiteren muss sichergestellt werden, dass B_1 für das Erzeugen von B_n und die Anfragedelegation nicht mehr Zeit benötigt, als bis zum Eintreffen einer neuen Anfrage vergeht. Das kann nichtblockierend erreicht werden, indem der Server mit einer höheren Priorität gestartet wird als alle oder zumindest die meisten anfragenden Clients.
Dabei sollte „Priority Remembrance“ (Kapitel 3.2.5) zum Einsatz kommen, um das „Stehlen von Prioritäten“ (Kapitel 1.2.2) zu verhindern. „Time-Slice-Donation“ (Kapitel 3.2.3) bietet sich ebenfalls zur Lösung an.
Benötigt die Delegation im Mittel weniger Zeit als eine Zeitscheibe, so tritt das Delegationszeitproblem nicht auf.

Dienstbeschränkung

Mit der Einführung von Delegation ist es dem Server auch möglich eine Policy zur Dienstbeschränkung (Kapitel 3.2.2) durchzusetzen, z. B. die maximale Anzahl der Anfragen pro Task oder Thread pro Zeiteinheit. Dabei ist es umso wichtiger das Delegationszeitproblem zu

lösen, weil auch das Durchsetzen einer Policy Zeit benötigt, die zur Delegationszeit hinzugerechnet werden muss. Bei entsprechenden Policies kann an der Beschränkung auf 128 Threads festgehalten werden, da die beschränkte Ressource so „kontrolliert bereitgestellt“ wird (siehe Kapitel 2.1.1).

Kosten der multiplen Delegation

Multiple Delegation ist im Allgemeinen nicht teuer. Ein Mehraufwand gegenüber herkömmlicher Dienstnutzung besteht lediglich darin, einen neuen Serverthread zu starten und diesem über den gemeinsam genutzten Speicher die benötigten Daten mitzuteilen. Letzteres muss jeweils für den konkreten Fall betrachtet werden.
Durch das Erzeugen neuer Serverthreads bei jeder Anfrage wird es möglich, sowohl die Kapazität der CPU, als auch des Speichers auszulasten. Damit wird das Problem von der IPC-Warteschlange hin zur Task-/Threadverwaltung (Kapitel 5.2) und dem Speichermanagement (Kapitel 4.1) verschoben.

Probleme bei multipler Delegation

Je nach Art der Server kann es bei der Anwendung von Delegation zu kritischen Abschnitten und Synchronisationsproblemen kommen. Dafür gibt es keine generelle Lösung, da sich die Aufgaben der verschiedenen Server schwer vereinheitlichen lassen. Die meisten Probleme werden beim Zugriff auf den gemeinsamen Speicher der Servertask auftreten, z. B. beim Registrieren eines neuen Threads beim Namensdienst. Dafür stehen allgemeine Mechanismen zum wechselseitigen Ausschluss zur Verfügung. Es ist aber bei der Konstruktion von Servern, die Delegation nutzen, darauf zu achten, dass diese kritischen Abschnitte so klein wie möglich gehalten werden. Konstruktionsempfehlungen für DoS-sicheren Server werden im Kapitel 6 gegeben. Des weiteren wird Delegation im Kapitel 3.2.7 derart abgewandelt, dass diese Probleme nicht mehr auftreten.

Delegation vs. Propagation

Der Hauptvorteil von Delegation gegenüber Propagation (Kapitel 2.2.1) besteht darin, dass der Empfangsstatus des Clients nicht geändert werden muss. Im besten Fall kommt Delegation völlig ohne Kerneintritt aus, nämlich dann, wenn Arbeiterthreads bereits aktiv auf Delegationen warten.

3.2.5 Priority-Remembrance

Um das „Stehlen von Prioritäten“ (Kapitel 1.2.2) bei der Delegation zu verhindern, bekommt der neue Serverthread B_n die gleiche Priorität wie der anfragende Client A. Dafür muss der Server Kenntnis über diese Priorität haben.

3.2.6 Zwischenbetrachtung

Im Ergebnis der bisherigen Betrachtung steht fest, dass Time-Slice-Donation (Kapitel 3.2.3) und multiple Delegation (Kapitel 3.2.4) zwei geeignete Mechanismen zur Vermeidung bzw. Eingrenzung von Denial-of-Service-Angriffen sind, während die Einführung einer fairen Auswahlstrategie nicht ausreichend Schutz bietet (Kapitel 3.2.1).

Die Delegation der Dienste an neue Threads erscheint dabei mächtiger, da sie eine beschränkte Ressource abschafft. Dabei setzt die Delegation aber Time-Slice-Donation voraus, um das Delegationszeitproblem zu lösen. Delegation bietet zudem jedem Server die Möglichkeit, eine eigene Sicherheitspolicy durchzusetzen. Dieser Ansatz wird im Kapitel 3.2.7 nochmals aufgegriffen und weiterverfolgt.

Im Ergebnis steht ein Ablauf, wie er in Abbildung 3.5 bzw. Abbildung 3.6 dargestellt ist.

Je nach Variante müssen Veränderungen am Kern vorgenommen werden. Multiple Delegation benötigt den Syscall `send_and_receive_from_task`, der eine Nachricht an einen bestimmten Thread T_i einer Task T absetzt und danach auf Antwort von einem beliebigen Thread der Task T wartet. Das ist notwendig, da ein einfacher Übergang in den Zustand `wait` („offenes Warten“) nach Absenden der Nachricht den Client angreifbar macht. Für Priority-Remembrance (Kapitel 3.2.5) muss der Server über die Priorität des Client informiert werden.

Multiple Delegation mit Donating-Call ist zu bevorzugen, weil dabei keine Donation-Time-Expiration (Abbildung 3.2) auftritt.

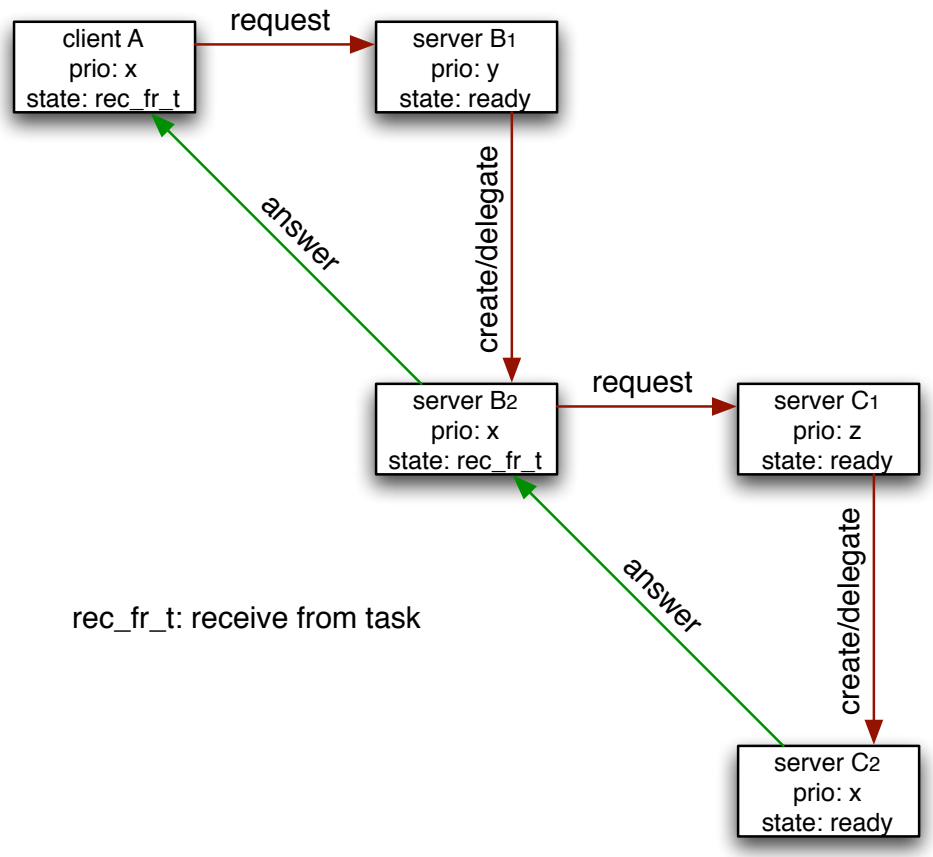

Abbildung 3.4: Priority-Remembrance

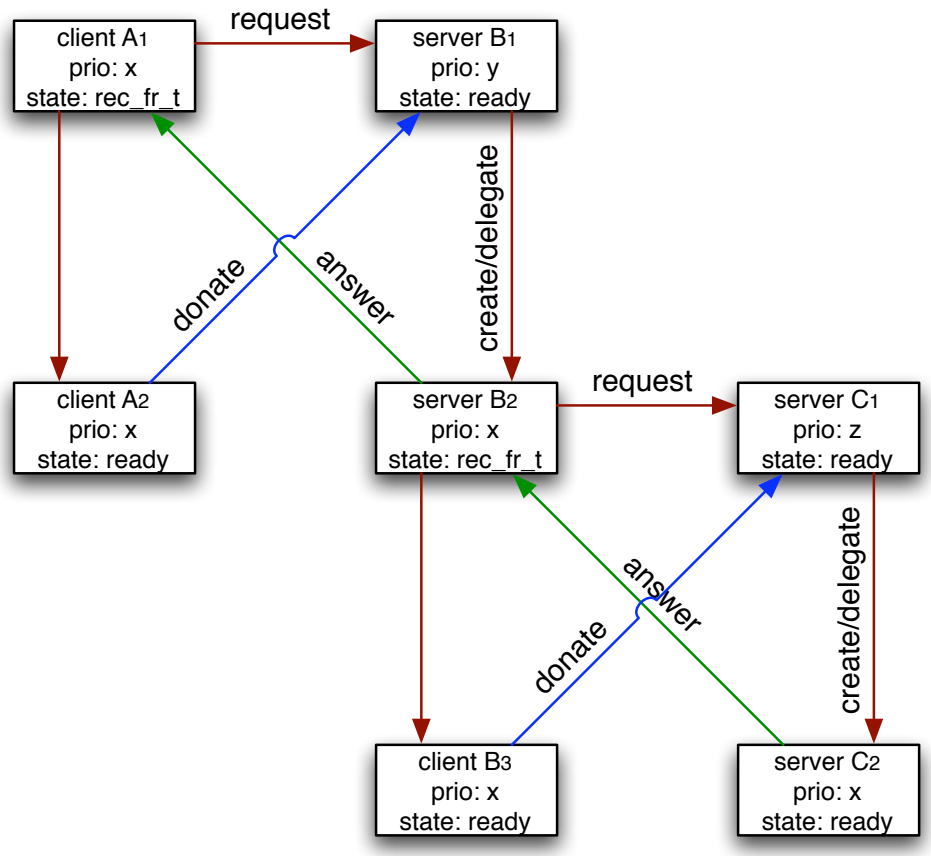

Abbildung 3.5: donierende multiple Delegation ohne „Donating-Call“

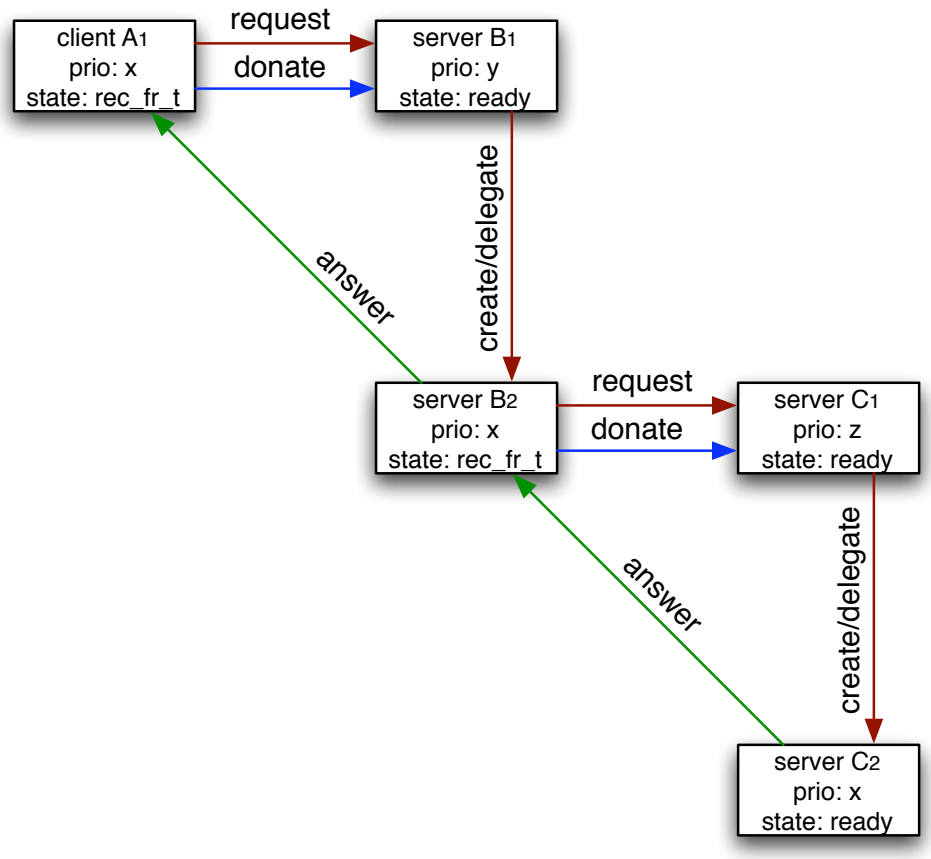

Abbildung 3.6: donierende multiple Delegation mit „Donating-Call“

3.2.7 Duale Delegation als Ersatz für Clans & Chiefs

Der im Kapitel 3.2.4 vorgestellte Delegationsmechanismus lässt sich in abgewandelter Form als Ersatz für Clans & Chiefs einsetzen.
Ein Server besteht hier grundsätzlich aus nur zwei Threads: dem Vermittler (coordinator) und dem Arbeiter (worker). Anfragen werden grundsätzlich an den Vermittler gestellt. Dieser delegiert die Anfragen an den Arbeiter, der ggf. die Antwort an den anfragenden Thread sendet. Die Delegation läuft dabei derart ab, dass die bei der Anfrage empfangenen Daten vom Verwalter zusammen mit der ThreadID des Anfragers in einen mehrelementigen Puffer geschrieben werden und der Arbeiter aus diesem Puffer liest.
Vorteile der dualen Delegation gegenüber multipler Delegation bzw. Clans & Chiefs:

- keine Synchronisationsprobleme, da es nur einen Arbeiterthread gibt
- Der Verwalterthread kann Policies zur Beschränkung von DoS-Angriffen durchsetzen.
- keine Isolation bei gelöstem Delegationszeitproblem möglich (Kapitel 3.2.4 und 2.2.4)
- kaum Leistungseinbußen

Nachteile der dualen Delegation gegenüber multipler Delegation bzw. Clans & Chiefs:

- Schutz nur durch geeignete Policy
- Bei großer Last und ungünstiger Policy kann der Puffer „volllaufen“ und so wiederum DoS-Angriffe ermöglichen.

Das „Volllaufen“ des Puffers kann durch Verwenden von dynamischen Datenstrukturen vermieden werden. Damit verbundene Angriffsmöglichkeiten auf den Speicher werden im Kapitel 4.1 besprochen.

3.2.8 Endpunkte

Endpunkte verlagern Teile des Delegationskonzeptes in den Kern, lösen aber wie im Kapitel 2.2.5 beschrieben das betrachtete DoS-Problem nicht. Auch ist es durch das Fehlen eines koordinierenden Threads nicht mehr möglich auf einfache Weise Policies durchzusetzen.

Delegation mittels Endpunkten

Das Delegationkonzept lässt sich aber so verändern, dass es die Vorteile von Endpunkten ausnutzt und zugleich seine Eigenschaften zum Schutz vor DoS behält.
Es wird zunächst nur eine Instanz des Servers gestartet. Dieser wartet „hinter“ dem Endpunkt auf Anfragen. Trifft eine Anfrage ein, kann er bei Bedarf seine Policy durchsetzen, startet dann ggf. eine weitere Instanz von sich selbst, die nun wiederum „hinter“ dem Endpunkt wartet, und bearbeitet die Anfrage. Danach beendet er sich selbst.

Eigenschaften dieser Delegationslösung

- Die Delegationszeit ist bei dieser Lösung immer konstant, da keine Daten übermittelt sonder nur ein neuer Thread gestartet werden muss. Das ist ein großer Vorteil gegenüber der ursprünglichen Idee.

- Es ist immer eine Instanz des Servers bereit Anfragen zu bearbeiten.
- Der Server kann entsprechende Policies durchsetzen.

Fazit

Endpunkte ermöglichen eine deutliche Vereinfachung des multiplen Delegationskonzeptes und machen duale Delegation obsolet.

3.2.9 Zusammenfassung

Zur Begrenzung von DoS- bzw. DDoS-Angriffen auf Server via IPC werden beide Lösungsansätze (Kapitel 2) umgesetzt:

1. Replikation in Form der multiplen Delegation
2. „Kontrollierte Bereitstellung beschränkter Ressourcen“ in Form der Durchsetzung sinnvoller Policies mittels dualer Delegation.

Dabei werden einige Probleme lediglich verschoben. So ergibt sich z. B. aus der dualen Delegation das Problem, dass ggf. immer mehr Speicher benötigt wird. Diese Probleme werden in späteren Kapiteln besprochen.

4 Entwicklung des „Dealman“ am Beispiel Speicher

In diesem Kapitel wird ein System zur kontrollierten Bereitstellung von Ressourcen mittels ökonomischer Modelle geschaffen.

4.1 Vermeidung von Angriffen auf den Nutzerspeicher

Kapitel 1.2.3 beschreibt zwei grundlegende Angriffe: einen speziellen indirekten Angriff auf den Kernspeicher durch übermäßiges Füllen der Mappingdatenbank und einen allgemeinen direkten Angriff auf den Nutzerspeicher von anderen Threads. Pager stellen hierbei einen Spezialfall dar, da der Dienst, den sie anbieten, das Bereitstellen von Speicher ist.
Der beschriebene IPC-Angriff auf Pager wird hier nicht näher betrachtet, da er durch sinnvollen Einsatz der im Kapitel 3.2.4 und 3.2.7 vorgeschlagenen Methoden bekämpft werden kann.
Das Kapitel 4.1 beschäftigt sich hingegen ausführlich mit dem zweiten Problem und zeigt zunächst grundlegende Lösungsansätze auf, um danach mit der Einführung des Memory-Accounting die Grundlage für ein allgemeines Ressourcen-Accounting (Kapitel 4.2) zu schaffen. Dieses wird dann im Kapitel 5.1 genutzt, um das erst genannte Problem zu lösen.

4.1.1 Sichere Pager durch bekannte Policies

Angriffe auf Pager sind durch sinnvolle Policies verhinderbar. Ob und wie stark der beschriebene Angriff ausgenutzt werden kann, hängt also in großem Maße von deren Arbeitsweise ab. Ein ganz primitiver Pager, der bei Pagefaults einfach entsprechende Kacheln mappt, stellt dem keinerlei Widerstand entgegen. Im Folgenden werden einige Policies besprochen. Dem Vorbild von [Not02] folgend, werden zunächst ein statischer- und ein dynamischer Ansatz betrachtet.

Statisch: Quotas

Die einfachste Methode besteht in der Limitierung der Kachelanzahl pro Task. Hat eine Task ihre maximale Anzahl an Kacheln bereits erhalten, so erhält sie keine neuen. Wie bereits in [Not02] festgestellt wurde, führt diese Policy zum einen zu einer schlechten Ressourcenauslastung und verbietet es zudem Programme mit erhöhten Speicherbedarf auszuführen. Auch kann damit die Dienstfähigkeit von Servern behindert werden, die für die Diensterbringung Speicher allokieren müssen, was wiederum zum Problem der beschränkten Ressource führt.

Dynamisch: Einsatz ökonomischer Modelle

Diese Probleme versuchen ökonomische Modelle aufzuheben, indem sie Ressourcen einen Preis und jeder Anwendung ein Budget zuordnen [Not02, MD88c, MD88b, MD88a]. Im Falle des

Speichers heißt das, Tasks bekommen periodisch einen bestimmten Betrag auf ihr Konto gutgeschrieben und müssen davon Miete für den belegten Speicher zahlen. Diese Kosten können sich ändern, je nachdem, wie viel Speicher zur Verfügung steht. Einem Angreifer wird also ziemlich schnell „das Geld ausgehen“, wenn er ohne Freizugeben mehr und mehr Speicher anfordert. Das Modell hat aber einen entscheidenden Nachteil - es ist aufwendig. Trotzdem wird es zu einem späteren Zeitpunkt aufgegriffen und bildet die Grundlage für das im Kapitel 4.1.4 konstruierte und im Kapitel 4.2 erweiterte ökonomische System.

Abschaffung der Beschränkung der Speichergröße

Ein anderer Ansatz ist, den Speicher als beschränkte Ressource abzuschaffen. Typischerweise erreicht man das durch Swapping. Sind alle Kacheln belegt, wählt der Pager nach einer bestimmten Strategie eine Kachel und lagert deren Inhalt auf einen Hintergrundspeicher (z. B. eine Festplatte) aus. Auf diese Weise wird der benötigte Platz geschaffen. Bei Bedarf werden die Daten wieder eingelagert. Auch diese Methode hat Nachteile. Neben der Zeit, die für den Mechanismus als solchen benötigt wird, ist vor allem der Zugriff auf den Hintergrundspeicher sehr teuer, betrifft das gesamte System und führt in bestimmten Situationen zu Thrashing. Zudem wird das Problem der knappen Ressource Speicher nun zu einem Problem der knappen Ressource Hintergrundspeicher.

Bewertung der Lösungsansätze

Es ist deutlich geworden, dass keiner der Ansätze das Problem bezüglich Kosten und Nutzen optimal löst. Eine genauere Untersuchung des dynamischen Modelles würde den Umfang dieser Arbeit sprengen. Aber auch so wird deutlich, dass in jedem Fall ständiger Mehraufwand die Folge ist. Im folgenden Kapitel wird daher eine Kombination der Ansätze vorgestellt.

4.1.2 Sichere Pager durch Task-bounded-Swapping

Eine Verbindung aus statischen Modell und Swapping stellt einen Kompromiss zwischen Fairness und Aufwand dar:

- Jeder Task wird bei ihrer Erzeugung ein bestimmtes Budget von Kacheln zugestanden, die ihre Threads anfordern können.
  Die Größe dieses Budgets kann statisch sein oder dynamisch festgelegt werden.
- Ist das Budget erschöpft, werden bei erneutem Anfordern Kacheln der Task geswappt.

Der Einfluss, der Ressource Busbandbreite, wird vernachlässigt. Der Aufwand der Behandlung einer Speicheranforderung erhöht sich dabei um das Abprüfen des aktuellen Budgets. Weiterer Overhead entsteht nur, wenn Threads das Kontingent ihrer Task auslasten und betrifft nur diese. Thrashing kann nur noch innerhalb von Tasks auftreten. Da das Swappen auf der Zeitscheibe des jeweiligen Threads erfolgt (ggf. Anwenden von Methoden aus Kapitel 3.2), ist es so auch nicht möglich, das System durch ständiges „Überbeanspruchen“ des Budgets auszubremsen.
Auch bezüglich Echtzeit entsteht kein neues Problem, da Echtzeit-Tasks ihren Speicherbedarf kennen, also wissen wann geswappt werden muss und die Kosten dafür ebenfalls bekannt sind.
Aber auch Task-bounded-Swapping stellt keine Endlösung dar. Da die Gesamtzahl der Kacheln begrenzt ist, kann bei restriktiver Budgetvergabe die Gesamtzahl der möglichen Tasks

begrenzt und so eine neue Angriffsmöglichkeit geschaffen werden. Das kann aber vernachlässigt werden, da die maximale Anzahl der Tasks in jedem Fall durch die Größe des physischen Speichers begrenzt ist und Probleme der Task- und Threadverwaltung im Kapitel 5.2 behandelt werden. Des weiteren bleibt das Problem der beschränkten Ressource Hintergrundspeicher.

Erweiterung der Grundidee

Eine bereits angesprochene Erweiterung dieses Mechanismus besteht darin, die Größe des Budgets dynamisch beim Start festzulegen. Das kann z. B., wie später beschrieben, vom Elternthread vorgenommen werden. So wird verhindert, dass Programme, die ihrer Natur entsprechend viel Speicher benötigen, ständig swappen müssen. Die erwähnte Begrenzung der Taskanzahl kann durch „intelligente“ Budgetvergabe aufgehoben werden.
Task-bounded-Swapping kann nicht verhindern, dass ein Angreifer viele neue
Tasks und Threads erzeugt, die alle ihr Budget ausschöpfen. Dieses Problem kann aber auf „Aktivitätsbomben“ (Kapitel 5.2) zurückgeführt werden.

4.1.3 Sichere Server durch Memory-Donation

Im allgemeinen Fall hat ein Server keine Möglichkeit seine Speicherlast sinnvoll zu kontrollieren, ohne damit seine Dienstfähigkeit zu beschränken.
Ein Angreifer kann durch ständiges Beanspruchen der Serverdienstleistung ggf. dessen Speicher füllen. Wird Task-bounded-Swapping (Kapitel 4.1.2) angewendet, so kann zudem die Leistungsfähigkeit des Servers eingeschränkt werden, wenn dieser sein maximales Kachelbudget aufgebraucht hat und ständig swappen muss.
Das Hauptproblem besteht darin, dass entsprechende Server ihren eigenen Speicher nutzen müssen. Analog zur im Kapitel 3.2.3 besprochenen Time-Slice-Donation wird daher hier der Mechanismus Memory-Donation vorgeschlagen. Um das zu ermöglichen wird im nächsten Kapitel das Konzept Task-bounded-Swapping zu einem vollwertigen Speicher-Accounting-System erweitert.

4.1.4 Memory-Accounting

Die im Kapitel 4.1.2 vorgestellten Budgets sind der erster Ansatz für ein Speicher-„Bezahlsystem“. In diesem Kapitel wird ein Speicher-Accounting-System vorgestellt, mit dem sich Task-bounded-Swapping und Memory-Donation implementieren lassen. Durch die Einführung von globalen Budgets und Wichtungen, kann so ein an das „ökonomische Modell“ (Kapitel 4.1.1, [Not02, MD88c, MD88b, MD88a]) angelehntes ökonomisches System geschaffen werden.

Ziele von Memory-Accounting

Ziele des Memory-Accounting, das später zu einem Accounting-System für beliebige Ressourcen erweitert wird, sind die Kontrolle des Zugriffs auf den Speicher und die Schaffung von Möglichkeiten des mehrseitig beschränkten Donierens.
Mehrseitig beschränktes Donieren beinhaltet folgende Forderungen:

- Client und Server müssen sich gegenseitig nicht vertrauen.
- Statt Donieren soll auch „Borgen“ (zeitlich beschränktes Überlassen - ab sofort Lending genannt) von Speicher möglich sein.

- Die Dauer des Lending soll verhandelbar sein.

Die Grundidee

Jede Task besitzt ein persönliches Kontingent an Gums (ab sofort synonym mit γ bezeichnet), die eine Währung darstellen. Pager können mit Hilfe dieser Gums Speicher bezahlpflichtig machen. Außerdem ist es möglich Gums an andere Tasks zu übertragen und so indirekt Speicher zu donieren. Beim Donieren wird dazu auf dem Konto des Empfängers ein neues Kontingent des Senders angelegt. Auf dieses „Unterkonto“ werden die Gums „überwiesen“. Pager können so bei der Speicherallokation immer das jeweils richtige Kontingent belasten lassen.

Das Übertragen von γ wird von einem Thread des Memory-Accounting-Managers vorgenommen. Im Falle von Lending wartet dieser Thread (ab sofort Lending-Observer genannt) nach dem Übertragen auf das Ablaufen der Lending-Frist. In dieser Zeit können Server und Client über eine eventuelle Verlängerung dieser Frist verhandeln. Ist die erste Frist abgelaufen prüft der Lending-Observer, ob die Frist verlängert wurde. Ist das der Fall, wartet er erneut auf deren Ablaufen. Gab es keine Verlängerung, wird das entsprechende Kontingent invalidiert. Invalidierte Kontingente können vom ihrem Client zurückgefordert werden. Der Client muss sich damit selbst um die Rückgabe der geliehenen γ kümmern. Auf diese Weise kann er dem Server eine gewisse „Karenzzeit“ einräumen bzw. eine Donierung mit zeitlich beschränkter Verfügbarkeitsgarantie durchführen.
Für die tatsächliche Rücknahme des indirekt donierten Speichers ist der Pager des jeweiligen Threads verantwortlich. Dieser wird entweder vom Memory-Accounting-Manager bei negativ gewordenen Kontingent informiert oder prüft selbstständig, z. B. bei einem Pagefault ob sich unter den Threads die er bedient „Schuldner“ befinden. In jedem Fall entzieht er dem „Schuldner“ die entsprechenden Kacheln. Damit er das kann, muss er sich bei jeder Allokation merken, wessen Kontingent diese belastet.

Memory-Accounting-Manager

Für die Umsetzung wird zunächst ein vertrauenswürdiger Server benötigt, der die Kontingente und weitere Daten verwaltet und entsprechende Dienste anbietet. Bei der Konstruktion dieses Servers haben Sicherheitsaspekte einen hohen Stellenwert. So darf zum Beispiel unrechtmäßiges Verändern von Kontingenten nicht möglich sein. Auch darf der Server seinerseits nicht via DoS angegriffen werden können. Deshalb sollte er z. B. die im Kapitel 3.2.4 beschriebene Methode der Delegation nutzen. Der Memory-Accounting-Manager bekommt an dieser Stelle keinen speziellen Namen, da das Konzept Memory-Accounting später zu einem Ressourcen-Accounting erweitert wird. Der hier vorgestellte Memory-Accounting-Manager wird dann durch den γ-Bank-Manager „Dealman“ ersetzt.

Benötigte Datenstrukturen

Genau wie bei einem Bankkonto, gibt es eine eindeutige ID. Um eine einfache Zuordnung zu ermöglichen, entspricht die KontoID der TaskID des Kontoinhabers: *tid*.

Zur Verwaltung der Unterkonten (Kontingente) wird eine Kontingenteliste benötigt. Dabei unterscheidet man zwischen dem persönlichen Kontingent einer Task und donierten b. z. w. geborgten Kontingenten. Das persönliche Kontingent bekommt die Task bei ihrer Erzeugung

zugewiesen. Donierte und geborgte Kontingente werden von den donierenden b. z. w. borgenden Tasks angelegt und enthalten zweckgebundene Gums.
Jedes Kontingent besteht dabei aus dem Quadrupel: $(cid, c, t_{lending}, valid)$.

- Die KontingentID *cid* entspricht der TaskID der Task, zu der dieses Kontingent gehört. Für das persönliche Kontingent, das zu jedem Konto gehört, gilt: $cid = tid$. Bei jedem anderen Kontingent entspricht *cid* der *tid* der donierenden- b. z. w. borgenden Task.
- Der Kontingentwert c steht für die Menge an Gums, die das Kontingent beinhaltet.
- Die neue Lendingfrist $t_{lending}$ zeigt dem Lending-Observer an, ob und um welchen Wert die Lendingfrist verlängert wurde.
- Das valid-Flag *valid* markiert ein Kontingent als valide, bzw. invalide. Invalide Kontingente können von ihren Erzeugern zurückgefordert werden. Das ist typischerweise nach Ablauf einer nicht verlängerten Lendingfrist der Fall.

Des weitern wird eine Liste von Ressourcen-Providern (RP) für jedes Konto verwaltet. Im Kontext des Memory-Accounting ist ein Ressourcen-Provider ein Thread, der einer Task Speicher mappt. In erster Linie ist das ihr Pager. RP verlangen für die angebotene Ressource Gums. Dazu weisen sie den Memory-Accounting-Manager an, das entsprechende Kontingent zu dekrementieren. Gibt ein Client Ressourcen an einen RP zurück, so lässt dieser das Kontingent wieder inkrementieren. Zum Schutz des Clients, dürfen nur RP, die in der RP-Liste eines Kontos stehen, dessen Kontingente belasten oder aufwerten lassen. Zum Schutz des Systems, darf ein RP ein Kontingent nicht stärker aufwerten, als er es abgewertet hat. Um das zu gewährleisten merkt sich der Memory-Accounting-Manager für jeden RP und jedes Kontingent, den Wert der Dekrementierungen.
RP soll es außerdem möglich sein, einen Umrechnungskurs festzulegen. Dieser Kurs gibt an, wie viele Gums eine Einheit der angebotenen Ressource bei ihm kostet. Der Kurs wird als Wichtung bezeichnet.
Ein Eintrag in der RP-Liste besteht damit aus folgenden Daten:

- der ThreadID des RP *rpid*
- der Wichtung w
- einer Liste der Kontingentabwertungen jedes Kontingents *sub*

Für die Verwaltung der RP-Liste, sowie Donation und Lending ist außerdem eine Capability *cap* pro Konto notwendig. Diese Capability erlaubt ihrem Besitzer die folgenden Aktionen:

1. Hinzufügen neuer RP zur RP-Liste des Kontos
2. Das Abfragen von Kontingenten des Kontos
3. Einleitung einer Donation von dem zugehörigen Konto
4. Einleitung eines Lendingvorgang von dem zugehörigen Konto

Abbildung 4.1 zeigt die pro Konto verwalteten Daten.

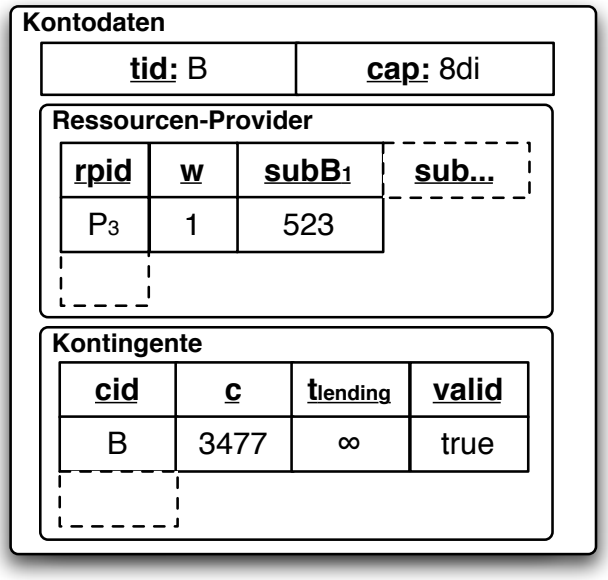

Abbildung 4.1: Vom Memory-Accounting-Manager verwaltete Daten

Angebotene Dienste

Die folgenden Dienste bietet der Memory-Accounting-Manager an:

- Abfragen von Kontingenten einer Task (Capability *cap* erforderlich)
- Anmelden eines neuen RP (Capability *cap* erforderlich)
- Dekrementieren eines Kontingents einer Task durch einen ihrer angemeldeten RP
- Inkrementieren eines Kontingents einer Task durch einen ihrer angemeldeten RP (maximal um den Wert aller Dekrementierungen (*sub*))
- Übertragen von Kontingent der Task zu einer anderen (Capability *cap* erforderlich)
- Änderung seiner Wichtung w durch einen angemeldeten RP
- Änderung der Lending-Frist $t_{lending}$ eines Kontingents durch den borgenden Client
- Rückübertragung eines invaliden (*valid*) kompletten Kontingents durch den Client nach Ablauf der Lending-Frist $t_{lending}$

Im Folgenden werden die in Memory-Accounting enthaltenen Ideen und Funktionsweisen im Detail erläutert.

Erzeugen von Tasks

Beim Start einer neuen Task wird für diese automatisch ein Konto beim Memory-Accounting-Manager angelegt. Dabei bekommt sie ein persönliches Kontingent, mit ihrer Tasknummer

als Kontingent-ID *cid*. Der Wert des persönlichen Kontingents c wird vom Elternthread festgelegt (Näheres dazu in den Kapiteln 5.2.1 und 5.2.1). Außerdem wird die Capability *cap* (Zufallszahl) für das Konto erzeugt. Diese Daten werden vom Memory-Accounting-Manager in einer entsprechenden Datenstruktur (Abbildung 4.1) abgelegt.
Der Pager der Task wird zusammen mit der Wichtung $w = 1$ und der Zahl seiner Kontingentabwertungen des persönliche Kontingents der Task $sub = 0$ in die RP-Liste eingetragen.
Es ist durchaus denkbar, die Möglichkeit anzubieten, das persönliche Kontingent c zu einem späteren Zeitpunkt nachträglich zu verändern. Mittels eines entsprechenden Protokolls kann ggf. jederzeit mit anderen Threads, insbesondere dem Elternthread über c „verhandelt“ werden.

Arbeitsweise des Pagers

Threads können bei ihrem Pager γ gegen Speicher tauschen. Dazu wird bei der Allokation ein Kontingent angegeben, das belastet werden soll. Dieses Kontingent muss auf dem Konto der zugehörigen Task existieren. Der Pager prüft nun, ob das Kontingent genug γ enthält, weist den Memory-Accounting-Manager an, dieses entsprechend zu dekrementieren und mappt bei einem Pagefault die benötigten Kacheln. Zudem führt er eine Mapping-Tabelle, die zu jeder Kachel das belastete Kontingent und die zum Zeitpunkt der Reservierung aktuelle Wichtung w enthält.

Die Rücknahme von Mappings kann auf zwei verschiedene Arten getriggert werden. Der Pager kann selbstständig zu einem beliebigen Zeitpunkt (z. B. wenn sein Speicher knapp wird) vom Memory-Accounting-Manager eine Liste aller von ihm bedienten Threads anfordern, die Kontingente mit negativen Werten besitzen und diesen die Mappings entziehen. Oder er wird direkt vom Memory-Accounting-Manager informiert, sobald entsprechende Kontingente negativ werden. Das ist möglich, da der Memory-Accounting-Manager durch die Kontodatenbank über alle Pager-Task-Zuordnungen Bescheid weiß.

Aufgrund der Verantwortlichkeit des Pagers für die Rücknahme von Mappings ergeben sich weiterhin vielfältige Möglichkeiten für „Bezahlpolicies“. Der Server B_1 kann beispielsweise mit seinem Pager folgende Vereinbarung treffen:

„Zieht der Client A_1 seine γ nach Ablauf der Lendingfrist $t_{lending}$ vom entsprechenden Kontingent zurück, dann gleiche dieses Kontingent mit dem persönlichen Kontingent von B aus und entferne das Mapping nicht.“

Anmelden weiterer Ressourcen-Provider

Nur in der RP-Liste der Task beim Accounting-Manager eingetragene Threads können Kontingente verändern. Deshalb müssen eventuell weitere RP bei diesem angemeldet werden. Das kann entweder ein Thread der Task selbst durchführen, oder der RP erhält die Capability der Task und meldet sich selbst an. In diesem Fall ist zu beachten, dass so dem RP ggf. mehr Rechte als gewünscht eingeräumt werden, da der Besitz der Capability auch zum Donieren und Borgen von Gums an andere Tasks berechtigt.

Capablilty vs. Liste der Ressourcen-Provider

Da der Besitz der Capability *cap* zum Anmelden neuer RP ermächtigt beinhaltet sie die Rechte der RP.
Die Liste der Ressourcen-Provider wird benötigt, da es oftmals nicht möglich ist, einem Ressourcen-Provider *cap* rechtzeitig zukommen zu lassen. Außerdem ermöglicht sie einige später vorgeschlagene Optimierungen.

Es wird davon ausgegangen, das RP über die angebotene Ressource direkt verfügen und diese nicht von einem anderen RP beziehen müssen.
Ist das doch der Fall (z. B. bei hierarchischen Pagern), so kann die folgende Semantik angewendet werden:

- Der RP besitzt selbst ein Konto
- Der RP besitzt zudem die Capability *cap* seines Clients
- Fordert der Client Speicher vom RP, so überweist dieser sich mit Hilfe der Capability *cap* die entsprechende Menge an γ auf sein Konto und kann dafür wiederum Speicher von einem seiner „Lieferanten“ erhalten.
- Gibt der Client den Speicher wieder zurück, werden die γ an diesen zurück überwiesen.

Das setzt voraus, dass der RP, solange er die Capability *cap* noch nicht besitzt, genug Speicher zur Verfügung hat um die Forderungen des Clients zu bedienen.

Memory-Donation

Eine grundlegende Frage, die sich bei Client-Server-Beziehungen stellt ist: Wer vertraut wem? In der Regel ist davon auszugehen, dass der Server seinen Clients nicht vertraut. Genau aus diesem Grund wird Memory-Accounting eingeführt. Es bleibt zu betrachten, wie die Donationprotokolle auszusehen haben, wenn Clients dem Server vertrauen, bzw. nicht vertrauen.

Zunächst wird der einfachere Fall des vertrauenswürdigen Servers dargestellt.
Client A_1 doniert an den Server B_1 [Falls B_1 keine Delegation nutzt (Kapitel 3.2.4) ist $i = 1$.].
Trusted-Donation-Protokoll:

- A_1 übermittelt B_1 seine Capability *cap*.
- B_i weist den Memory-Accounting-Manager M an, der Task B das benötigte Kontingent von A zu übertragen.
- M führt diese Anweisung aus und schickt eine Bestätigung in Form der *tid* von A_1 und der Menge des übertragenen Kontingents $c_{donated}$ an B_i mit Sendetimeout 0. Reichte das Kontingent von A_1 nicht aus, ist $c_{donated} = 0$.

A_1 muss hierbei darauf vertrauen, dass B_1 nur so viele Gums an sich selbst übertragen lässt, wie vereinbart und diese ggf. nach Abschluss der Dienstleistung an A_1 zurückgibt. Insbesondere für Server, die grundlegende Systemfunktionen implementieren bietet, sich dieses Verfahren an, da die Server im Allgemeinen zur Trusted Computing Base gehören.

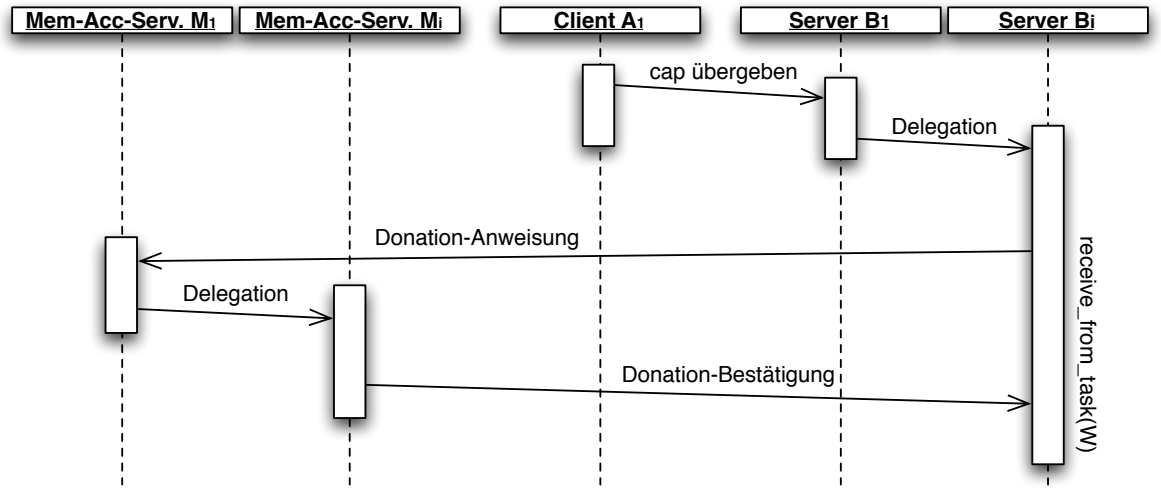

Abbildung 4.2: Trusted-Donation-Protokoll: Donation mit vertrauenswürdigem Partner

Da Memory-Accounting auch für die Konstruktion jener Server nutzbar gemacht werden soll, deren Clients dieses Vertrauen nicht entgegenbringen, wird nun das Protokoll zur Client-sicheren-Donation vorgestellt.
Client A_1 doniert an den Server B_1 (Falls B_1 keine Delegation nutzt (Kapitel 3.2.4) ist $i = 1$.).
Untrusted-Donation-Protokoll:

- A_1 teilt B_1 die Absicht der Donation mit.
- B_i bestätigt das und wartet daraufhin auf Antwort vom Memory-Accounting-Manager M.
- A_1 weist M an, eine bestimmte Menge von Gums vom persönlichen Kontingent seiner Task an die Task B zu übertragen und übermittelt zudem die ThreadID von B_i.
- M führt diese Anweisung aus und schickt eine Bestätigung in Form der *tid* von A_1 und der Menge des übertragenen Kontingents $c_{donated}$ an den Beschenkten B_i mit Sendetimeout 0. Reichte das Kontingent von A_1 nicht aus, ist $c_{donated} = 0$.

Mit Hilfe von Abbildung 4.3 wird deutlich, dass es hier eine neue Angriffsmöglichkeit auf den Server B_1 gibt. Leitet A_1 nach der Bestätigung von B_i nicht die Donation ein, wird B_i zum Zombie. Diesem und weiteren Sicherheitsaspekten des Memory-Accounting widmet sich Kapitel 4.1.4. Es sei daher an dieser Stelle lediglich angemerkt, dass dieses Problem lösbar ist.

Memory-Lending

Lending, das „zeitlich beschränkte Überlassen von Speicher“, wurde als eines der Hauptziele des Memory-Accountings im Kapitel 4.1.4 definiert. Das entsprechende Protokoll wird hier präsentiert.
Wie bereits im Kapitel 4.1.4 grundlegend beschrieben wird jedes Übertragen von Gums von einem eigenen Thread des Memory-Accounting-Managers durchgeführt. Im Falle von Lending heißt dieser Thread Lending-Observer.

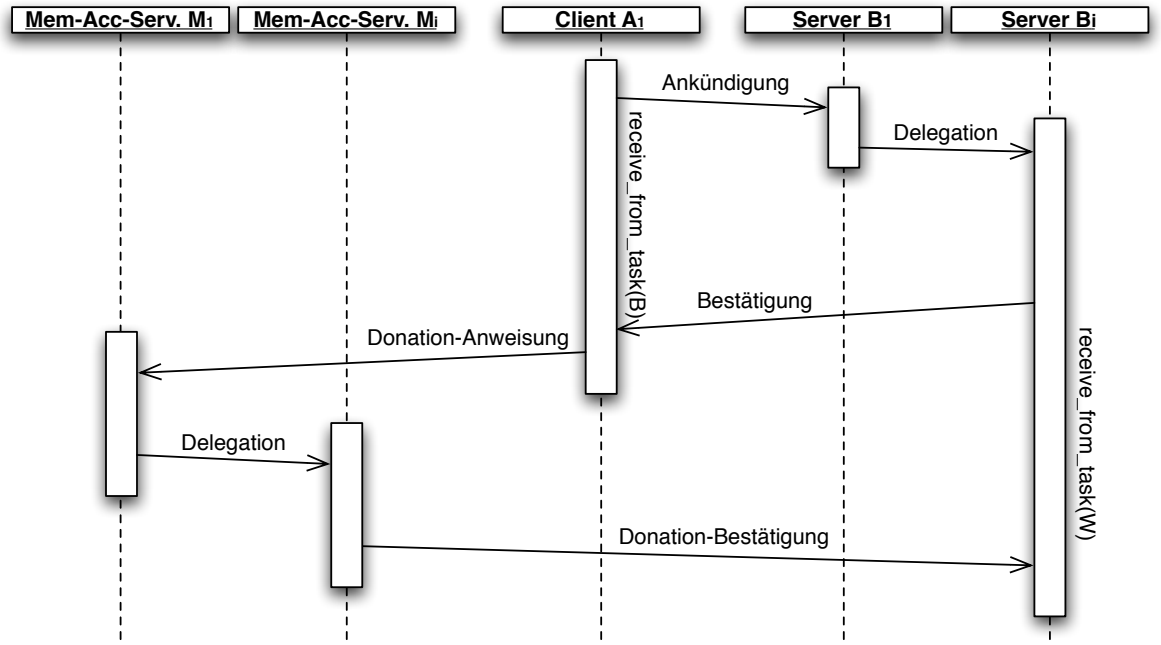

Abbildung 4.3: Untrusted-Donation-Protokoll: Donation mit nicht vertrauenswürdigem Partner

Client A_1 doniert mit Hilfe des Lending-Observers LO_1 zeitlich beschränkt an B_1 [Falls B_1 keine Delegation nutzt (Kapitel 3.2.4) ist $i = 1$.]:

- A_1 teilt B_1 die Absicht des Lending mit.
- B_1 bestätigt das und wartet daraufhin auf Antwort vom Lending-Observer LO_1.
- A_1 weist dem Memory-Accounting-Manager M entsprechend an und übermittelt dabei die zu übertragende Menge an Gums, sowie die Lending-Frist.
- M delegiert an LO_1
- LO_1 überträgt die Gums auf das entsprechende Kontingent, setzt dessen neue Lending-Frist $t_{lending}$ auf 0 und schickt an B_1 eine Bestätigung mit Sendetimeout 0.
- LO_1 blockiert bis zum Ablauf der aktuellen Lending-Frist
- B_1 kann nun bei Bedarf mit A_1 über eine Verlängerung der Lending-Frist verhandeln. Kommt es dabei zu einem Konsens, so weißt A_1 M an, die neue Lending-Frist $t_{lending}$ entsprechend zu setzen.
- Nach Ablauf der alten Lending-Frist prüft LO_1, ob $t_{lending}$ sich geändert hat. Ist das der Fall, setzt er $t_{lending}$ wieder auf 0 und wartet erneut. Ist das nicht der Fall, so invalidiert er das entsprechende Kontingent durch Umsetzen dessen valid-Flags *valid*.

Es sollte deutlich sein, das Memory-Donation ein Spezialfall von Memory-Lending mit unendlicher Lending-Frist ist.

Unterschiedliche Wichtung

Benutzen die Pager der beiden Donation- bzw. Lending-Partner unterschiedliche Wichtungen w, so ist die Donation im allgemein beschriebenen Fall 4.1.3 unfair, da die Wertigkeiten der übertragenen Gums auf beiden Seiten nicht übereinstimmen. Aus diesem Grund verwaltet der Memory-Accounting-Manager den Wert dieser Wichtung w. Stimmen die Wichtungen nicht überein, werden sie vor der Übertragung ineinander umgerechnet.

Zusammenfassung der wichtigsten Grundlagen

- Jede Task bekommt bei ihrem Start ein persönliches Kontingent c an Gums zugewiesen.
- Nur Threads (sogenannte Ressourcen-Provider), die in der RP-Liste einer Task stehen, können dieses Kontingent verändern.
- Zum Eintragen von Threads in die RP-Liste ist die Capability *cap* notwendig, die jede Task bei ihrer Erzeugung ebenfalls erhält.
- Für jeden RP wird die Menge an Gums verwaltet, die er bereits einem Kontingent entzogen hat (*sub*).
- Pager können Kontingente maximal um den entsprechenden Wert *sub* erhöhen.
- Für jeden RP wird eine Wichtung w verwaltet, die den seinen Preis einer Speichereinheit angibt.
- Jede Gums-Donation wird von einem eigenen Thread des Memory-Accounting-Managers durchgeführt. Auf diese Weise kann neben „normalem“ Donieren auch Lending durchgeführt werden. Der ausführende Thread des Memory-Accounting-Managers heißt in diesem Fall „Lending Observer“.

Sicherheit des Memory-Accounting

Wie bereits in 4.1.4 angesprochen, müssen der Memory-Accounting-Manager sowie die genutzten Protokolle vor Angriffen und Manipulation sicher sein. In Folgenden werden die möglichen Lücken untersucht.

$\rightsquigarrow$ Unberechtigtes Aufwerten des Kontingents: *Kann ein kollaborierender Pager seiner Task unrechtmäßige Gums verschaffen?*
Gibt ein Thread Speicher wieder an seinen Pager zurück, so wertet dieser das Kontingent des Threads wieder auf. Damit ein kollaborierender Pager das nicht unkontrolliert tut und der Task somit ein höheres Kontingent verschaffen kann, prüft der Memory-Accounting-Manager vor jeder Erhöhung den Wert *sub*. Dieser gibt an, um wie viel der Pager das Kontingent bereits abgewertet hat. Maximal um diesen Wert kann es wieder erhöht werden. Ein Pager kann seinem Thread also maximal so viele Gums zukommen lassen, wie durch ihn bereits abgezogen wurde.

$\rightsquigarrow$ Unberechtigtes Abwerten des Kontingents: *Kann ein böswilliger Thread einer fremden Task unrechtmäßig Gums entziehen?*

Nur beim Memory-Accounting-Manager für die Task registrierte Pager können Kontingent abwerten. Registriert wird beim Start der Task lediglich der jeweilige Pager. Weitere können nur mit Zutun eines Threads der Task registriert werden. Threads dürfen daher nur vertrauenswürdige RP registrieren.

⇝ Unberechtigtes Donieren von Kontingent: *Kann ein böswilliger Thread einer fremden Task Gums stehlen?*
Nur Threads, die die Capability *cap* der Task besitzen, können deren Kontingent donieren. Diese Capability kennen zunächst nur die Threads der Task. Es ist diesen aber möglich *cap* Threads anderer Tasks mitzuteilen. Daher liegt es in der Verantwortung jedes einzelnen Threads, *cap* nur mit Vertrauenswürdigen zu teilen. Ohne Zutun eines Threads der Task ist es also keinem Thread einer anderen Task möglich Gums zu „stehlen“.

⇝ Unberechtigtes Verlängern der Lending-Frist: *Kann ein böswilliger Server geborgte Gums länger behalten, als sein Client erlaubt hat?*
Nur der Thread, der das Lending eingeleitet hat, ist berechtigt den Wert von $t_{lending}$ durch den Memory-Accounting-Manager verändern zu lassen. Der Empfänger der Gums hat somit keine Möglichkeit diese Frist selbst zu verlängern.

⇝ DoS-Angriff auf einen Donation-Partner bei nicht vertrauenswürdiger Donation:
Wie in Abbildung 4.3 erkennbar kann im Falle der nicht vertrauenswürdigen Donation ein Angreifer A_1 eine Donation ankündigen, und nach erfolgter Bestätigung durch den Partner B_1 nicht einleiten. B_1 wartet jetzt vergeblich auf die Donations-Bestätigung durch den Memory-Accounting-Manager M. Zunächst relativiert sich dieses Problem durch sinnvolles Setzen des Empfangstimeouts bei B_1. Weiterhin sollte B_1 Delegation (Kapitel 3.2.4) anwenden, damit B_1 für die Zeit des Timeouts nicht blockiert ist. B_1 kann zudem jeweils vor der Delegation eine Policy (z. B.: begrenzte Anzahl der Anfragen pro Thread oder Task pro Zeit) abprüfen und so Zombies vermeiden.

⇝ DoS-Angriffe auf den Memory-Accounting-Manager: *Wie sicher ist der Manager selbst?*
Der im Kapitel 1.2.3 beschriebene Angriff, dessen Verhinderung/Eingrenzung Ziel der Einführung von Memory-Accounting war, ist auch auf dem Memory-Accounting-Manager anwendbar, da dieser bedingt durch die vorgestellte Funktionsweise dynamisch Speicher allokieren muss. Es gibt genau drei Situationen, in denen das der Fall ist:

- beim Registrieren einer Task, also dem Eröffnen eines neuen Kontos
- beim Registrieren eines neuen Ressourcen-Providers
- beim Donieren/Lenden, also dem Anlegen eines neuen Kontingents

Das Problem wird gelöst, indem der Memory-Accounting-Manager ein Konto für sich selbst führt und bei Diensten, die zu den genannten drei Situationen führen, eine Donation fordert bzw. diese eigenmächtig durchführt.

⇝ Umgehen des Systems:
Das Memory-Accounting-System ist so konstruiert, dass es nicht zwangsläufig genutzt werden muss. Server, die dynamisch Speicher allokieren müssen, **können** es nutzen, um nicht angreif-

bar zu sein. Pager **können** das Kontingent ihrer Threads entsprechend verändern, aber sie müssen es nicht.
Der Grund dafür ist, dass Memory-Accounting nicht kostenlos ist. Es kostet IPC, Speicher und natürlich CPU-Zeit. Zudem ist das Verfahren zwischen vertrauenswürdigen Komponenten nicht notwendig.
Es bleibt daher noch zu klären, ob auch nicht vertrauenswürdige Komponenten es umgehen können.

Dazu werden folgende Szenarien betrachtet:

- Pager kollaborieren mit dem Angreiferthread
- ein Client will Serverdienste nutzen ohne zu donieren

Bekommt ein Angreiferthread Speicher von anderen Threads gemappt, ohne das diese sein Kontingent anpassen, so ist das unproblematisch, da auch diese Threads ihren Speicher in der Regel von einem Pager bekommen haben. Irgendwo in der Hierarchie existiert aber ein Pager (z. B. ein angepasster Sigma0), der Memory-Accounting unterstützt. So wird keiner unbeteiligten Task Speicher weggenommen.
Das zweite Szenario ist noch weniger kompliziert. Verlangt ein Server für die Diensterbringung eine Donation, dann muss der Client Gums übergeben. Tut er das nicht, kann der Server diesem den Dienst verweigern.

$\rightsquigarrow$ Fälschen der Capability *cap*:
Die vom Memory-Accounting-Manager pro Konto verwaltete Capability *cap* ist eine Zufallszahl, die von einem Angreifer erraten werden kann. Das ist ein grundlegendes Problem bei Capabilities. Es gibt aber Bestrebungen, mit Hilfe von L4.sec [Völ, Kau05] fälschungssichere Userland-Capabilities zu konstruieren. Daher erscheint es sinnvoll, *cap* später durch eine solche sichere Capability zu ersetzen.

Optimierungen

Viele der vorgestellten Abläufe lassen sich optimieren. Zum Beispiel kann ein Pager die Accountingdaten cachen und nur „von Zeit zu Zeit“ mit dem Accounting-Manager abgleichen. Es ist dabei auch möglich, Abfrage und Dekrementieren in einem Aufruf durchzuführen.
Beim Donieren/Lenden sollte die Ankündigung bzw. das Übermitteln der Capability *cap* zusammen mit dem Aufruf der Serverdienstleistung erfolgen.
Vertraut der Server B_i dem Client A_1, kann auch auf die Donationbestätigung durch M verzichtet werden, indem B_i auf diese einfach nicht wartet.

In allen genannten Fällen kann IPC eingespart werden.

Kosten

Tabelle 4.1 zeigt die Anzahl zusätzlicher IPC-Nachrichten für die beiden Donation-Protokolle. Dabei wird von der optimierten Variante (Übermittlung von *cap* bzw. der Ankündigung zusammen mit dem Aufruf) ausgegangen. Hinter dem Long-IPC verbirgt sich jeweils die

Tabelle 4.1: Memory-Accounting: IPC-Kosten

Aktion	Short-ICP	Long-IPC
Trusted-Donation/Lending-Protokoll	1	1
Untrusted-Donation/Lending-Protokoll	2	1

Donation-Anweisung an den Memory-Accounting-Manager.
Die zusätzlichen IPC-Kosten bei der Speicherallokation und bei Pagefaults werden an dieser Stelle nicht genau betrachtet, da diese wesentlich von der Implementation und weiteren Optimierungen abhängen. Vermutlich werden bei der Speicherallokation je ein Long- und ein Short-IPC notwendig sein, wenn der Pager dem Memory-Accounting-Manager anweist, das entsprechende Kontingent zu dekrementieren, und dieser daraufhin eine Bestätigung schickt.

Der Speicherbedarf gliedert sich in drei Teile (die genauen Kosten sind in Tabelle 4.2 aufgelistet):

1. fixer Bedarf pro Konto bestehend aus:
   - KontoID *tid*
   - Capability *cap*
   - persönlichem Kontingent (Daten siehe nächster Punkte)
2. variabler Bedarf der Kontingenteliste bestehend aus folgenden Daten pro Kontingent:
   - KontingentID *cid*
   - Kontingentwert c
   - neue Lendingfrist $t_{lending}$
   - Valid-Flag *valid*
3. variabler Bedarf der RP-Liste bestehend aus folgenden Daten pro Ressourcen-Provider:
   - Ressourcen-Provider-ID *rpid*
   - Wichtung w
   - Abwertung *sub* (pro Kontingent)

4.1.5 Memory-Accounting am Beispiel DOpE

Das Desktop Operating Envivonment [Fes02, FH03] ist ein echtzeitfähiges Fenstersystem für DROPS. DOpE verwaltet sogenannte Widgets. Das sind die kleinsten Elemente der GUI, wie z. B. Fenster, Rahmen, Buttons, Scrollbalken, u. s. w. Clients können mittels dieser Widgets Benutzeroberflächen erstellen, die von OPpE gezeichnet werden.

Tabelle 4.2: Memory-Accounting: Speicher-Kosten

Daten	**Speicherbedarf**
KontoID *tid*	4 Byte (int)
Capability *cap*	4 Byte (int)
KontingentID *cid*	4 Byte (int)
Kontingentwert *c*	4 Byte (int)
neue Lendingfrist $t_{lending}$	4 Byte (int)
Valid-Flag *valid*	1 Byte (char)
Ressourcen-Provider-ID *rpid*	4 Byte (int)
Wichtung *w*	4 Byte (int)
Abwertung pro Kontingent *sub*	4 Byte (int)

DOpE muss für jedes Widget, dass einer seiner Clients gezeichnet haben möchte, dynamisch Speicher allokieren. Damit ist ein typischer DoS-Angriff möglich:
Ein bösartiger Thread lässt DOpE solange neue Widgets erzeugen, bis dessen Speicher erschöpft ist.
DOpE eine Policy der Art: maximale Anzahl an Widgets pro Thread, oder besser: maximale Menge an Speicher pro Thread, durchsetzen zu lassen hilft nicht gegen DDoS-Angriffe und schränkt zudem die Funktionalität des Systems stark ein.

Mit Hilfe von Memory Accounting lässt sich dieses Problem auf einfache Art und Weise lösen:

- Da DOpE zur Trusted Computing Base gehört, kann das Trusted-Donation-Protokoll (Abbildung 4.2) angewendet werden. Dazu übermittelt der Client die Capability *cap* an DOpE, sobald ein oder mehrere Widgets gezeichnet werden sollen.
- DOpE errechnet nun den benötigten Speicherbedarf und lässt sich vom Memory-Accounting-Manager die entsprechende Menge an γ „überweisen“. Dabei wird auf dem Speicherkonto von DOpE ein Kontingent des Clients angelegt.
- Dieses Kontingent kann DOpE nun nutzen, um den benötigten Speicher zu allokieren.

Clients müssen DOpE also indirekt den Speicher donieren, der für die Verwaltung ihrer Widgets benötigt wird. Damit ist DOpE, auf die beschriebene Art, nicht mehr angreifbar.

4.1.6 Zusammenfassung

Es gibt verschiedene mehr oder weniger gute Ansätze zur Begrenzung von DoS-Angriffen auf den Nutzerspeicher. Memory-Accounting, das viele Elemente der ökonomischen Modelle enthält, wurde hier ganz bewusst eingeführt, weil sich dieses Konzept in erweiterter Form auch zur Verwaltung verschiedener anderer Ressourcen nutzen lässt.

Diese Erweiterung erfolgt nun im folgenden Kapitel 4.2.

4.2 Der γ-Bank-Manager „Dealman“

Nachdem mit Memory-Accounting ausführlich ein Konzept zur Vermeidung bzw. Eingrenzung von DoS-Angriffen auf den Hauptspeicher vorgestellt wurde, soll das Konzept nun verallgemeinert werden.

4.2.1 Motivation

Warum sollte Accounting einzig und allein für die Verwaltung von Speicher eingesetzt werden? Im vorherigen Kapitel wurde deutlich gemacht, wie mächtig ein Bezahlsystem sein kann und welche vielfältigen Möglichkeiten es bietet. Man stelle sich Ressourcen wie CPU-Zeit, Speicher, Rechte zum Erzeugen von Tasks und Threads, Rechte zum Kommunizieren mit anderen Threads, Zugriff auf das Netzwerk usw. als Waren in einem Wirtschaftssystem und Gums als dessen Währung vor. Die Rahmenbedingungen für ein solches erweitertes ökonomisches System sollen in diesem Kapitel geschaffen werden.

4.2.2 Das erweiterte ökonomische System

Die Grundidee entspricht im wesentlichen der aus Kapitel 4.1.4 mit dem Unterschied, dass es hier nicht mehr allein um Speicher geht, sonder um beliebige Ressourcen.

Erweiterung gegenüber dem Memory-Accounting

- Prinzipiell gibt es für jede Ressource einen Ressourcen-Provider, der die tatsächliche Vergabe und den Entzug durchsetzt. Ressourcen-Provider sind das allgemeine Äquivalent zu Pagern bei der Speicherverwaltung (wobei es neben dem Pager durchaus weitere Memory-Provider geben kann).
- Pro Knoten existiert ein γ-Bank-Manager, der sogenannte „Dealman“ (Deal-Manager), der verschiedene Konten sowie Unterkonten, die Kontingente, verwaltet. Dabei kann eine Task oder ein Thread z.B. jeweils Konten für den Speicher, für die CPU-Zeit, für bestimmte Rechte usw. besitzen. Diese Konten werden beim Start bzw. bei der Erzeugung von Tasks/Threads angelegt.
- Der „Dealman“ bietet eine weitere Funktion an: die lokale Überweisung. Dadurch wird es möglich, γ von einem Konto zu einem anderen des gleichen Inhabers zu übertragen. Hier ist zu beachten, dass Überweisungen nicht zwischen beliebigen Konten stattfinden dürfen, da ein potentieller Angreifer sonst sein gesamtes „Vermögen“ auf das Konto der Ressource überweisen könnte, die er angreifen will.
- Der „Dealman“ verwaltet für jeden Kontoinhaber eine Rechtematrix, die festlegt, zwischen welchen Konten Überweisungen stattfinden dürfen und die nachträglich nicht geändert werden kann.

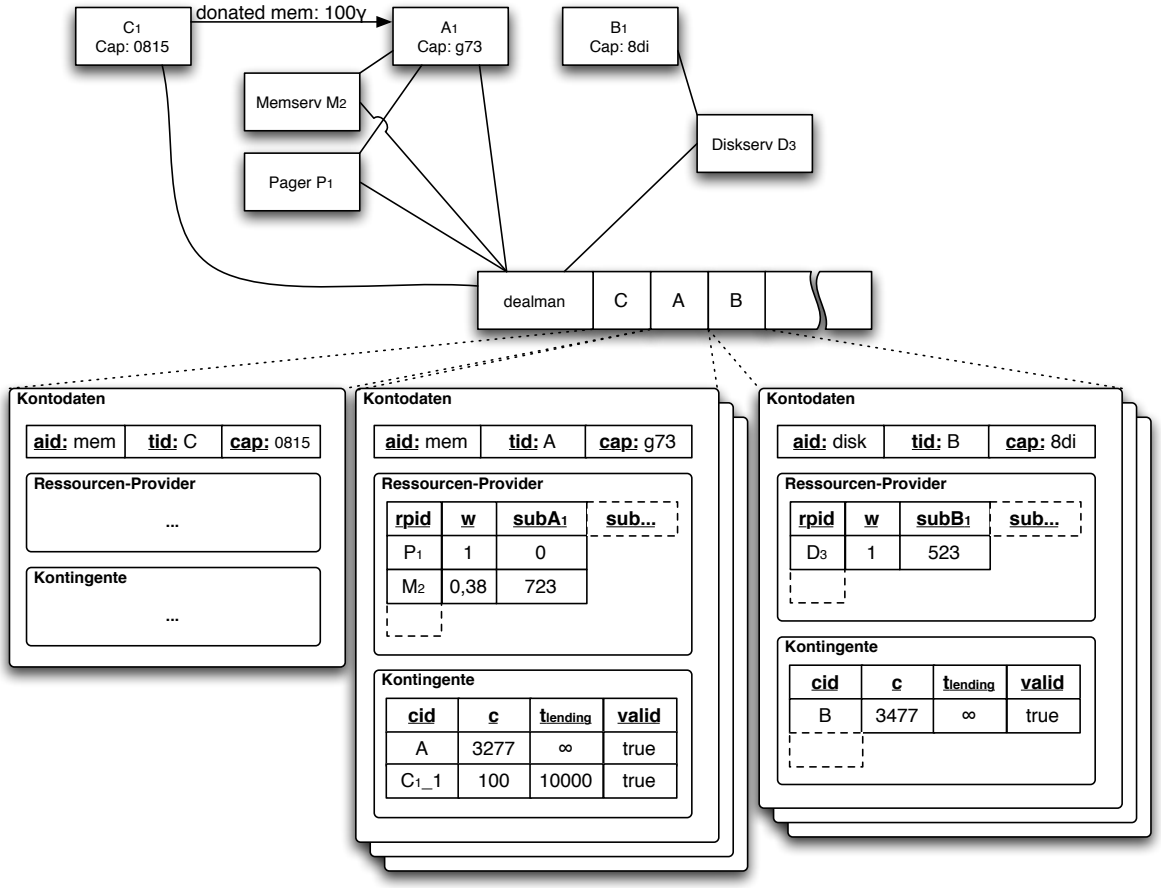

Abbildung 4.4: Beispielszenario des „erweiterten ökonomischen Systems“

4.2.3 Neue Möglichkeiten

Durch die genannten Erweiterungen kann nun ein vielfach erweiterter Handel mit Ressourcen stattfinden. Es bleibt zu untersuchen, in wie weit das sinnvoll und wünschenswert ist. Ein durchaus sinnvolles Szenario wird im folgenden beschrieben.

Task-bounded-Swapping im erweiterten ökonomischen System

Im Kapitel 4.1.2 wurde die Idee des Task-bounded-Swapping vorgestellt. Allerdings verwandelt diese Policy das Problem beschränkter Hauptspeicher lediglich in das Problem beschränkter Hintergrundspeicher. Im beschriebenen erweiterten ökonomischen System lässt sich dieses Manko nun abschwächen.
Für dieses Szenario wird davon ausgegangen, dass jeweils ein RP für den Hauptspeicher (Pager) und für den Hintergrundspeicher (Disk-Provider) existiert. Andere Ressourcen wie z. B. die beschränkte Ressource Busbandbreite werden auch an dieser Stelle nicht betrachtet. Typischerweise ist mehr Hintergrundspeicher als Hauptspeicher vorhanden, weshalb die Kosten pro Einheit (hier sinnvollerweise eine Speicher-Kachel) sich entsprechend unterscheiden.

So könnte z. B. eine Einheit Hauptspeicher beim Pager $500\,\gamma$ kosten, während eine Einheit Hintergrundspeicher beim Disk-Provider nur mit $5\,\gamma$ zu Buche schlägt.
Tasks, deren Pager Task-bounded-Swapping anwenden, benötigen jeweils zwei Konten beim „Dealman“: eines für den Hauptspeicher und eines für den Hintergrundspeicher.
Kommt es nun zu einem Pagefault und der Pager stellt fest, dass der Thread nicht genug Gums für eine weitere Speicher-Kachel besitzt, so weist er den „Dealman“ an, $5\,\gamma$ vom Hauptspeicherkonto auf das Hintergrundspeicherkonto zu übertragen, „kauft“ im Namen des Threads beim Disk-Provider eine entsprechende Einheit an Hintergrundspeicher und lagert die Daten einer Kachel des Threads auf diesen aus. Damit steht die Kachel wieder zur Verfügung und der Pagefault kann behandelt werden.

4.2.4 Notwendige Erweiterungen und Optimierungen

Spätestens dann, wenn die Konten des „Dealman“ für das Scheduling genutzt werden sollen, ist der Aufwand beim Abfragen und Verändern von Kontoständen durch IPC nicht mehr vertretbar. Folgende Erweiterung löst dieses Problem:

Beim Start des „Dealman“ (beim Booten des Systems) wird eine Menge vertrauenswürdiger Ressourcen-Provider (Pager, Scheduler, Taskmanager ...) festgelegt, die bestimmte Daten der Kontostruktur schreib-/lesbar gemappt bekommen und so die Kosten für bereitgestellte Ressourcen selbst verbuchen können. Im Kapitel 5.2.2 wird deutlicher, warum das notwendig ist.

4.2.5 Integration in DROPS

In DROPS werden Ressourcen von Ressourcenmanagern verwaltet. Threads können bei diesen Ressourcenmanagern Reservierungen vornehmen. Die einfachste Integration des vorgestellten Ressourcen-Accounting-Systems besteht darin, dass diese Ressourcenmanager die „Dealman“-Konten bei Entscheidung über Ressourcenreservierungen als maßgebliches Kriterium heranziehen.

4.2.6 Zusammenfassung

Der γ-Bank-Manager „Dealman“ schafft durch der Einführung von Ressourcenkonten die Grundlage für ein ökonomisches System, das folgende Möglichkeiten bietet:

- „Kontrollierte Bereitstellung“ beliebiger Ressourcen, in Zusammenarbeit mit deren Ressourcen-Providern und damit die Begrenzung der meisten DoS-Probleme
- Handeln mit Ressourcen gleichen oder ähnlichen Typs

5 Nutzung des „Dealman“ für andere Ressourcen

5.1 Vermeidung von Angriffen auf den Kernspeicher

Der im Kapitel 1.2.3 beschriebene Angriff auf den Kernspeicher durch Überfüllen der Mapping-Datenbank illustriert ein grundlegendes Problem: Der Kern muss um bestimmte Aufgaben zu erfüllen, dynamisch Speicher allokieren.

5.1.1 Grundlegende Lösung

In L4.sec [Völ, Kau05] wird dazu ein Objekt namens Kernel-memory eingeführt (Kapitel 2.2.6). Diese spezielle Art des Donierens löst das Problem mit der Mapping-Datenbank. Jede Task besitzt eine bestimmte Menge an Kernspeicher. Ist dieser belegt, können erst dann wieder Speichermappings durchgeführt werden, wenn dem Kern Flexpages doniert werden.

5.1.2 Problem des begrenzten Kernspeichers

Ein weiteres Problem besteht darin, dass je nach Hardwarearchitektur die Menge des Speichers, der als Kernspeicher genutzt werden kann, begrenzt ist. Liegt diese Grenze unter der gesamt verfügbaren Menge an Hauptspeicher, dann stellt der Kernspeicher eine spezielle, nicht replizierbare, beschränkte Ressource dar, die „kontrolliert bereitgestellt“ werden muss.

Kernspeicher als handelbares Gut

Diese kontrollierte Bereitstellung kann durch Nutzung des γ-Bank-Managers „Dealman“ sowie einem neuen Ressourcen-Provider durchgesetzt werden. Dazu wird ein Kernspeicher-Konto beim „Dealman“ angelegt. Der Ressourcen-Provider für den Kernspeicher, ist von nun an der Einzige, der Nutzerspeicher in Kernspeicher konvertieren darf und bietet genau das als Dienst an. Ein Thread kann diesen Dienst nutzen, sofern sich genug Gums auf dem entsprechenden Kernspeicherkonto befinden. Damit ist auch eine kontrollierte Rücknahme von konvertierten Speicherseiten und das Borgen (Lending) von Kernspeicher möglich.

5.1.3 Zusammenfassung

Um Angriffe auf den Kernspeicher zu verhindern, sind grundlegende Veränderungen am Kern nötig, die in [Kau05] beschrieben und teilweise implementiert sind. Diese Veränderungen ermöglichen zusammen mit dem Accounting eine Vermeidung von DoS-Angriffen auf den Kernspeicher.

5.2 Task- und Threadverwaltung

Die im Kapitel 1.2.4 beschriebenen Angriffe lassen sich auf zwei Hauptprobleme reduzieren:

1. die beschränkte Ressource Anzahl der Tasks,
2. die beschränkte Ressource CPU-Zeit.

Problem 1 lässt sich durch Einführung eines neuen Ressourcen-Providers für Tasks lösen. Dieser führt die Erzeugung von Tasks kontrolliert durch.
Zur Lösung des zweiten Problems müssen Veränderungen am Ressourcen-Provider der Ressource CPU, dem Scheduler, vorgenommen werden.

5.2.1 Der Taskmanager $\tau 0$

$\tau 0$ ist ein Server, dessen Dienstleistung das Erzeugen von Tasks ist.

Systemstart

Beim Start von $\tau 0$ (beim Systemstart) erzeugt dieser alle TaskIDs, indem er alle Tasks inaktiv startet. Eine TaskID ist dabei eine Repräsentation für das Recht, diese Task zu erzeugen. Damit ist $\tau 0$ ab diesem Zeitpunkt als einziger in der Lage neue Tasks zu erzeugen und kann dieses kontrolliert tun. Dazu verwaltet $\tau 0$ einen Taskbaum, der die Information enthält, aus welcher Task heraus eine andere erzeugt wurde.

Als nächstes legt $\tau 0$ für sich selbst beim „Dealman“ verschiedene Konten für alle kontrolliert verwalteten Ressourcen (Speicher, CPU-Zeit, Disk, Tasks, etc.) an, ermittelt das Angebot jeder Ressource und schreibt sich selbst den kompletten Gegenwert an Gums auf seine jeweiligen Konten gut. Das ist möglich, da $\tau 0$ einer jener im Kapitel 4.2.4 angesprochenen vertrauenswürdigen Ressourcen-Provider ist.

$\tau 0$ trägt sich nun in die Wurzel des Taskbaums ein.

Die ersten weiteren Tasks werden von $\tau 0$ erzeugt. Darunter kann sich beispielsweise eine Shell befinden. Für jede Task, die $\tau 0$ erzeugt, werden die Konten beim „Dealman“ angelegt. $\tau 0$ verteilt die aus der Angebotsermittlung erhaltenen Gums dabei fest vorgegeben auf die ersten Tasks.

Ab diesem Zeitpunkt können Threads mit Hilfe von $\tau 0$ weitere Tasks erzeugen.

Taskerzeugung/-beendigung

Der Thread T_1 möchte eine neue Task S erzeugen.
Ablauf:

- T_1 stellt eine entsprechende Anfrage beim Taskmanager $\tau 0$.
- $\tau 0$ prüft durch Abfrage des Task-Kontostandes von T beim „Dealman“, ob noch Tasks erzeugt werden dürfen.

- Falls ja passiert folgendes:
  - $\tau 0$ legt beim „Dealman“ neue Konten für die neue Task S an (Taskkonto, Speicherkonto, ...).
  - $\tau 0$ überträgt eine bestimmte Menge an Gums der Konten von T auf die entsprechenden Konten von S.
  - $\tau 0$ reduziert das Task-Konto von T um den Preis für eine neue Task.
  - $\tau 0$ trägt S unterhalb von T in den Taskbaum ein.
  - $\tau 0$ erzeugt S.

Invers dazu erfolgt das Beenden einer Task:

- Beendet der letzte Thread von S, so wird das z. B. durch einen Servicethread, an $\tau 0$ gemeldet.
- $\tau 0$ durchsucht den Taskbaum nach dem Erzeuger bzw. dem nächst höheren „Verwandten“ von S und findet T.
- $\tau 0$ erhöht das Task-Konto von T um den Preis für eine neue Task.
- $\tau 0$ überträgt alle Gums der Konten von S auf die entsprechenden Konten von T.
- $\tau 0$ löscht alle Konten von S beim „Dealman“.
- $\tau 0$ entfernt den Eintrag für S aus dem Taskbaum.

Tasks vs. Threads

In L4V2 können zu jeder Task maximal 128 Threads gehören. Diese werden beim Erzeugen einer neuen Task angelegt und können mittels `lthread_ex_regs` gestartet werden. Damit ist die Zahl der möglichen Threads, die ein Angreifer erzeugen kann, bei Nutzung von $\tau 0$ ebenfalls kontrollierbar und muss hier nicht gesondert betrachtet werden. In späteren Versionen von L4 gibt es die Beschränkung auf 128 Threads pro Task nicht mehr. Dann stellt die Größe des dafür vorgesehenen Hauptspeichers eine obere Grenze für die Anzahl der Threads dar und macht diese wiederum zu einer beschränkten Ressource. Die Lösung dieses Problems ist analog zu der, in diesem Kapitel vorgestellten, kontrollierten Taskbereitstellung.

Integration in DROPS

$\tau 0$ kann durch Erweiterung des mit dem L4Env [Gro03] eingeführten Taskserver „simple_ts“ geschaffen werden.

5.2.2 Scheduling

Mit der im letzten Kapitel (Kapitel 5.2.1) beschriebenen Kontrolle der Anzahl an Aktivitäten geht auch eine Eingrenzung der DDoS-Möglichkeiten einher. Allerdings ist es einem Angreifer noch immer möglich 128 Angreiferthreads zu starten. Außerdem besteht nach wie vor das Problem des Missbrauchs von Time-Slice-Donation (Kapitel 1.1).

Die zur Verfügung stehende CPU-Zeit ist hier der kritische Punkt. Der Scheduler soll sicherstellen, dass jeder Thread entsprechend seiner Priorität Rechenzeit bekommt. Lässt sich das durch Starten neuer Threads umgehen, ist dieses Ziel verfehlt.
Es wird daher in diesem Kapitel beschrieben, wie das „erweiterte ökonomische System“ zur „kontrollierten Bereitstellung“ der Ressource CPU genutzt werden kann.

Zeitscheibenlimitierung

Jeder Thread läuft in L4V2 auf einem Scheduling-Kontext. Dieser Kontext bestimmt unter anderem die für das betrachtete Problem relevante aktuelle Zeitscheibe. Im Folgenden wird daher nur noch von Zeitscheiben gesprochen. Echtzeitthreads können entsprechend ihrer obligatorischen und optionalen Teile mehrere solcher Zeitscheiben besitzen [HLR+01].
Daraus ergibt sich die Idee, die Anzahl der von einem Thread ausgehenden Zeitscheiben zu begrenzen. Dazu wird ein weiteres Konto beim „Dealman“ eingeführt: das CPU-Konto. Das Budget einer Task ist dabei ein Maß für die Anzahl zur Verfügung stehender Zeitscheiben, wobei aber im Falle von Nicht-Echtzeit-Threads nur jeweils eine davon für den „persönlichen Gebrauch“ vorgesehen ist.
Konkret bedeutet das: Das CPU-Budget legt fest, wie viele lauffähige Threads aus einer Task heraus erzeugt werden können.

Erweiterung des Scheduling

Dazu muss der Scheduler vor dem Aktivieren eines neuen Threads prüfen, ob dessen CPU-Budget groß genug ist.

Taskerzeugung

Ergänzend zum im Kapitel 5.2.1 beschriebenen Vorgang der Taskerzeugung muss $\tau 0$ für jeden der 128 Threads der neuen Task ein CPU-Konto beim „Dealman“ anlegen. Abgesehen vom ersten Thread beträgt deren Budget 0.

Threaderzeugung

T_1 startet mittels `lthread_ex_regs` einen neuen Thread T_2. Damit dieser lauffähig ist, muss T_1 entsprechend dem aktuellen Preis einer Zeitscheibe Gums auf das CPU-Konto von T_2 übertragen lassen, damit dieser vom Scheduler aktiviert werden kann.

Policy im Kern

Scheduling findet im Kern statt, da für das Umschalten von Thread-Kontexten und Adressräumen die entsprechenden Rechte benötigt werden. Die Entscheidung, wie hoch das CPU-Budget sein muss, damit ein Thread aktiviert werden darf, ist eine Policy. In einem Mikrokern-basierten System sollen Policies aber so weit möglich im Userland durchgesetzt werden.
Um dennoch eine sinnvolle Zeitscheibenlimitierung durchführen zu können ohne dabei gegen die Konstruktionskonventionen von Mikrokern-basierten Systemen zu verstoßen, kann ein Usermode-Scheduling eingeführt werden. Dazu wird, aus Sicht des ökonomischen Systems, nicht mehr der L4-Scheduler als CPU-Ressourcen-Provider angesehen, sondern ein Server im Usermode. Dieser ist verantwortlich dafür zu sorgen, dass der L4-Scheduler nur die Threads

auswählen kann, deren CPU-Konto entsprechend gefüllt ist.

Eine Möglichkeit, das zu tun besteht darin, Thread-Prioritäten zu verändern, bei unzureichendem CPU-Budget abzusenken. Dazu wird der Usermode-Scheduler entweder periodisch oder bei jeder CPU-Kontenänderung vom „Dealman“ aktiviert und passt die Thread-Prioritäten entsprechend der Kontostände an. Das alles passiert völlig transparent für den Kern-Scheduler.

Time-Slice-Donation auf Nutzerebene

Da CPU-Zeit nun eingeschränkt als handelbares Gut angesehen werden kann, ist es möglich Gums vom CPU-Konto zu donieren. Das stellt eine indirekte Form von Time-Slice-Donation auf Nutzerebene dar.

5.2.3 Verhinderung von DDoS-Angriffen

Im Kapitel 2 wurde unter anderem gesagt, dass sich Distributed-Denial-of-Service-Attacken nur verhindern lassen, „indem es unmöglich gemacht wird, eine entsprechende Infrastruktur zur Koordination mehrerer Angreifer zu nutzen.“. Tasks und Threads stellen genau diese Infrastruktur dar. Durch die Kontrolle ihrer Erzeugung können DDoS-Angriffe verhindert werden.

5.2.4 Zusammenfassung

Mit der Einführung des Taskmanagers $\tau 0$ und der Nutzung des γ-Bank-Managers „Dealman“ wird die beschränkte Ressource Anzahl der Tasks zu einem handelbaren Gut im Sinne des ökonomischen Systems und dadurch gemäß Kapitel 2.1.1 „kontrolliert bereitgestellt“.

Das CPU-Konto und die beschriebenen Anpassungen am Scheduler und $\tau 0$ begrenzen zudem die beiden grundlegenden Probleme DDoS und Missbrauch von Time-Slice-Donation. Gegeben durch den CPU-Kontostand ist festgelegt, wie viele aktive Threads ein Angreifer erzeugen kann, wie viele Threads maximal an einer DDoS-Attacke beteiligt sind, und wie viele Threads dem Angreifer ihre Zeitscheibe donieren können. So ist es diesem nicht mehr möglich, mehr Rechenzeit als vorgesehen zu bekommen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

6.1.1 Richtlinien zur Konstruktion DoS-sicherer Server

Im Laufe dieses Belegs wurden Richtlinien zur Konstruktion DoS-sicherer Server entwickelt. Diese sollen an dieser Stelle zusammengefasst werden.

Verwenden von Delegationsmechanismen

Zunächst muss analysiert werden, wie lange der Server im Mittel für die Bearbeitung einer Anfrage benötigt.

- Multiple Delegation:
  Kommt dabei heraus, dass dafür eine Zeitscheibe nicht genügt, so sollte „multiple Delegation“ (Kapitel 3.2.4) genutzt werden. Dabei müssen ggf. Wettlaufsituationen beim Zugriff auf die Ressourcen des Servers vermieden werden.

- Time-Slice-Donation:
  Bei sehr langen Bearbeitungszeiten oder Delegationszeitproblemen (Kapitel 3.2.4), sollte der Server auf Time-Slice-Donation bestehen. Diese kann entweder direkt, wie im Kapitel 3.2.3 oder indirekt wie im Kapitel 5.2.2 erfolgen.

- Duale Delegation:
  In den meisten Fällen wird ein Server die Anfragebearbeitung innerhalb seiner Zeitscheibe schaffen. Dann genügt es auf „duale Delegation“ (Kapitel 3.2.7) zurückzugreifen.

Ressourcen-Donation

Bedingt die Art des angebotenen Dienstes ein dynamisches Allokieren von Ressourcen, so sollte der Server die entsprechende Menge an Gums für jede der Ressourcen vom anfragenden Client verlangen (Kapitel 4.1.4 und 4.2). Dabei ist Lending (Kapitel 4.1.4) der Donation immer vorzuziehen, da es vom Client weniger Vertrauen in den Server verlangt.

Sicherheit vs. Effizienz

Es sollte ein guter Kompromiss zwischen notwendiger Sicherheit und den Kosten dafür gefunden werden. So erscheint es z. B. als sinnvoll, nur dann eine Memory-Donation zu fordern, wenn der dynamisch allokierte Speicher für eine längere Zeit benötigt wird, wie es bei einem Namensdienst der Fall ist. Ähnlich verhält es sich bei der Delegation. Ist die ständige Verfügbarkeit des Dienstes eher unwichtig, kann ggf. komplett auf Delegationsmechanismen verzichtet werden.
Kostengünstige Policies vs. teure Mechanismen:

In unkritischen Bereichen kann es genügen „duale Delegation“ mit einer passenden Policy (z. B. die oft genannte Begrenzung der Anzahl an Anfragen pro Thread/Taks pro Zeiteinheit) einzusetzen.

6.1.2 Ungelöste Probleme

Einige Annahmen, die im Laufe dieses Belegs gemacht wurden führen zu Problemen.

Experimentelle Ansätze

An mehreren Stellen wird auf Arbeiten anderer verwiesen, die noch nicht stabil, b. z. w. nur teilweise implementiert sind. Damit ist z. B. die L4-Schnittstelle L4.sec gemeint, die insbesondere für die Abwehr von Angriffen auf den Kernspeicher zwingend benötigt wird.

SMP - Replikation der Ressource CPU

Dieser Beleg geht von Ein-Prozessor-Systemen aus. Delegation ist nur dann sinnvoll, wenn während der Zeit der Delegation keine weiteren Anfragen eintreffen können. Das Delegationszeitproblem (Kapitel 3.2.4) wird bisher gelöst, indem der Server entweder mit erhöhter Priorität läuft oder aber die Delegation innerhalb seiner oder einer donierten Zeitscheibe schafft. Existieren aber mehrere Prozessoren, dann funktioniert diese Vorgehensweise nicht mehr.
Vermutlich müssen im Falle mehrerer CPUs auch andere der vorgestellten Konzepte neu überdacht werden.

6.2 Ausblick

Die in diesem Beleg beschriebenen Konzepte stellen überwiegend erste Entwürfe dar. Eine praktische Umsetzung muss erst noch geleistet werden.
Keines der neu vorgestellten Konzepte existiert zur Zeit für L4/DROPS. Deshalb soll im folgenden ein Überblick gegeben werden, was zu tun ist.

6.2.1 Delegationsframework

Beide im Kapitel 3 beschriebenen Formen der Delegation müssen zur Zeit von den jeweiligen Servern „per Hand“ implementiert werden. Je nach Art des angebotenen Dienstes und Stils des jeweiligen Entwicklers werden sich diese Implementierungen stark voneinander unterscheiden. Um das zu vereinheitlichen und so den Anreiz der Nutzung von Delegation zu stärken wird die Schaffung eines entsprechenden Frameworks empfohlen, das dem Programmierer den größten Teil dieser Arbeit abnimmt. Dabei sollte auch Priority Remembrance (Kapitel 3.2.5) direkt umgesetzt und nicht dem Entwickler aufgebürdet werden. Da Delegation die wenigsten Veränderungen am bestehenden System erfordert wird empfohlen diese zuerst zu integrieren.

6.2.2 Implementation

γ-Bank-Managers „Dealman“

Um die Vorteile des erweiterten ökonomischen Systems nutzen zu können, muss der γ-Bank-Managers „Dealman“ (Kapitel 4.2) implementiert werden. Die angesprochenen Optimierun-

gen sollten unbedingt umgesetzt werden, damit der „Dealman“ von den Basisdiensten sinnvoll genutzt werden kann.

Taskmanager $\tau 0$

Zur Vermeidung von Aktivitätsbomben ist ein Taskmanager wie im Kapitel 5.2.1 beschrieben notwendig. Es wird empfohlen den mit L4env [Gro03] eingeführten Taskserver „simple_ts“, entsprechend zu erweitern.

6.2.3 Anpassung der Basisdienste

Die Ressourcen-Provider müssen die Dienste des „Dealman“ in Anspruch nehmen und entsprechend den Entzug und die Zuteilung von Ressourcen, wie im Kapitel 4.1.4 exemplarisch beschrieben, durchsetzen, damit das System von den neuen Möglichkeiten des Handels mit Ressourcen profitieren kann.

6.2.4 Untersuchung der Folgen von ökonomischen Systemen

Die Idee ein ökonomisches System innerhalb eines Betriebssystems zu schaffen wirft eine Menge an Fragen auf. Das Ziel dieses Belegs war es, Konzepte zu finden, die Denial-of-Service-Angriffen entgegenwirken. Das „erweiterte ökonomische System“ erfüllt diesen Zweck. Aber es hat weit mehr Potential. Die Möglichkeit mit Ressourcen zu Handeln, diese zu „verborgen“ und zu „verschenken“ kann auf unterschiedliche Weise genutzt werden. Durch die Einführung einer abstrakten „Währung“, der Gums (γ), ist es denkbar Markt- und Wirtschaftsstrukturen in Betriebssystemen zu schaffen.
Hier gibt es eine ganze Menge an Forschungspotential. Zum Beispiel:

- Wie lassen sich volks- und betriebswirtschaftliche Gesetze und Strukturen nutzen und welche Auswirkungen hat das auf die Softwareentwicklung?
- Wie effizient sind derartige Markt- und Wirtschaftsstrukturen.
- Sind derartige Markt- und Wirtschaftsstrukturen außerhalb von Schutzkonzepten überhaupt wünschenswert?

An dieser Stelle wird klar, dass dieses Thema deutlich über die Grenzen der Informatik hinausreicht und eine Brücke zu anderen Wissenschaften, insbesondere der BWL und VWL, schlägt.

Anhang A

Anhang

A.1 Durchsetzung von Time-Slice-Donation

Die Durchsetzung von Time-Slice-Donation (Kapitel 3.2.3) kann so erfolgen:

- Bei einer Anfrage gibt der Server die CPU mittels `thread_switch(nil)` ab, da er zunächst auf jeden Fall auf der eigenen Zeitscheibe läuft.
- Wird der Server wieder aktiviert, prüft er, wessen Zeitscheibe gerade aktiv ist. Ist es die eigene, doniert der anfragende Client definitiv nicht
  (vorausgesetzt $Prio_{server} <= Prio_{client}$) und der Server bricht die Abarbeitung ab.
- Bei sehr langen Bearbeitungszeiten sollte in bestimmten Zeitabständen immer wieder geprüft werden, wessen Zeitscheibe gerade aktiv ist. Ist es die eigene, wird die CPU abgegeben und bei der nächsten Aktivierung erneut geprüft. Ist dann wiederum der eigene Scheduling-Kontext aktiv, doniert der Client nicht!

Das ist nur eine erste Idee, die verdeutlichen soll, dass das Problem grundsätzlich lösbar ist!

Abbildungsverzeichnis

1.1 Beispiel für einen möglichen Missbrauch von Time-Slice-Donation 13

3.1 Time-Slice-Donation 20
3.2 Donation-Time-Expiration 21
3.3 Delegation 23
3.4 Priority-Remembrance 25
3.5 donierende multiple Delegation ohne „Donating-Call“ 26
3.6 donierende multiple Delegation mit „Donating-Call“ 27

4.1 Vom Memory-Accounting-Manager verwaltete Daten 36
4.2 Trusted-Donation-Protokoll: Donation mit vertrauenswürdigem Partner . . . 39
4.3 Untrusted-Donation-Protokoll: Donation mit nicht vertrauenswürdigem Partner 40
4.4 Beispielszenario des „erweiterten ökonomischen Systems“ 47

Tabellenverzeichnis

4.1 Memory-Accounting: IPC-Kosten 44
4.2 Memory-Accounting: Speicher-Kosten 45

Glossar

Aktivitätsbombe Eine Aktivitätsbombe ist ein Thread, der nichts anderes tut, als neue Threads zu starten, bzw. Tasks zu erzeugen. Dabei treten zwei Probleme auf: zum einen ist irgendwann die maximale Zahl an Aktivitäten des Systems erreicht und es können keine weiteren erzeugt werden, zum anderen verbrauchen diese Aktivitäten ggf. Rechenzeit und lasten das System aus. Der Name Aktivitätsbombe ist von dem Begriff Forkbombe abgeleitet, die ein Unixprogramm beschreibt, das fortwährend den Syscall fork() aufruft und somit das beschriebene bewirkt.

Borgen Lending

BWL Betriebswirtschaftslehre

Clans & Chiefs Clans & Chiefs ist ein L4-Konzept zur Schaffung von Schutzdomains, dass von Jochen Liedtke entwickelt wurde.

Client Dienstnutzer

DDoS Distributed-Denial-of-Service

Dealman Dealman steht für Deal-Manager. Es ist die Bezeichnung für einen allgemeinen Accounting-Manager, der vergleichbar ist mit einer Bank im Sinne des Finanzwesens. Der Dealman verwaltet γ (Gums) auf verschiedenen Konten und Unterkonten (Kontingenten).

Donation Übergeben einer Ressource an eine andere Task b. z. w. einen anderen Thread

DOpE Desktop Operating Environment

DoS Denial-of-Service

DROPS Dresden Realtime OPerating System

Echtzeit Rechtzeitigkeit

FIFO First In First Out (Auswahl- und Verdrängungsstrategie)

GUI graphical user interface - graphische Benutzerschnittstelle

Gums Gums sind die Währung im Kontext des eingeführten ökonomischen Systems. Gums ist ein Kunstwort, das nur im Plural existiert. Es heisst also sowohl 2 Gums wie auch 1 Gums. Das Symbol für Gums ist der griechische Buchstabe γ. Ja, ich hatte bei der Einführung dieser Währung Gummibären vor Augen :-).

IPC Inter-Prozess-Kommunikation

Kachel Im Kontext von virtuellem Speicher wird eine physische Speicherzelle als Kachel bezeichnet.

Kernelspace Speicherbereich, auf den nur der Kernel Zugriff hat.

Kontingent Kontingente sind eine Art Unterkonto bei einem Accounting-Manager (siehe Dealman). Kontingente werden beim Donieren und Lenden von γ (Gums) auf dem entsprechenden Konto des Empfängers angelegt. Der Sinn der Kontingente besteht darin, sowohl Servern als auch Clients eine bessere Kontrolle über donierte/geborgte γ zu ermöglichen.

L4 L4 ist der Name eines Mikrokerns der zweiten Generation. L4 wurde ursprünglich von Jochen Liedtke entwickelt und wird in Form von Michael Hohmuths Fiasco an der TU-Dresden eingesetzt und weiterentwickelt.

L4V2 L4V2 steht für die zweite Version des L4-Kerns. Diese Spezifikation bildet die Grundlage dieses Belegs

Lending zeitweises Übergeben einer Ressource an eine andere Task b. z. w. einen anderen Thread (auch Borgen)

LIFO Last In First Out (Auswahl- und Verdrängungsstrategie)

Mikrokern Mikrokerne sind kleine Betriebssystemkerne, die im Gegensatz zu monolitischen Kernen nur grundlegende Dienste wie Adressräume, Aktivitäten und Kommunikation zur Verfügung stellen.

Page siehe Seite

Pagefault Versucht ein Thread auf eine Speicherseite zuzugreifen, deren Kachel er nicht gemappt hat, oder die verdrängt ist, spricht man von einem Pagefault. Der Kern aktiviert in einem solchen Fall den Pager des Threads, der den Pagefault behandelt.

Pager Thread, der für die Behandlung von Pagefaults zustandig ist.

Policy Regelwerk

Ressource Betriebsmittel

Ressourcen-Provider Server, die grundlegende Ressourcen, wie Speicher, Netzbandbreite, CPU-Zeit, ..., verwalten, b. z. w. zur Verfügung stellen.

RP Ressourcen-Provider (sowohl Singular, als auch Plural)

Scheduling Auswahl eines bereiten Threads und Zuteilung der CPU an diesen.

Seite Im Kontext von virtuellem Speicher wird eine virtuelle Speicherzelle als Seite (auch Page) bezeichnet.

Server Diensterbringer

Swappen siehe Swapping

Swapping Stellt ein entsprechender Pager beim Behandeln eines Pagefaults fest, dass alle Kacheln belegt sind, so lagert er den Inhalt einer oder mehrerer Kacheln auf einen Hintergrundspeicher aus und bei Bedarf wieder ein. Dieser Mechanismus wird Swapping genannt.

Task Tasks sind im Kontext von L4 Adressräume. Zu jeder Tasks gehören 128 Threads.

Thrashing Thrashing ist die Situation, dass ein swappender Pager bei der Behandlung eines Pagefaults eine Kachel eines fremden Threads verdrängt.

Thread Aktivitätsträger

Totzeit Zeit, in der ein Server nicht empfangsbereit ist, da er gerade eine vorangegangene Anfrage bearbeitet. Clients, die in dieser Zeit Anfragen an den Server stellen werden in dessen Senderwarteschlange eingereiht.

Userland Eingeschränkte 'Umgebung', in der Nutzerprogramme laufen

Userspace Speicherbereich auf den Nutzerprogramme zugreifen konnen

VWL Volkswirtschaftslehre

Zombie Ein Zombie ist ein Thread, der vergeblich auf eine Nachricht von einem anderen Thread via IPC wartet und somit bis zum Ablauf seines Empfangstimeouts blockiert ist.

Literaturverzeichnis

[AH99] Au, Alan und Gernot Heiser: *L4 User Manual*. The University of New South Wales, Version 1.14 Auflage, 03 1999.

[dos] *http://de.wikipedia.org/wiki/Denial_of_Service*.

[Fes02] Feske, Norman: *DOpE - a graphical user interface for DROPS*. Diplomarbeit, Technische Universität Dresden, 2002.

[FH03] Feske, Norman und Herman Härtig: *DOpE - a Window Server for Real-Time and Embedded Systems*. Technischer Bericht, Technische Universität Dresden, 2003.

[glo] *http://www.cs.vu.nl/globe/*.

[Gro03] Group, Operating System Research: - *L4Env- An Environment for L4 Applications*. Technischer Bericht, Technische Universität Dresden, 2003.

[HLR+01] Hamann, Claude-Joachim, Jork Löser, Lars Reuther, Sebastian Schönberg, Jean Wolter und Hermann Härtig: *Quality-Assuring Scheduling - Using Stochastic Behavior to Improve Resource Utilization*. Technischer Bericht, Technische Universität Dresden, 2001.

[Hoh] Hohmuth, Michael: *The Fiasco kernel: System Architecture*.

[HRW+99] Härtig, Hermann, Lars Reuther, Jean Wolter, Martin Borriss und Torsten Paul: *Cooperating Resource Managers*. Technischer Bericht, Technische Universität Dresden, 1999.

[Kau05] Kauer, Bernhard: *L4.sec Implementation Kernel Memory Management*. Diplomarbeit, Technische Universität Dresden, 2005.

[Lie98] Liedtke, Jochen: *Lava Nucleus (LN) Reference Manual - 486 Pentium PentiumPro - Version 2.2*, 1998.

[Lie99] Liedtke, Jochen: *L4 Version X in a Nutshell*. Technischer Bericht, Universität Karlsruhe, 1999.

[LJI97] Liedtke, Jochen, Trent Jaeger und Nayeem Islam: *Preventing Denial-of-Service Attacks on a µ-Kernel for WebOSes*. Technischer Bericht, IBM Research Division - T.J. Watson Research Center, 1997.

[mai] *http://www.irc-mania.de/DDOS.php*.

[MD88a] Miller, Mark S. und K. Eric Drexler: *Comparative Ecology: A Computational Perspective*. The Ecology of Computation, 1988.

[MD88b] Miller, Mark S. und K. Eric Drexler: *Incentive Engineering: for Computational Resource Management*. The Ecology of Computation, 1988.

[MD88c] Miller, Mark S. und K. Eric Drexler: *Markets and Computation: Agoric Open Systems*. The Ecology of Computation, 1988.

[Not02] Nothnagel, Jörg: *Ressourcenverwaltung in DROPS*. Diplomarbeit, Technische Universität Dresden, 2002.

[PCT03] Popescu, Bogdan C., Bruno Crispo und Andrew S. Tanenbaum: *A Certificate Revocation Scheme for a Large-Scale Highly Replicated Distributed System*. Technischer Bericht, Dept. of Computer Science, Vrije University, Amsterdam, 2003.

[Pfi] Pfitzmann, Andreas: *Sicherheit in Rechnernetzen: Mehrseitige Sicherheit in verteilten und durch verteilte Systeme*.

[Reu05] Reusner, Rene: *Implementierung eines Echtzeit-IPC-Pfades mit Unterbrechungspunkten für L4/Fiasco*. Diplomarbeit, Technische Universität Dresden, 2005.

[SPvS04] Sivasubramanian, Swaminathan, Guillaume Pierre und Maarten van Steen: *Replicating Web Applications On-Demand*. Technischer Bericht, Dept. of Computer Science, Vrije University, Amsterdam, 2004.

[Ste04] Steinberg, Udo: *Quality-Assuring Scheduling in the Fiasco Microkernel*. Diplomarbeit, Technische Universität Dresden, 2004.

[SWH05] Steinberg, Udo, Jean Wolter und Hermann Härtig: *Fast Component Interaction for Real-Time Systems*. Technischer Bericht, Technische Universität Dresden, 2005.

[Völ] Völp, Markus: *L4.Sec Preliminary Microkernel Reference Manual*.