Interprozesskommunikation IPC

Seminar Konzepte von Betriebsystem-Komponenten

Denis Koslowski
koslowski.d@web.de

04.07.2005

Interprozesskommunikation

Gliederung

1. Was ist IPC?

Wenn Prozesse untereinander kommunizieren, aufeinander warten oder Daten und Ressourcen austauschen müssen, findet so genannte Interprozesskommunikationen statt.

Linux ist ein Multitasking-System ist. Dazu gehört auch, dass die meisten der vielen Prozesse die nebeneinander herlaufen, so gut wie nichts voneinander darüber wissen, z.B. welche Daten diese enthalten. Realisiert wird das durch einen Speicherschutz. Dieser dient zum Schutz des Betriebssystems, denn würde ein Prozess auf die Speichersegmente eines anderen Prozesses zugreifen, könnte das die Stabilität der Programme oder gar des Gesamtsystems beeinträchtigen. Voraussetzung (hardwaremäßig) ist dafür eine Speicherverwaltungseinheit (MMU).

Dennoch gibt es Anwendungsfälle, wo man sich gerne einen "Tunnel" zu einem anderen Prozess graben muss. Diese Fälle sind:

- 1. Mehrere Prozesse müssen spezielle Daten gemeinsam verwenden.
- 2. Die Prozesse sind untereinander abhängig und müssen aufeinander warten.
- 3. Daten müssen von einem Prozess zu einem anderen weitergereicht werden.
- 4. Die Verwendung von System-Ressourcen müssen koordiniert werden.

Für solche Anwendungsfälle wurden bereits im System V (SysV) zuverlässige und bewährte Techniken entwickelt, die heute in vielen Unixen (u. a. auch Linux) implementiert sind. Diese Mechanismen werden unter dem Begriff "Interprozesskommunikation" (kurz IPC) zusammengefasst.

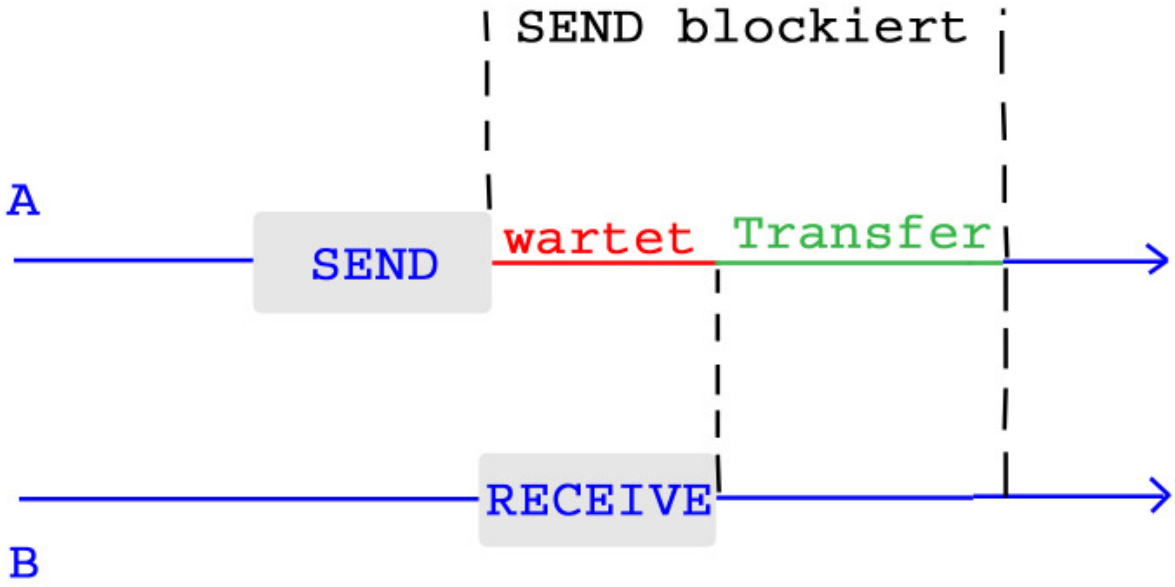
2. IPC: Datentransfer

2.1 Begriffe und Konzepte

Transfer von Daten von einem Prozess zu einem anderen Prozess kann auf 2 Arten erfolgen:

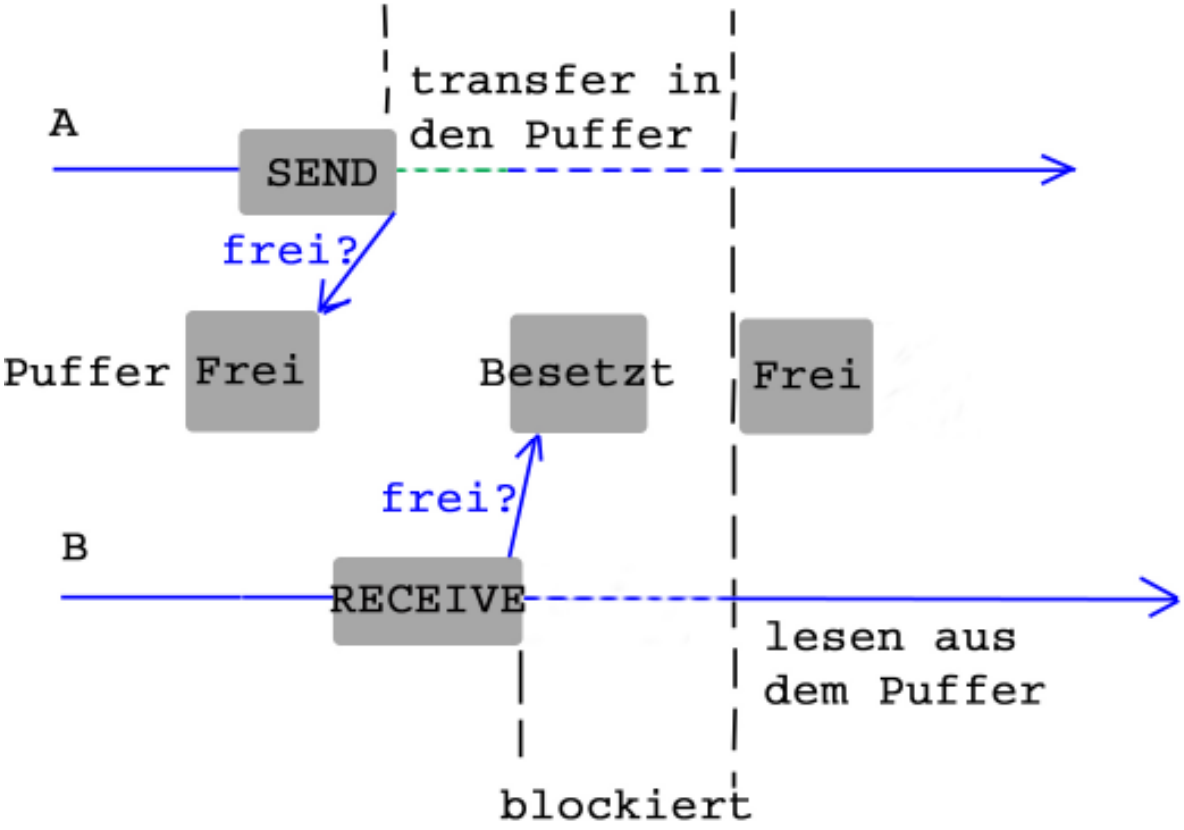
- gemeinsame Variablen, d. h. die Prozesse haben einen gemeinsamen Datenbereich
- einer gemeinsamen Kommunikationskanal, die Nachrichten werden versandt/empfangen

Der Kommunikationsverlauf kann synchron und blockierend oder asynchron (und blockierend oder nicht-blockierend)
Synchron und blockierend:



Die Daten werden direkt von Prozess A nach Prozess B transportiert.

asynchron und blockierend:



Prozess A schreibt die Daten in den Zwischenpuffer (bounded buffer). Der Prozess B kann aus dem Puffer erst dann lesen, wenn Prozess A in den Puffer nicht mehr schreibt.

2.2(Namenlose)Pipes

Eine Pipe ist ein unidirektionaler Kommunikationskanal zwischen zwei verwandten Prozessen (sie kann auch innerhalb eines einzigen Prozesses genutzt werden - inwiefern das sinnvoll ist, ist eine andere Frage). Sie haben Pipes sicherlich schon des Öfteren in der Konsole verwendet. Beispielsweise:

```
you@host > ps ax | less
```

Hiermit haben Sie in der Shell zwei Prozesse gestartet. Ein Prozess führt das Kommando ps ax aus und schreibt das Ergebnis gewöhnlich an die Standardausgabe. Durch die Pipe (|) wird allerdings diese Standardausgabe an den Prozess less weitergeleitet. less liest hierbei die Daten von der Standardeingabe ein und gibt aus, was ihm das Kommando ps durch die Pipe so schickt. Natürlich

erfolgt die Ausgabe mit dem gewöhnlichen Komfort, den less ihnen auch sonst bietet.
Wenn die Daten in beide Richtungen ausgetauscht werden sollen, muss eine zweite Pipe dazu verwendet werden. Sie dürfen sich eine Pipe gerne wie ein Rohr vorstellen, bei dem Daten in der einen Seite (Prozess A) hineingesteckt werden und bei Prozess B wieder herauskommen

Eine Pipe dient außer zur Kommunikation zwischen zwei Prozessen hervorragend zur Flusskontrolle. Dies daher, weil eine Pipe nur eine bestimmte Menge an Daten aufnehmen kann (normalerweise 4 Kbyte, 8 Kbyte oder 32 Kbyte; siehe Konstante PIPE_BUF in limits.h oder auch ulimit -p). Ist die Pipe (bzw. deren Puffer) voll, wird ein Prozess mindestens so lange angehalten, bis mindestens ein Byte aus der vollen Pipe gelesen wurde und wieder Platz vorhanden ist, um die Pipe wieder mit Daten zu befüllen. Andersherum dasselbe Bild, ist die Pipe leer, wird der lesende Prozess so lange angehalten, bis der schreibende Prozess etwas in diese Pipe schickt.
Es gibt also eine Schreibseite und eine Leseseite bei einer Pipe. Somit ist also nur eine Kommunikation in einer Richtung möglich (half-duplex). Sie können sich das Beispiel oben so vorstellen:

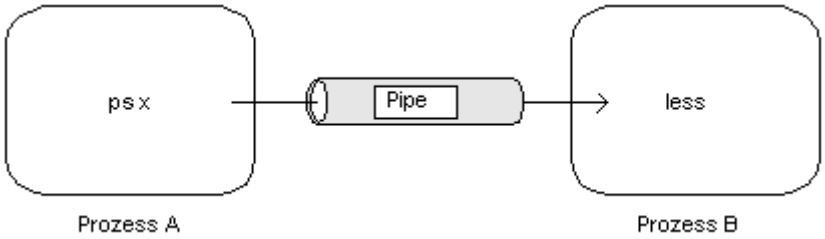


Abbildung 1 : Kommunikation zwischen zwei Prozessen über eine Pipe

2.3 Benannte Pipes (FIFO-Pipes)

Mit den Pipes können Sie allerdings nur mit den Prozessen kommunizieren, die miteinander verwandt sind. Also nur zwischen Vater- und Kind-Prozessen. Mit FIFOs (benannten Pipes) haben Sie nun die Möglichkeit, mit einem völlig fremden Prozess zu kommunizieren (Daten austauschen), da solche Pipes über einen Dateinamen angesprochen werden können.

Intern (wenn man den Kernel betrachtet) ist ein FIFO (FIFO = First In First Out) tatsächlich nichts anderes als die Implementierung einer namenlosen Pipe. Der Systemaufruf mkfifo() bedeutet also nichts anderes, als dass eine Pipe als Datei repräsentiert wird. Wer mittels pipe() eine Pipe erstellt, kreiert auch eine Datei,

wie mit mkfifo()! Diese sieht man zwar nicht, weil sie in einem Dateisystem versteckt ist, dass man nicht mounten kann - aber wer mal in */proc/1234567/fd* nachgesehen hat, sieht bei einigen "pipe:[1692]" mit pipe()-generierte Pipes. Ähnliches gilt übrigens auch für Sockets (TCP, auch PF_UNIX) ("socket:[456789]"). FIFOs werden wie erwartet mit z. B. */dev/initctl* angezeigt (siehe */proc/1/fd/*). PF_UNIX-Sockets werden ebenfalls als socket:[]-Objekt dargestellt, auch wenn diese eigentlich einen Dateinamen besitzen.

Auf der Shell lässt sich ein FIFO folgendermaßen erstellen:

```
you@host > mkfifo fifo1
```

Bei einem Blick ins aktuelle Arbeitsverzeichnis finden Sie das FIFO unter folgendem Eintrag:

```
you@host > ls -l
prw-r--r-- 1 tot users 0 2003-12-07 10:53 fifo1
```

Am p am Anfang erkennen Sie das FIFO. Sie könnten jetzt etwas in das FIFO schreiben:

Konsole 1

```
you@host > echo Der erste Eintrag in das FIFO > fifo1
```

Konsole 2

```
you@host > echo Der zweite Eintrag in das FIFO > fi-
fo1
```

Beide Dialogstationen blockieren im Augenblick und warten, bis die Daten im FIFO ausgelesen werden. Wir öffnen eine dritte Konsole und lesen ihn aus:

```
you@host > cat fifo1
Der zweite Eintrag in das FIFO
Der erste Eintrag in das FIFO
```

Natürlich müssen Sie auch die Zugriffsrechte für das FIFO vergeben, wer in dieses FIFO etwas schreiben und wer aus ihm lesen darf. FIFOs sind unidirektional. Die gelesenen Nachrichten werden aus dem FIFO gelöscht.

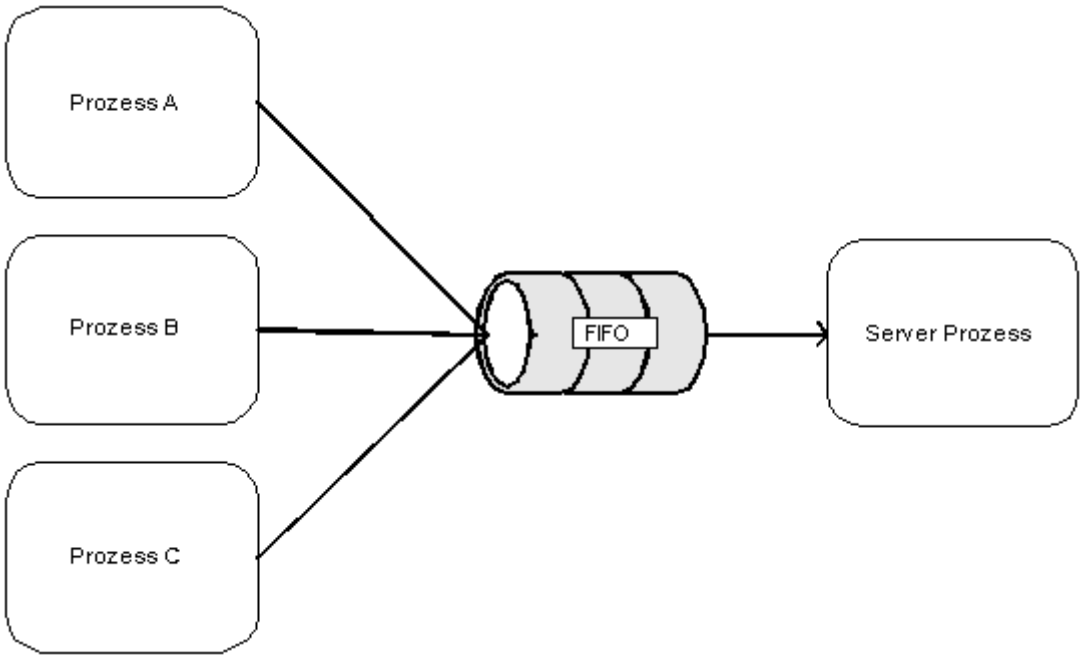


Abbildung 2 : Kommunikationsmodell eines FIFOs (First In First Out)

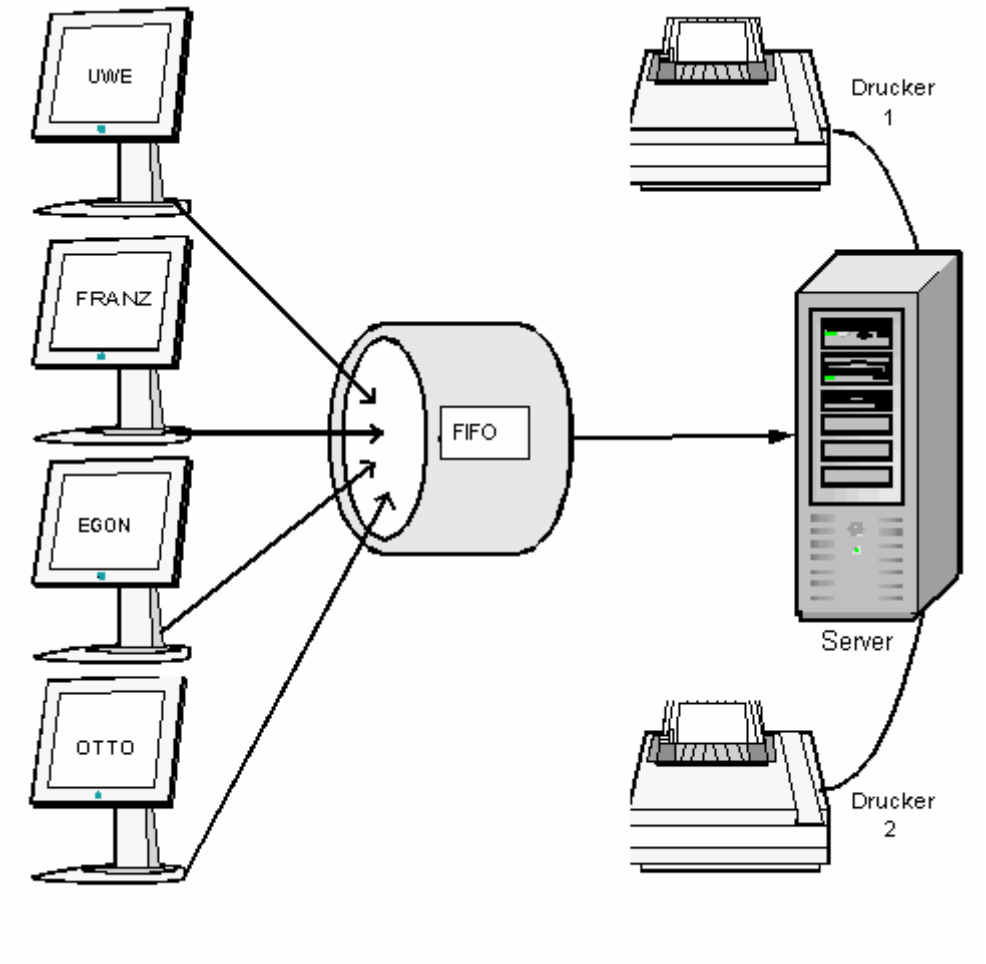


Abbildung 2.1: Typisches Client/Server-Beispiel mittels eines FIFOs

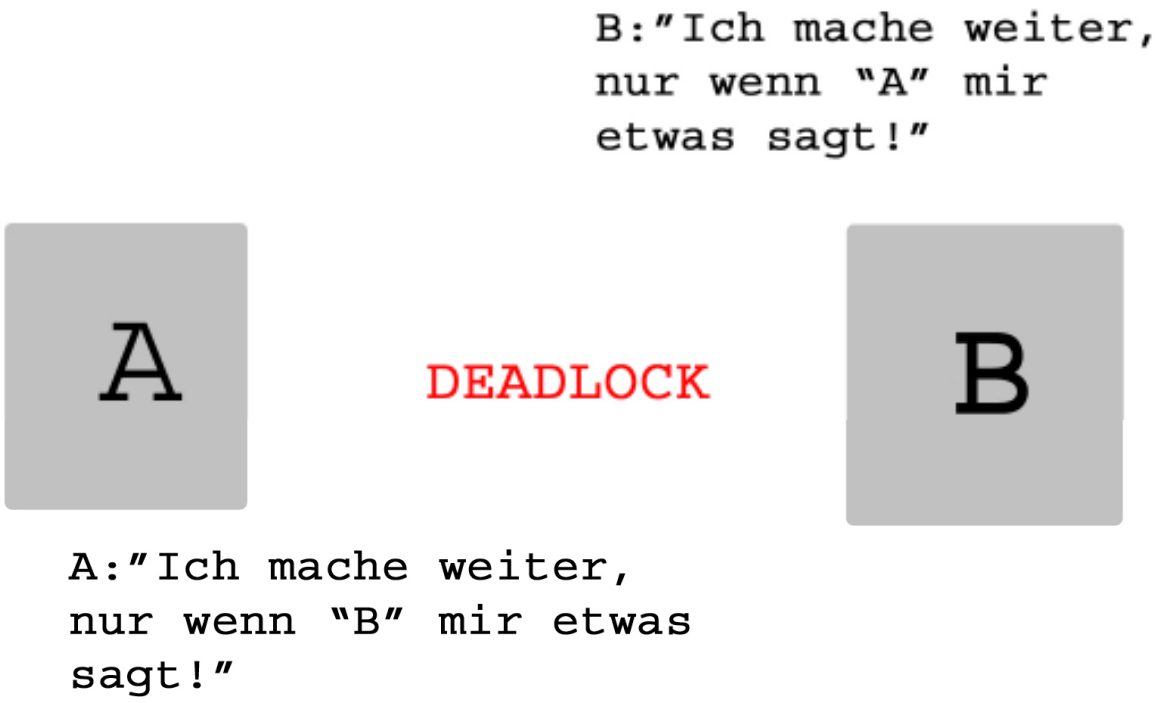
2.4 Shared Memory (Gemeinsamer Speicher)

Mit dem Shared Memory-Mechanismus können Sie mit mehreren Prozessen auf einen gemeinsamen Datenspeicherbereich zugreifen. Um dies zu realisieren, muss zuerst ein Prozess diesen gemeinsamen Datenspeicher anlegen. Anschließend müssen alle anderen Prozesse, die ebenfalls darauf zugreifen sollen, mit diesem Datenspeicher bekannt gemacht werden. Dies geschieht, indem der Speicherbereich im Adressraum der entsprechenden Prozesse eingefügt wird. Ebenfalls muss hierbei den Prozessen mitgeteilt werden, wie diese auf den Speicherbereich zugreifen können (lesend/schreibend). Wurde all dies erledigt, kann der Datenspeicherbereich wie ein gewöhnlicher Speicherbereich verwendet werden.
Leider wurde mit dem Shared Memory IPC keine explizite Synchronisation zur Verfügung gestellt, weshalb man diese Kontrolle selbst noch mit z.B. Sperren oder Semaphore herstellen muss.

3. IPC: Synchronisation

3.1 Probleme

Die Synchronisation bringt die Aktivitäten verschiedener nebenläufigen Prozesse in eine gewünschte Reihenfolge. Sie ist notwendig in den Fällen 1, 2 und 4 (aus dem Punkt 1), und für die Vermeidung von Deadlocks und Livelocks. Deadlock (Verklemmung) ist ein Zustand, in dem die beteiligten Prozesse wechselseitig auf den Eintritt von Bedingungen warten, die nur durch andere Prozesse in dieser Gruppe selbst hergestellt werden können.



Lifelock ist ein deadlock-ähnliches Zustand, in dem die beteiligten Prozesse zwar nicht blockieren, sie aber auch keine wirklichen Fortschritte in der weiteren Programmausführung erreichen. Dieser Zustand ist nicht eindeutig erkennbar.

3.2 Semaphore

Semaphoren sind Zugriffsvariablen, auf die nur mit bestimmten Funktionen zugegriffen werden kann. Eine Semaphore kann mehrere Werte annehmen kann aber auch binär sein. Mit den Prozessen können Sie eine Semaphore abfragen und überprüfen, ob diese einen bestimmten Wert hat. Es ist auch möglich, dass mit einer Funktion auf mehrere Semaphore gleichzeitig zugegriffen wird. Gewöhnlich verwendet man Semaphoren zur Synchronisation beim Zugriff auf gemeinsam benutzte Betriebsmittel/Datenstrukturen. Wenn z.B. der gemeinsamer Speicher (Shared Memory) von zwei oder mehreren Prozessen gleichzeitig verwendet wird, muss verhindert werden, dass diese gleichzeitig schreiben, oder dass ein Prozess liest, während ein anderer zur gleichen Zeit schreibt.

Das Prinzip ist recht einfach. Will man mit einem Semaphor einen bestimmten Abschnitt schützen, wird ein vorhandener Zähler getestet, ob der Wert größer als

0 ist und anschließen dekrementiert. Ist der Wert nicht größer als 0, wird der Prozess in einen Wartezustand versetzt, da sich scheinbar ein anderer Prozess gerade in diesem kritischen Codeausschnitt befindet. Wenn ein Prozess einen kritischen Codebereich verlassen will, muss dieser wiederum den Zähler inkrementieren, damit der kritische Bereich für andere Prozesse wieder zur Verfügung steht.

3.3 Lock Files (Sperrdateien)

Eine recht primitive Form der IPC sind so genannte Lock Files (Sperrdateien), nicht zu verwechseln mit den Dateisperren (Record Locking). Dabei werden mehrere Prozesse mithilfe einer einfachen (Sperr-)Datei synchronisiert. Es wird praktisch eine Datei (meist im Verzeichnis /tmp) angelegt, worauf ein Prozess nur Schreibrechte hat. Die Synchronisation erfolgt jetzt durch einen anderen Prozess, der ebenfalls versucht, dieselbe Datei mit Schreibrechten anzulegen. Schlägt dieser Versuch fehl, existiert gerade eine entsprechende Datei. Dieses Fehlschlagen stellt praktisch die Sperre für den Prozess da. Jetzt wartet der abgewiesene Prozess eine gewisse Zeit (meistens mit einem simplen sleep-Aufruf), bevor dieser erneut versucht, eine Datei mit entsprechenden Namen und mit Schreibrechten anzulegen. Die Freigabe dieser Sperre erfolgt dann durch den entsprechenden Prozess, der diese Datei erzeugt hat, über das Freigeben der Sperrdatei mit unlink().

So toll sich dies in der Theorie anhören mag, in der Praxis ist diese Form der IPC nur bedingt tauglich. Zum einen ist das Problem, sobald der Superuser hier mitspielen will, funktioniert die Synchronisation nicht mehr, da dieser immer schreiben darf. Und zum anderen wird hier mit dem aktiven Warten durch das verschwenden von Prozessorzyklen (mit bspw. sleep()) nicht garantiert, welcher Prozess als nächstes Zugriff auf die Sperrdatei hat. So kann es passieren, dass ein Prozess eventuell nie zum Zuge kommt.

4. Zusammenfassung

IPC ist Transfer von Daten zwischen Prozessen und die Synchronisation dieses Transfers
Der Transfer erfolgt über gemeinsame Speicherbereiche(Variablen) oder über einen gemeinsamen Kommunikationskanal
Die Kommunikation kann synchron oder asynchron verlaufen.
Pipe – unidirektional, verwandte Prozesse, asynchron
Fifo - unidirektional, fremde Prozesse, asynchron
Synchronisation bringt die Aktivitäten der verschiedenen Prozesse in eine bestimmte Reihenfolge
Semaphore – Variablen, die Zugriff auf die gemeinsam benutzte Betriebsmittel synchronisieren
Shared Memory – ein gemeinsamen Speicherbereich, Synchronisation ist notwendig
Sperrdateien – einfaches Synchronisationsmittel, viele Nachteile
Livelock ist schlimmer als Deadlock, da Livelock nicht eindeutig erkennbar ist

5. Quellen

1. Jürgen Wolf, „Linux-Unix-Programmierung“ , 1. Auflage, Galileo Computing, 2004
2. Andrew S. Tannenbaum, „Moderne Betriebsysteme“, 2., überarbeitete Auflage, PEARSON Studium, 2002
3. Softwaresysteme 1, Wolfgang Schröder-Preikschat, 2004, Folien zur Vorlesung