-Asynchronous Inter Process Communication-
Seminar SS 2004

vorgetragen von Steffen Knapp

8. Oktober 2004

Inhaltsverzeichnis

I Einleitung 3

II Kommunikationsmodell 4

1 Wer kommuniziert mit wem? 4

2 Synchron vs. Asynchron 4
2.1 Synchron 4
2.2 Asynchron 4

3 Datenübertragung 5

III Motivation für asynchrones IPC 6

IV Abstraktes Design 6

4 Optimierungen 6
4.1 Shared Memory 6
4.2 Co-Location 7
4.3 Per Receiver Proxy 7

V Implementierung in L4 7

5 Implementierungsdetails 7
5.1 Adressierung der IPC Nachrichten 7
5.2 Implementierung des Proxy 7

6 Asynchrones Senden 8

7 Asynchrones Empfangen 8
7.1 Beispiel 8

VI Implementierung bei Verisoft 8

8 Kompatibilität bezüglich Vamos / SOS **9**
8.1 Neues Prozessmodell 9
8.2 Datenübertragung 9

9 Die eigentliche Kommunikation **10**
9.1 ipc_send_async() 10
9.2 Replys 10
9.2.1 Trennung von Senden und Empfangen 10

VII Anhang **11**

10 Grundbegriffe **11**
10.1 Prozesse 11
10.2 Speicherverwaltung 11
10.3 Mode 12

11 Kostenfrage **12**

12 Techniken der Datenübertragung **12**

13 Wichtige Konzepte bei IPC **13**
13.1 Timeouts 13
13.2 Empfangsarten 13

VIII Literaturverzeichnis **14**

Teil I
Einleitung

Diese Ausarbeitung befasst sich mit IPC und geht dabei im Speziellen auf asynchrone Implementierungen ein.
Die Abkürzung IPC steht für Inter Process Communication, also der Kommunikation zwischen Prozessen (in diesem Fall speziell auf Mikrokerne zugeschnitten). Dabei werden minimale Grundkenntnisse in der Architektur von Mikrokernen vorausgesetzt [1].

Zuerst werden die Kommunikationsmodelle kurz vorgestellt. Im nächsten Kapitel wird der Einsatz des asynchronen Modells diskutiert. Dem abstrakten Design der Inter Prozess Kommunikation folgen Implementierungsdetails speziell auf den L4 Kernel bezogen. Aus aktuellen Gründen wird anschließend tiefer auf Details eingegangen; besonders im Bezug auf die Kompatibilität der asynchronen Kommunikation mit Entwicklungen des Lehrstuhls von Prof. W. J. Paul (VAMOS und SOS).

Die Einführung der IPC- und Memory Management-Grundbegriffe ist absichtlich in den Anhang verlegt worden. Dort können die elementaren Begriffe zum Verständnis dieser Ausarbeitung in stichpunktartiger Form nachgelesen werden. Einsteigern wird geraten, den Anhang zuerst zu lesen. Kenner der Thematik können gleich loslegen...

[1] Diese wurden im Laufe des Seminars von Andreas Steinel vermittelt

Teil II
Kommunikationsmodell

1 Wer kommuniziert mit wem?

Diese Ausarbeitung betrifft die Kommunikation zwischen Prozessen/ Threads über Adressraumgrenzen hinweg. Die Kommunikation innerhalb von Adressräumen wird nicht näher betrachtet, da Sender und Empfänger demselben Adressraum innewohnen und somit direkten Zugriff auf die Daten haben.

2 Synchron vs. Asynchron

Im Folgenden ist der Sender der initiale Sender, der so genannte Proxy die Zwischenstelle (siehe unten) und der Empfänger der eigentliche Adressat der Nachricht.

Als Parameter eines *ipc_send()* wird im Folgenden der Nachrichtenadressat übergeben. Im Falle eines *closed-wait* stellt der Parameter den Prozess/Thread dar, welcher vom Empfänger spezifiziert wurde. Ist bei einem *receive* kein Parameter angegeben, handelt es sich um ein *open-wait*.

2.1 Synchron

Bei der synchronen Inter Prozess Kommunikation leitet der Sender durch ein *ipc_send(to_receiver)* an den Kernel die Kommunikation ein. Durch einen Mode-Wechsel geht die Kontrolle an den Kernel. Dieser bewirkt die Blockierung des Senders, sprich: Es wird dafür gesorgt, dass der Sender nicht mehr gescheduled wird.

Der Empfänger kann zu einem beliebigen Zeitpunkt kundtun, dass er gerne IPC-Nachrichten empfangen möchte, und zwar mittels *ipc_receive()*. Abermals geht die Kontrolle an den Kernel, der die Daten vom Sender zum Empfänger überträgt und die Blockierung des Senders aufhebt.

2.2 Asynchron

Asynchrones IPC wird auf folgende Weise realisiert: Ein dedizierter Proxy wartet auf Nachrichten vom Sender (*ipc_receive(from_sender)*). Wenn der Sender nun

durch ein *ipc_send(to_proxy)* die Kommunikation einleitet, kann der Kernel die Daten sofort zum Proxy kopieren. Der Sender ist nur für die Übertragung der Daten selbst blockiert. Der dedizierte Proxy ist nun für die Zustellung der Nachricht an den Empfänger zuständig.
Er initiiert nun seinerseits die Kommunikation durch einen *ipc_send(to_receiver)*. Die Kontrolle geht an den Kernel, der Proxy wird blockiert und der eigentliche Empfänger kann zu einem beliebigen Zeitpunkt die Nachricht empfangen.

Die oben beschriebene synchrone Kommunikation findet also zweifach Anwendung: einerseits zwischen Sender und Proxy, andererseits zwischen Proxy und Empfänger.

Aufgrund besserer Übersicht wird im Folgenden nur von Kommunikation bzw. Datenübertragung zwischen Prozessen / Threads gesprochen. Dies entspricht dem logischen Kommunikationsweg. Die reale Kommunikation verläuft jedoch immer über den Kernel.

3 Datenübertragung

Daten können auf drei Arten übertragen werden: per Flexpages, String-Dopes oder per Register.

Flexpages sind dynamische Verweise auf Memory Pages (Unterteilung innerhalb des physikalischen Speichers). Flexpages können entweder an andere Prozesse übergeben (*donate*) oder mit anderen Prozessen geteilt werden (*share*). Beide Methoden werden durch Ändern des Adressmappings realisiert (*zero copy*) und sind dadurch sehr schnell.

Mit Hilfe von **String-Dopes** können Daten spezifiziert werden (Pointer auf Kribbelchen [2]). Diese werden durch Kopieroperationen zum Empfänger übertragen. Zuvor muss der Empfänger jedoch sicherstellen, dass Speicherplatz für den Puffer alloziert ist. Aufgrund der Kopieroperationen ist diese Art der Datenübertragung sehr langsam.

Kleinere Datenmengen können mit Hilfe von **Registern** übertragen werden. Dafür stehen 3 Register innerhalb der IPC Nachricht bereit. Diese Form der Datenübertragung ist schnell, jedoch in der Größe stark beschränkt.

[2] Kribbelchen ist die landläufige Bezeichnung für (Kartoffel-) Puffer. Eine Wette mit seiner Freundin zwingt den Verfasser an dieser Stelle zur Verwendung ihres Lieblingswortes ;-)

Teil III
Motivation für asynchrones IPC

Der wichtigste und offensichtlichste Vorteil liegt in der minimalen Blockierung des Senders (siehe Kommunikationsmodell). Dies ermöglicht eine effizientere Prozessausführung sowie eine höhere Verfügbarkeit des Senders.

Asynchrones IPC ermöglicht es, das Senden vom Empfangen der Nachrichten zu trennen. Auf die Vorteile, die diese Trennung konkret bietet, wird gegen Ende dieser Ausarbeitung noch einmal genauer eingegangen werden.

Desweiteren sind Optimierungen möglich. So kann beispielsweise durch mehrere dedizierte Proxies die Blockierung innerhalb von Nachrichten aufgelöst werden: Da jeder Proxy dabei für die Zustellung einer Nachricht zuständig ist, blockieren die Nachrichten sich nicht mehr gegenseitig.

Durch die Nutzung von asynchroner Inter Prozess Kommunikation treten allerdings auch Nachteile auf. So verdoppelt sich die Anzahl der Nachrichten.
Es müssen mehr Daten übermittelt werden (größerer Overhead) und die Nachrichtenwege sind länger (höhere Latenzzeiten). Zudem fällt Verwaltungsarbeit für den Proxy an.

Teil IV
Abstraktes Design

4 Optimierungen

4.1 Shared Memory

Der Begriff Shared Memory bezeichnet eine Region im physikalischen Memory, welche in die Adressräume verschiedener Prozesse gleichzeitig abgebildet wird. Somit haben die beteiligten Prozesse vollen Zugriff auf die Daten in dieser Region. Aus praktischen Gründen (Datenintegrität) sollten Daten nicht zwischen mehr als zwei Prozessen geteilt werden.

Angenommen, sowohl der Sender mit dem Proxy, als auch der Proxy mit dem

Empfänger besitzen jeweils ein solches Shared Memory, dann müssten die Daten für die Übertragung nur noch zwischen den beiden Shared Memory Regionen kopiert werden. Dies spart eine Kopie in Vergleich zum eben beschriebenen naiven Modell.

4.2 Co-Location

Die Idee den Proxy in den Adressraum des Senders zu integrieren basiert auf der konsequenten Weiterentwicklung der obigen Idee. Der Proxy ist dabei einfach als Thread implementiert und hat vollen Zugriff auf die Daten des Senders.

4.3 Per Receiver Proxy

Wie bereits unter 'Motivation' angesprochen, ist es sinnvoll, für jeden Empfänger einen Proxy zur Verfügung zu stellen. Dies hebt die Blockierung innerhalb der Nachrichten selbst auf und sorgt so für höhere Parallelität. Mittels co-location ist die Umsetzung dieser Optimierung einfach zu handhaben.

Teil V
Implementierung in L4

5 Implementierungsdetails

5.1 Adressierung der IPC Nachrichten

Threads sind systemweit eindeutig gekennzeichnet durch einen UID (Unique Identifier). IPC Nachrichten werden mit Hilfe des UID adressiert.

5.2 Implementierung des Proxy

Der Proxy ist im Adressraum des Senders als Thread untergebracht (*co-located*). Des weiteren steht pro Empfänger ein Proxy zur Verfügung (*per receiver proxy*). Wie bereits ausgeführt, spart dies die Datenübertragung zum Proxy und optimiert die Sendereihenfolge der IPC Nachrichten.

6 Asynchrones Senden

Der Sender möchte Daten an einen Thread, der sich in einem anderen Adressraum befindet, senden. Er teilt dem Proxy, welcher innerhalb des Sender-Adressraums residiert, diese Absicht mit. Der Proxy hat direkten Zugriff auf die Daten und ist von nun an für die Übermittlung zum Empfänger zuständig.

7 Asynchrones Empfangen

Neben der Möglichkeit, auf asynchronem Wege Nachrichten zu senden, kann es von Nutzen sein, Nachrichten auch auf asynchronem Wege zu empfangen. Dies bietet den Vorteil, dass die Daten bereits in den Adressraum des empfangen Prozesses transferiert werden können und dieser nur noch über deren Eingang informiert werden muss.

7.1 Beispiel

Angenommen, der Sender (UID 1) rechnet mit eine Antwort auf eine zu sendende IPC Nachricht. Er teilt dem Sende-Proxy (UID 2) mit, dass er eine Nachricht senden will und informiert ihn zusätzlich, dass er die Antwort vom Empfangs-Proxy (UID 3) erwartet. Dieser Empfangs-Proxy residiert wie auch der Sende-Proxy im Adressraum des Senders.

Der Sende-Proxy sendet nun die Nachricht an den Empfänger, ändert aber den Absender UID auf 3 (Empfangs-Proxy). Der Empfänger bearbeitet die Nachricht und schickt die Antwort an den vermutlichen Absender (UID 3).

Der Empfangs-Proxy ist nun dafür verantwortlich den ursprünglichen Sender über den Eingang der Antwort zu benachrichtigen. Auf die Daten kann dieser dann direkt zugreifen (selber Adressraum).

Dieses Konzept bietet einige potentielle Angriffsmöglichkeiten (beispielsweise durch das Ändern der Absenderadressen). Dies kann allerdings unterbunden werden, indem nur Änderungen der Absenderadresse auf UID's erlaubt werden, welche innerhalb des Adressraums des sendenden Threads liegen.

Teil VI
Implementierung bei Verisoft

8 Kompatibilität bezüglich Vamos / SOS

Der im Verisoft Projekt eingesetzte Kernel ist ein Dialekt von L4ka, getauft auf dem Namen Vamos (*Verified Architecture Micro-kernel Operating System*).
SOS steht für *Simple Operating System* und ist das auf den Kernel aufsetzende Betriebssystem. Als Programmiersprache kommt ein 'gesäubertes' C zum Einsatz, welches C0 genannt wird.

8.1 Neues Prozessmodell

Das im Verisoft Projekt eingesetzte Prozessmodell unterscheidet sich von demjenigen in L4. Während Prozesse bei L4 in Threads unterteilt werden können, sind sie in Vamos atomar. Aus diesem Grund ist es nicht möglich, Proxies im Adressraum des Senders unterzubringen (kein *co-location*). Dies hat zur Folge, dass Daten erst in den Adressraum des Proxy transferiert werden müssen, um sie von dort weiter an den Empfänger zu leiten.
Daher ist es sehr aufwendig, für jeden Empfänger einen Proxy zur Verfügung zu stellen (eingeschränktes *per receiver proxy*).

Das größte Problem bei einer performanten Implementierung von asynchronem IPC stellt die Übertragung der Daten zum Proxy dar.

8.2 Datenübertragung

Wie bereits erwähnt können auf drei verschiedenen Wegen Daten per IPC übertragen werden: Flex-Pages, String-Dopes und Register.

C0 Programme arbeiten nicht direkt auf dem Speicher, sondern auf Datenstrukturen. Aus der Sicht der Prozesse kann dabei keine Aussage darüber getroffen werden, wo im Speicher Platz für diese Datenstrukturen allokiert ist. Deshalb wird das **Flex-Page** Konzept in SOS / Vamos nicht eingesetzt.

Stattdessen werden die Daten mit Hilfe von **String-Dopes** weitergegeben. Theoretisch wäre es auch möglich, kleine Daten per **Register** zu übermitteln, dies ist jedoch z.Z. nicht implementiert.

C0 besitzt ein restriktives Typsystem. Alle verwendeten Typen sowie deren Speicherbedarf müssen zur Kompilierung der Programme bekannt sein. Davon sind insbesondere auch Pointer betroffen. Diese können nach ihrer Kompilierung nur noch auf einen Typ von Ziel zeigen, dessen Speicherplatz statisch ist. Somit haben Puffer in C0 eine fixe Größe.
Bei großen Datenmengen müssen die Daten deshalb auf mehrere kleine Nachrichten verteilt werden.

9 Die eigentliche Kommunikation

9.1 Mögliche Implementierung von *ipc_send_async()*

Eine Möglichkeit, asynchrones IPC zu implementieren, besteht darin, den Sender zu forken und den geforkten Sender als Proxy umzufunktionieren. Benutzt man *copy on write* bei beim Forken, kann man einen Proxy erzeugen, ohne alle Daten explizit per Kopieropperationen übertragen zu müssen. Der Proxy ist quasi ein Clone des Senders und kann ohne weiteres auch mehrere Nachrichten übertragen (große Datenmengen). Man erreicht also einen ähnlichen Zustand wie co-location bei L4.

9.2 Mögliche Implementierung von Replys

Die augenscheinlichste Möglichkeit, Antworten des Empfängers zuzustellen ist, diese an den Proxy zurückzuschicken, welcher die Antwort an seinen Vaterprozess weiterleitet.
Die zweite Option besteht darin, die Antwort direkt an den ursprünglichen Sender zurückzuschicken und den Proxy bei diesem Vorgang gar nicht zu beteiligen. Diese Methode bietet einen offensichtlichen Vorteil: Sie splittet Senden und Empfangen auf zwei Prozesse auf. Dazu ein Beispiel:

9.2.1 Trennung von Senden und Empfangen

Prozess A möchte an Prozess B eine Nachricht übermitteln. A sendet also ein *ipc_send(to_B)* und wird blockiert. Als nächstes wird Prozess B gescheduled, welcher seinerseits eine Nachricht an A senden möchte. Nach dem *ipc_send(to_A)* von B wird dieser Prozess ebenfalls blockiert. Selbst wenn die beide Prozesse ihre Blockierung durch Timeouts beschränken, kann diese Situation bei jedem weiteren Kommunikationsversuch auftreten. Es besteht also die Gefahr eines Deadlocks.

Benutzt man nun die oben beschriebene zweite Möglichkeit, Replys zu implementieren, tritt folgende Situation ein: Sender A forked sich zu A*. Der geforkte Prozess ist nun dafür zuständig, die Nachricht an B zu übertragen. A* ist nun blockiert. Wie im obigen Beispiel wird nun Prozess B gescheduled, welcher sich seinerseits ebenfalls forked zu B*. B* initiiert die Nachrichtenübermittlung an A. Prozess A kann nun die Nachricht annehmen, da das Senden ja von A* erledigt wird. Ebenso kann B die Nachricht von A* annehmen.

Teil VII
Anhang

10 Grundbegriffe

Im Folgenden werden wichtige Grundbegriffe geklärt. Die gegebenen Definitionen sind vom Abstraktionsgrad so gewählt, dass sie zum Verständnis der vorliegenden Arbeit ausreichen. Demjenigen, der sich gerne etwas näher informieren möchte, sei das Skript zur Vorlesung 'Informatik II' vom SS 2004 empfohlen.

10.1 Prozesse

- **Adressräume** sind Mengen von virtuellen Adressen. In einer 32 Bit Architektur entspricht dies 2^{32} Adressen.
- Ein **Prozess / Task** ist ein Programm in Ausführung (incl. Umgebung wie beispielsweise Register). Jeder Prozess arbeitet in seinem eigenen Adressraum.
- **Threads** sind Unterteilungen innerhalb von Prozessen.

10.2 Speicherverwaltung

Prozesse arbeiten mit individuellen Adressräumen. Diese Adressräume werden auf den physikalischen Speicher partiell abgebildet (gemappt). Da die Prozesse nur Zugriff auf Speicheradressen haben, für welche ein solches Mapping existiert, kann auf diese Weise eine Zugriffskontrolle für den physikalischen Speicher realisiert werden. Bei einem Zugriff auf Adressen, für die keine physikalische Speicherabbildung existiert, tritt ein *Pagefault Interrupt* auf.

10.3 Mode

Man unterscheidet zwischen zwei verschiedenen Modi: User Mode und Kernel Mode.

Im **User Mode** findet die unter 'Speicherverwaltung' beschriebene Adressübersetzung statt. Bei Schreib-/Lese- Operationen auf dem Speicher wird also die Adresse als virtuelle Adresse angesehen und in eine physikalische Adresse übersetzt.

Anders im **Kernel Mode**: Hier wird direkt mit Hilfe der Adresse auf den physikalischen Speicher zugegriffen.

Ihrem Namen entsprechend arbeiten die User Prozesse im User- und der Kernel im Kernel Mode.

11 Kostenfrage

Der Microkernel-Ansatz steht und fällt mit der Performance des IPC. Aus diesem Grund wird an dieser Stelle auf zwei Wechsel eingegangen, die Kosten verursachen. Diese Kosten sind im Sinne von Zeiteinheiten zu verstehen, die bei Auftreten der Wechsel für deren Bearbeitung nötig sind.

Als **Kontext-Wechsel** bezeichnet man den Wechsel von der Ausführung eines Prozesses zu der Ausführung eines anderen. Dies beinhaltet einen großen administrativen Aufwand (Sichern der alten und Laden der neuen Umgebung).

Mode-Wechsel sind Wechsel zwischen den oben beschriebenen Modi. Diese implizieren die Sicherung der Prozessumgebung.

12 Techniken der Datenübertragung

Zwei Techniken verdienen im Bezug auf IPC eine nähere Betrachtung:

Als **zero copy** werden Datenübertragungsvorgänge bezeichnet, bei denen die Daten nicht explizit kopiert, sondern mit Hilfe von Änderungen der Speicherabbildungen übertragen werden. Dabei werden lediglich die Abbildungen selbst geändert; diese sind für jeden Prozess in Form von Tabellen hinterlegt.

Bei **copy on write** werden die Daten *vorläufig* ebenfalls durch Änderung der Speicherabbildung übertragen. Vor der Änderung dieser Daten werden dann allerdings Kopien angelegt. Der Kopiervorgang wird also so lange hinausgezögert, bis die Daten verändert werden.

13 Wichtige Konzepte bei IPC

13.1 Timeouts

Im oben beschriebenen Modell kann der empfangende Prozess den sendenden beliebig lange blockieren. Um diese Sicherheitslücke zu schließen können so genannte 'Timeouts' spezifiziert werden. Dies sind einfache count-down Zähler, die bei Ablauf der Zeit die Blockierung mit einer Fehlermeldung abbrechen. Die IPC Nachricht gilt nach Abbruch als nicht übermittelt. Es kann gegebenenfalls versucht werden die Übertragung zu einem späteren Zeitpunkt zu wiederholen.

13.2 Empfangsarten

Ein potentieller Empfänger hat die Auswahl zwischen zwei Empfangsarten. Einerseits kann er von jedem Sender Nachrichten entgegennehmen (**open wait**), andererseits steht es dem Empfänger frei, einen speziellen Sender zu spezifizieren und nur von diesem Nachrichten zu empfangen (**closed wait**).

Teil VIII
Literaturverzeichnis

L4 User Manual
Alan Au, Gernot Heiser
School of Comuter Science and Engineering, The University of New South Wales
March 15, 1999

Diploma Thesis about Asynchronous IPC
Stefan Götz
System Architecture Group, University of Karlsruhe
May 30, 2003

L4 eXperimental Reference Manual
System Architecture Group
Dept. of Computer Science Universität Karlsruhe
October 2, 2003