Diplomarbeit

Verallgemeinerung von mmap() am Beispiel des Memory Mappings von Pipes in Linux

ausgeführt am

Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter der Anleitung von

o.Univ.Prof. Manfred **Brockhaus**
und
Univ. Ass. Anton M. **Ertl**
als verantwortlich mitwirkendem Universitätsassitenten

durch

Peter **Syrowatka**

Schlossalle 28, 2512 Tribuswinkel

17. Mai 2000 (Peter **Syrowatka**)

Kurzfassung

Memory Mapping (API-Funktion `mmap()`) bietet eine Möglichkeit auf Dateien zuzugreifen. Dabei entfallen Kopieroperationen, die bei Verwendung der normalen I/O-Funktionen `read()` `write()` intern durchgeführt werden. Diese Methode wird von den Anwendungsprogrammierern aber nicht sehr oft verwendet, da sie nicht auf Stream-Objekte wie Pipes, Sockets oder Terminals angewendet werden kann. Die Standard-I/O Methoden stellen hier einen Weg zur konsistenten Handhabung der unterschiedlichen Objekte zur Verfügung.
Um diesen Nachteil zu eliminieren, stelle ich ein Modell für einen erweiterten `mmap()`-Aufruf vor. Mit dieser Erweiterung ist es möglich, auch Stream-Objekte in den Adressraum einzublenden. Dadurch kann auch bei Verwendung der alternativen, mmap()-bedingten Programmiermethode eine konsistente Struktur der Programme realisiert und durch die Vermeidung der Kopieroperationen eine Effizienzsteigerung erreicht werden.
Die Eigenschaften des Speicherobjektes, welches durch das Einblenden eines Stream-Objektes entsteht, orientieren sich am Verhalten einer eingeblendeten Datei, behält aber einige Eigenheiten des Ursprungsobjektes. So ist zum Beispiel die Größe einer Pipe oder eines Sockets per Definition unbestimmt. Diese Eigenschaft bleibt auf jeden Fall auch beim Einblenden erhalten. Abhilfe bietet die Nutzung der großen virtuellen Adressräume, die auf aktuellen Architekturen zur Verfügung stehen.
Am Beispiel des Memory Mappings für Pipes spezifiziere ich das Modell des erweiterten `mmap()` im Detail und unterziehe es einer Prüfung und demonstriere es anhand einer beispielhaften Implementierung für Linux.
Das erweiterte Memory Mapping stellt einige Anforderungen an das Virtual Memory Management, vor allem den Übergang der Verwaltung von *Page-Granularität* auf *Byte-Granularität*.

Als Demonstrations- und Benchmark-Programme habe ich einfache Anwendungen (`cat`, `wc` und `grep`) entsprechend angepasst und zwei Messreihen durchgeführt. Es erfolgte ein Vergleich zwischen den Applikationen auf Standard-I/O- und `mmap()`-Basis. Dabei schneiden die Programme mit Memory Mapping erwartungsgemäß besser ab (Basissysteme: FreeBSD, SCO, Sun Solaris und Linux). Der Gewinn betrifft vor allem die Systemzeit. Die erhaltenen Messwerte für die Standard-I/O-Methode sind mindestens doppelt so hoch wie bei Verwendung von `mmap()`.

In der zweiten Messreihe wurden die Laufzeiten von Programmen, die über eine Pipe untereinander Daten austauschen, ermittelt. Der Zugriff erfolgt mit `read()/write()` oder mit `mmap()`. Das Ergebnis dieser Messungen entspricht leider nicht ganz den Erwartungen. Aufgrund der nicht optimierten, beispielhaften Implementierung kommt es zu einer leichten Erhöhung der Laufzeit (Gesamt-Zeit steigt ca. um Faktor 1,1 im `mmap()`-Fall).

Abstract

Memory mapping (API function `mmap()`) is a alternative way of accessing files. Compared to the IO functions `read()` and `write()` it avoids unnecessary copy operations. The method is not widely used, though, because its standard implementations cannot be applied to stream objects like pipes, sockets or (pseudo-)terminal devices. This consistent applicability of the standard IO functions to all kinds of objects makes them more attractive.

To eliminate this disadvantage I present the model of an extended `mmap()` call, that allows for mapping arbitrary stream objects into a program's address space. A program using this extended call can be written adhering to only one paradigm of interacting with external objects, the one of `mmap()`, and still avoid unnecessary copying.

The properties of the memory object that is created by mapping a stream object, are modeled after the behavior of a mapped file, though they still keep some properties of the original object, e.g. the size of a pipe or a socket is by definition not known. Although `mmap()` cannot change this, the large virtual address spaces of current architectures make the model very useful.

Using the example of memory mapped pipes I specify the model of extended `mmap()` in detail, benchmark it and demonstrate a sample implementation for Linux.

Extended memory mapping, especially the shift from page granularity to byte granularity, requires some refinements in virtual memory management.

I have adapted some simple commands (`cat`, `wc` and `grep`) and compared their performance to the unmodified versions implemented on basis of standard IO. As expected, the versions using `mmap()` of ordinary files performed better. The difference being mostly in system time used. For the standard IO implementations the measurements required more than twice as much time for `mmap()`.

A second experiment measured the run-times of programs communicating via a pipe. Access methods were either `read()/write()` or `mmap()`. Here we see that the result was not as expected. The slight increase of by factor of 1.1 for `mmap()` can be attributed to the sample implementation not being optimized.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Der herkömmliche Weg Dateien zu bearbeiten folgt einem Schema, welches im Wesentlichen auf den System-Aufrufen `open()`, `read()`, `write()` und `close()` aufbaut. Dazu kommt noch die Funktion `lseek()`, mit der eine Positionierung der Ein-/Ausgabe-Position in der geöffneten Datei erfolgen kann. Wird durch die Anwendung von `read()/write()` die I/O-Position kontinuierlich weitergeschoben, so kann mit `lseek()` diese Position frei gesetzt werden - ein *wahlfreier* Zugriff auf den Inhalt ist möglich.

Die Ein-/Ausgabe von Daten in der Datei erfolgt über die Funktionen `read()/write()` in Blöcken beliebiger Größe. Ein Aufruf einer dieser Funktionen verursacht im Prinzip einen Kopiervorgang. Die Daten werden vom Puffer des I/O-Systems in einen Puffer des Anwendungsprozesses übertragen. Unter Unix wird diese Methode nicht nur für die File-I/O verwendet, sondern dient als Standard-Modell für jede Art der Ein-/Ausgabe. Das geht soweit, daß die Einstiegspunkte zu den Gerätetreibern in Form von *special Files* in Unix-Filesystem repräsentiert sind (Verzeichnis `/dev`). Dadurch kann praktisch auf jede Ressource nach dem Standard-I/O Schema zugegriffen werden ([Bac87], [LMKQ88], [MJLF84]).

Diese Vereinheitlichung ist natürlich sehr von Vorteil für den Anwendungsprogrammierer, da er kaum spezielle Fallunterscheidungen entsprechend unterschiedlicher Datenquellen vornehmen muß.
Intern bedient man sich bei Unix-Systemen eines *Buffer-Caches* über den die File-I/O-Operationen ablaufen. Dadurch wird verhindert, daß das Anwendungsprogramm auf die relativ langsamen externen Speichermedien warten muß. Wird ein Block aus einer Datei eingelesen, so bleibt eine Kopie im Cache wodurch bei einem weiteren Zugriff sofort auf diese Kopie im Hauptspeicher zurückgegriffen werden kann. Bei einer Schreiboperation wird der Block vorerst nur in den Cache im Hauptspeicher übertragen und das Anwendungsprogramm kann bereits fortgesetzt werden. Die tatsächliche Übertragung der Daten auf das externe Medium erfolgt asynchron zu einem späteren Zeitpunkt.

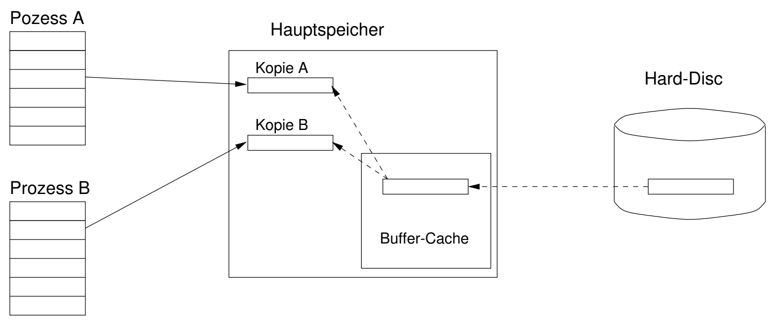

Abbildung 1: File-Zugriff mit traditioneller Unix-I/O

Diese Methode hat allerdings den Nachteil, daß ein und derselbe Inhalt mehrmals im Hauptspeicher gehalten wird. Neben dem, im Prinzip unnötigen Verbrauch von Speicherplatz, ist es vorallem der durch die Kopieroperation entstehende Aufwand, der sich nachteilig auf das Laufzeitverhalten von Programmen auswirkt (Abb.: 1).

Bei der alternativen Methode der File-I/O mit dem System-Call `mmap()` wird gerade dieser Kopieraufwand vermieden. Mit Hilfe dieser Funktion wird ein Bereich im virtuellen Adressraum des Prozesses mit dem Inhalt einer Datei assoziert. Die Datei kann danach in direkter Form durch einfache Speicherreferenzierung angesprochen werden. Die Einblendung kann nur einen Teil oder aber auch die vollständige Datei umfassen. Damit ist ein wahlfreier Zugriff wesentlich leichter möglich, als bei Verwendung der Standard-File-I/O. Wird die Datei sequentiell berarbeitet, ist bei Anwendung der Standard-I/O ein iteratives Einlesen von einzelnen Dateiblöcken in einen internen Puffer nötig. Wird Memory Mapping verwendet, kann die Datei als Ganzes erfasst werden. Das ermöglicht in vielen Fällen wesentlich einfachere Strukturen im Anwendungsprogramm.

Alternativ wird heute oft eine Datei zuerst vollständig in den Speicher eingelesen (*slurp*). Danach kann mit den gleichen Methoden wie bei einer eingeblendeten Datei gearbeitet werden. Allerdings kommen zu den Kosten für die Kopieroperation noch zusätzlichen die Kosten des tatsächlichen, hohen Speicherbedarfs, da beim Einlesen der Datei sofort der entsprechende physische Speicherplatz benötigt wird.

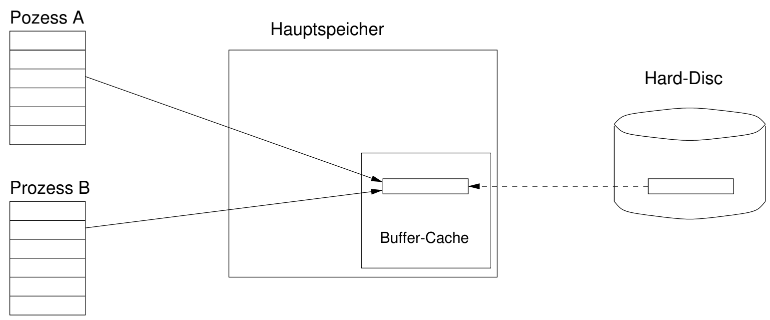

Abbildung 2: File-Zugriff mit File-Mapping

Beim Memory Mapping hingegen werden interne Strategien des Virtual Memory Managements ausgenutzt. Der prinzipielle Ablauf erfolgt dabei über das *Exception Handling*. Dabei kommt das Prinzip des *Demand Pagings* zum Einsatz. Beim Einblenden der Datei, werden vorerst lediglich die Verwaltungsstrukturen entsprechend eingerichtet. Den virtuellen Adressen wird der Inhalt erst dann zugewiesen (der virtuellen Page wird ein Page-Frame zugeordnet), wenn das erste Mal auf sie zugegriffen wird. Dabei kommt es zu einer *Page-Exception* und der Exception Handler fordert vom I/O-Subsystem den zugeordneten Block des externen Mediums an. Befindet sich der gesuchte Inhalt bereits im Cache, braucht er nicht extra kopiert zu werden, sondern es kann sofort der Page-Frame weiterverwendet werden. Er wird also zwischen verschiedenen Funktionen geteilt (*Page Sharing*, Abb.: 2). Zu der Ersparnis an Rechenzeit für das Kopieren kommt auch noch die optimale Nutzung des vorhandenen Speichers. Soll eine Datei exklusiv (private) eingeblendet werden, kann der Inhalt trotzdem vorerst gemeinsam genutzt werden. Erst beim ersten schreibenden Zugriff darauf braucht eine private Kopie erzeugt werden (*Copy on Write, CoW*).

All diesen Vorteilen steht aber der entscheidende Nachteil gegenüber, daß diese Methode nur auf reguläre Dateien anwendbar ist. In sehr vielen Fällen repräsentieren die zu behandelnden Filedeskriptoren aber characterorientierte Objekte wie Pipes, Sockets oder Terminals. In diesen Fällen hat die Standard-I/O-Methode einfach den Vorteil, daß man sich darum nicht zu kümmern braucht. `read()/write()` können hier wie gewohnt eingesetzt werden. Soll eine Anwendung mit beiden Datenquellen arbeiten können, würde die Integration der `mmap()`-Methode praktisch den doppelten Programmieraufwand verursachen. Betrachtet man zum Beispiel die große Klasse der, unter Unix üblichen Filterfunktionen zur Dateimanipulation (`wc, grep, sed, awk, cat, tr,` ...), wird das Problem leicht sichtbar. Allen diesen Kommandos kann eine Datei als Parameter übergeben werden. In diesem Fall könnte die Bearbeitung unter Verwendung von `mmap()` erfolgen. Viel häufiger aber werden solche Kommandos in Filterketten auf einen Datenstrom angewendet, der mit Hilfe von *Pipes* aufgebaut wird. Jedem mit Unix vertrauten Anwender sind Kommandozeilen ein Begriff, wo mit Hilfe von `'|'` die Ein- und Ausgabekanäle der einzelnen Programme miteinander verbunden werden. Realisiert wird das, indem durch die *Shell* Pipes erzeugt werden, die als Standardein- und Standardausgabe für die Kommandos eingerichtet werden.

Auf die Pipes kann aber `mmap()` nicht angewendet werden. Der Programmierer müsste also eine Fallunterscheidung im Programm vornehmen und seine Applikation sozusagen doppelt implementieren - einmal für die Anwendung von `mmap()` bei regulären Dateien und einmal mit den Standard-I/O Mitteln für die Anwendung auf FIFOs.

Bei den unter Unix zur Kommunikation zwischen Programmen auf Rechnern in Netzwerken verwendeten *Sockets* ist die Situation überhaupt so, daß der Zugriff ausschließlich nur mit den Standard-I/O-Methoden möglich ist.

Es ist einzusehen, daß daher die Anwendung von `mmap()` nicht sonderlich lukrativ erscheint und man sich auf die Mittel der Standard-I/O beschränkt.

Ein modifizierter `mmap()`-Aufruf, der auf alle Objekte anwendbar ist, würde sicherlich dazu beitragen, die Akzeptanz dieser Programmiermethode bei den Anwendungsprogrammierern zu erhöhen. In der vorliegenden Arbeit, wird ein Modell für die Erweiterung der `mmap()`-Funktion auf charakterorientierte Objekte vorgestellt. Anhand von Pipes wird das Modell genau spezifiziert.

1.2 Virtueller Speicher

Die Methode des *Memory Mappings* als solche, erfordert ein spezielles Modell der Speicherorganisation. Es ist unter dem Begriff der *virtuellen Speicherverwaltung* bei allen modernen Betriebsystemen in Verwendung ([Sta95], [Den97], [Dor76], [Dei90], [RYT$^+$87], [Tan87]).

Um den vorhandenen, begrenzten physischen Speicher zu erweitern, wird ein gewisser Teil des sekundären Speichers herangezogen, um temporär Inhalte aus den Hauptspeicher auszulagern. Die Programme sollen sich natürlich um solche Dinge nicht kümmern müssen. Diese Vorgänge sollen für sie transparent ablaufen. Dem Programm wird ein *kontinuierlicher, virtueller Adressraum* zur Verfügung gestellt (*künstliche Kontinuität*). Hinter dem Modell des *virtuellen Speichers* werden alle Details der Organisation und Verwaltung der tatsächlichen Speicherressourcen verborgen. Die Arbeit selbst wird von der *Memory Management*-Komponente des Betriebssystems übernommen.

Eine elementare Aufgabe der virtuellen Speicherverwaltung ist die Umsetzung der virtuellen Adressen in physische Adressen. Allerdings sind für sie folgende Umstände zu berücksichtigen:

- Diese Adresstransformation muß für jede einzelne (logische) Adresse in dem Augenblick erfolgen, wo auf sie zugegriffen wird (Dynamic Address Translation). Es ist daher von elementarer Bedeutung, daß diese Abbildung schnell erfolgt, da ansonsten die Effizienz des Computersystems auf ein nicht tolerierbares Maß absinken würde. Um diesen Leistungsgrad zu erreichen wird auf jeden Fall eine Unterstützung durch die Hardware benötigt (**MMU**, Memory Management Unit).

- Für die Zuordnung von logischer zu realer Adresse muß vom Betriebssystem Information gespeichert werden. Erfolgt die Abbildung für jede Adresse einzeln, so ist es leicht einzusehen, daß für diese Verwaltungsinformation sehr viel Speicherplatz gebraucht wird, mitunter mehr als tatsächlich vorhanden ist. Der Ausweg liegt in der Blockung der Adressräume. Dadurch braucht pro Block mit n Elementen nur 1 Informationseinheit für die Zuordnung abgelegt werden.

In der Praxis haben sich drei Organisationsmodelle für die virtuelle Speicherverwaltung durchgesetzt:

- Paging
- Segmentierung
- Kombination Paging/Segmentierung

1.2.1 Paging

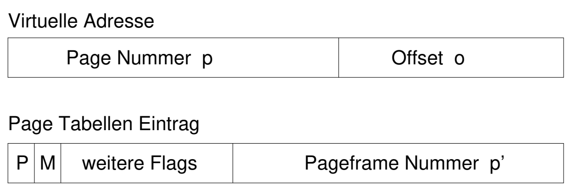

Abbildung 3: Virtuelle Adresse und Page Tabellen Eintrag

Beim *Paging* wird der virtuelle Adressraum in Blöcke gleicher Größe, den *Pages* (p), geteilt. Der physische Speicher wird ebenfalls in solche Blöcke, die

Pageframes (p'), geteilt. Die Zuordnung einer virtuellen Adresse zu einem realen Speicherplatz erfolgt nicht mehr direkt über eine $Adresse \rightarrow Adresse$, sondern über eine $Page \rightarrow Pageframe$ Zuordnung (Abb.: 4).

Für diese Zuordnung wird eine Adressübersetzung benötigt. Diese Aufgabe wird von der MMU ausgeführt. Die in Abb. 3 angegebenen Flags **P** und **M** werden ebenfalls von der MMU benötigt. **P** ist das *present* Flag. Es zeigt, ob der Pagetable-Entry auf einen Pageframe verweist. Ist es nicht gesetzt, löst die MMU beim Zugriff auf eine Adresse in dieser Seite eine *Page Exception* aus, wodurch die Kontrolle an das Betriebssystem übergeben wird und die Speicherverwaltung für das Einlesen des geforderten Inhaltes sorgen kann.

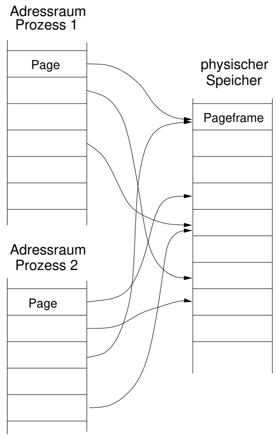

Abbildung 4: Page - Pageframe Zuordnung beim Paging

Dazu wird allerdings ein freier Pageframe benötigt. Sind alle Frames bereits belegt, so kann Speicherplatz auf einem Massenspeichermedium für die zwischenzeitliche Auslagerung von Speicherseiten verwendet werden. Diese Methode wird als *Demand Paging* bezeichnet.

Soll eine Seite ausgelagert werden, so wird das **M**-Flag (*modified*) dazu verwendet, anzuzeigen, ob der Inhalt des betreffenden Frames verändert wurde, oder nicht. Damit wird der Speicherverwaltung signalisiert, daß die Seite ausgelagert werden muß. Ist es nicht gesetzt, kann der Inhalt der Seite verworfen werden, da er jederzeit wieder aus der ursprünglichen Quelle (zB. Programmdatei) neu eingelesen werden kann. Für das *Demand Paging* ist noch ein weiteres Flag zu verwenden. Damit wird angezeigt daß sich der benötigte Inhalt im Auslagerungsbereich befindet. Das *present* Flag alleine zeigt ja nur an, ob die Seite im Speicher ist oder nicht. Von wo der Inhalt eingelesen werden soll, läßt sich mit dieser Information alleine nicht feststellen.

Der hier beschriebene Abbildungsvorgang wird als *Direct Mapping* bezeichnet. Das Problem dabei ist, daß die Pagetabelle im Hauptspeicher liegt. Dadurch sind pro Adressreferenz mindestens zwei Zugriffe auf den Hauptspeicher notwendig (ein zusätzlicher um den Pagetabellen Eintrag zu lesen), was insgesamt zu langsam ist. Um den Übersetzungsvorgang zu beschleunigen kommt ein hardwareunterstützter, schneller, assoziativer Cache für eine kleine Anzahl von

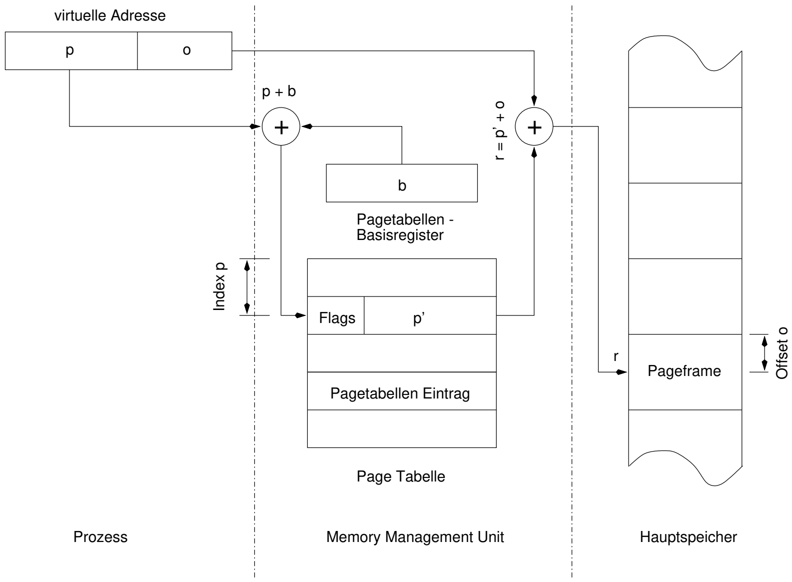

Abbildung 5: Adressübersetzung beim Paging

Pagetabellen-Einträgen zum Einsatz. Durch die Verwendung dieses *Translation Lookaside Buffers* (TLB) läßt sich die Zugriffszeit auf PTEs signifikant verringern ([CE85], [Dei90]). Erst wenn ein Eintrag nicht im TLB gefunden werden kann, wird auf die Pagetabelle in Hauptspeicher zugegriffen. Es liegt also ein kombiniertes direkt/assoziativ Mapping Verfahren vor ([Sta95]).

Der Schutz des Speichers vor unberechtigten Zugriffen ist einfach dadurch realisiert, daß ein Prozess immer nur mit logischen Adressen in seinem virtuellen Adressraum operiert. Er hat keine Möglichkeit direkt auf Pageframes und damit auf physikalische Adressen zuzugreifen. Weiters wird zu den einzelnen Pages eine Kombination von Zugriffsmodi (*read, write, execute*) im Pagetabellen Eintrag (Abb. 3) abgelegt. Damit können die einzelnen Abschnitte des Prozess-Images (*Text, Data, Stack, Heap ...*) entsprechend geschützt werden. Die Kontrolle dieser Modi erfolgt ebenfalls durch die MMU. Wird ein unerlaubter Zugriff versucht, so löst die MMU eine *Exception* aus und übergibt die Kontrolle an eine entsprechende Betriebssystemfunktion.

Ein Pageframe kann nicht nur von einem einzigen Pagetabellen-Eintrag referenziert werden. Es kann mehrere solcher Einträge geben, die auf diesen Frame verweisen. Das können auch Einträge in Pagetabellen sein, die unterschiedlichen Prozessen zugeordnet sind. Damit ist das *Sharing* von Speicherinhalten zwischen mehreren Prozessen möglich. Nachdem aber das Paging transparent für den User-Prozess erfolgt, können die für das Sharing nötigen Schritte nicht auf Anwendungsebene durchgeführt werden. Das Betriebssystem muß entsprechende Systemaufrufe zur Verfügung stellen.

1.2.2 Segmentierung und Kombination Paging/Segmentierung

Die Organisation des Speichers in Form von Segmenten kann als allgemeiner Fall des Pagings gesehen werden. Beim Paging wird mit fixer Blockgröße gearbeitet. Die Segmente können eine beliebige Größe annehmen. Sie kann sogar dynamisch geändert werden. Damit lassen sich dynamische Datenstrukturen sehr gut modellieren.

Die Adressübersetzung erfolgt analog zum Paging. Im Gegensatz zu diesem muß aber in einem Segmenttabellen-Eintrag zusätzlich noch die Größe des Segments verwaltet werden. Insgesamt ist die Verwaltung des Speichers auf Segmentbasis aber aufwendiger, da wieder auf Strategien zur optimalen Speicherbelegung (*Placement*-Algorithmen) zurückgegriffen werden muß, um die externe Fragmentierung des Speichers klein zu halten.

Daher wird, wenn heute noch die Segmentierung des Speichers verwendet wird, die Kombinationsmöglichkeit zwischen Segmentierung und Paging angewendet. Dabei werden die Segmente selbst wieder auf Pages abgebildet und die Speicherverwaltung erfolgt letztlich wieder mit den Paging-Methoden. Der Adressübersetzungsvorgang ist dabei aber komplexer, da er zweistufig über Segment- und Page-Tabellen erfolgt.

1.3 Memory Mapping

Memory Mapping beschreibt nicht nur das Einblenden einer Datei in den Adressraum mit Hilfe der API-Funktion `mmap()`. Der Begriff definiert eine allgemeine Methode des Speichermanagements. Allgemein gesagt besteht *Memory Mapping* darin, ein (Speicher-)Objekt auf einen bestimmten Bereich des virtuellen Adressraumes abzubilden. Entscheidend dabei ist, daß das Objekt durch die Abbildung in den virtuellen Adressraum *direkt adressierbar* ist. Ein Zugriff auf das Objekt erfolgt damit direkt durch einen Speicherzugriff ohne der Zuhilfenahme diverser Subsysteme oder Betriebssystemfunktionen. Was alles als Objekt in diesem Sinne gilt, hängt vom Speichermodell der betrachteten Plattform ab.

Das einfachste Beispiel für Memory Mapping ist die Zuordnung eines Pageframes zu einer virtuellen Page. Der virtuelle Adressraum ist vorerst lediglich ein kontinuierlicher Adressbereich, in dem jede Adresse von einem Prozess oder vom Betriebssystem grundsätzlich angesprochen werden kann. Um aber tatsächlich an einer Adresse etwas ablegen oder von dort etwas einlesen zu können, muß der virtuellen Adresse eine Adresse des physischen Speichers zugeordnet werden. Der physische Speicher wird also an eine Stelle in den virtuellen Adressraum eingeblendet.

Auf Anwendungsebene steht Memory Mapping in Form des System-Calls `mmap()` zur Verfügung. Auf der Betriebssytem-Ebene wird es vielfältiger verwendet. Beispiele der Anwendung in verschiedenen Betriebssystemen sind im Abschnitt 6.1 nachzulesen.

1.3.1 Die API-Funktion `mmap()`

Dem Anwender wird Memory Mapping in Form des System-Calls `mmap()` angeboten. Seinen Ursprung hat der Aufruf in der *System V*-Entwicklungslinie (VM-

Modell von SVR4/SunSolaris, [Vah96]). In der *BSD4.3* Definition war zwar eine Definition des Aufrufes enthalten, es fehlte aber die Implementierung. BSD4.4 ([MBKQ96]) basierte Systeme verfügen bereits alle über einen `mmap()`-Aufruf.

Mit diesem Aufruf können Dateien in den Adressraum des Preozesses eingeblendet werden. Der Inhalt der Datei ist dann direkt über Speicherreferenzierung anzusprechen.

Laut *UNIX98* Spezifikation ([Ope98]) bzw. POSIX ([POS90]) ist der System-Call `mmap()` folgendermaßen festgelegt:

```
#include<sys/mman.h>

void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off)
```

Bei einem Aufruf der Form

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

wird die Datei, die durch den Filedeskriptor `fildes` angegeben ist, in den Adressraum des Prozesses eingeblendet, wobei durch `len` und `off` der betreffende Teil der Datei genauer angegeben werden kann. Der Offset `off` gibt den Anfang des Bereiches innerhalb der Datei an, aus `off` und `len` ergibt sich das Bereichsende. Wird ein Offset angegeben, so muß er auf einer Page-Grenze liegen. Die obere Grenze wird automatisch auf die nächsthöhere Page-Grenze gezogen, wobei der Inhalt zwischen dem berechneten Bereichsende und der oberen Bereichsgrenze undefiniert ist.

Mit `addr` kann die Stelle im virtuellen Adressraum, an dem die Datei eingeblendet werden soll, vorgegeben werden. Wird ein NULL-Pointer für `addr` übergeben, so wird die Anfangsadresse vom Betriebssystem festgelegt. Der Wert von `addr` muß in Kombination mit den Flag MAP_FIXED auch page-aligned sein. Ansonsten wird er automatisch auf eine benachbarte Page-Grenze gelegt.

Ist der Aufruf erfolgreich, so enthält `pa` die Anfangsadresse der Einblendung. Im Fehlerfall liefert `mmap()` den Wert MAP_FAILD, der in `sys/mman.h` definiert ist.

Der Parameter `prot` legt die Zugriffsrechte für den Bereich fest, welche aus den in `sys/mman.h` definierten Konstanten PROT_READ, PROT_WRITE und PROT_EXEC durch bitweises *inklusiv* ODER gebildet werden können. Zusätzlich steht noch die Konstante PROT_NONE zur Verfügung, um jeglichen Zugriff auf den Bereich zu verbieten.

`flag` kann Werte auf Basis der ebenfalls in `sys/mman.h` definierten Konstanten MAP_SHARED, MAP_PRIVATE und MAP_FIXED annehmen. MAP_-SHARED und MAP_PRIVATE legen die Wirkung von Schreibzugriffen des einblendenden Prozesses auf das Objekt fest. Schreibzugriffe bei MAP_SHARED verändern den Inhalt des Objekts. Änderungen an MAP_PRIVATE Objekten sind nur für den Prozess sichtbar. Das Objekt selbst wird nicht verändert.

Mit MAP_FIXED wird festgelegt, daß die Einblendung genau an der Adresse `addr` im virtuellen Adressraum des Prozesses zu erfolgen hat. Dabei werden alle Einblendungen die die Pages im Intervall $[pa, pa + len)$ betreffen, entfernt.

1.3.2 Verwandte Funktionen

Beim Einblenden einer Datei in den virtuellen Adressraum mit `mmap()` wird ein geschlossener Bereich von virtuellen Adressen als zusammengehörig ausgewiesen. Entsprechend den Flags und Protection-Werten die dem Aufruf übergeben wurden, hat jede Page in diesem Bereich die selben Attribute. Unabhängig von der, im jeweiligen Modell verwendeten Terminologie, werden diese Bereich in der Folge als VMAs (*Virtual Memory Area*) bezeichnet.

Diese VMAs unterstehen der Kontrolle durch den Anwender. Um sie manipulieren zu können, stehen einige weitere Funktionen zur Verfügung:

- `munmap()`: Wird eine Einblendung zum Teil oder vollständig nicht mehr benötigt, kann sie mit dieser Funktion wieder ausgeblendet werden.

- `mprotect()`: Ermöglicht es die Zugriffsrechte auf einen VMA zu verändern.

- `msync()`: Diese Funktion stellt das Äquivalent zum `sync()`-Aufruf, mit dem die I/O-Buffer geleert werden können, dar. Mit ihrer Hilfe wird der Speicherinhalt mit dem Inhalt am externen Medium abgeglichen.

- `mlock()`: Um sicherzustellen, daß kritische Inhalte nicht in den Auslagerungsbereich gelangen können, bietet `mlock()` die Möglichkeit die Pages eines VMAs im Hauptspeicher zu fixieren. Zum Aufheben einer solchen Sperre wird `munlock()` verwendet.

- `munlock()`: siehe `mlock()`.

Die Anwendung der angegebenen Funktionen kann auf einen gesamten VMA oder nur auf einen Teil eines solchen erfolgen. Da ein VMA immer einen Bereich mit gleichen Attributen beschreibt, kann es dabei zur Teilung eines VMAs in maximal drei neue Bereiche kommen.

Der Bereich auf den die gewünschte Funktion angewendet wird, wird in Form von zwei Parametern angegeben: die Anfangsadresse und die Länge.
Für diese Werte gilt, wie beim `mmap()`-Aufruf, daß der Bereichs-Anfang auf einer Page-Grenze liegen muß, die Länge aber einen beliebigen Endwert festlegen kann. Dabei stellt sich aber die Frage, welcher Endwert aber dann tatsächlich verwendet wird.
Der Standard definiert das Verhalten in dem Sinn, daß die Anwendung der geforderten Operation auf alle angegebenen Teile garantiert wird. Das bedeutet, daß der obere Grenzwert bei Bedarf auf die nächsthöhere Page-Grenze gezogen wird (*maximierendes Verhalten*).

1.3.3 Programmieren mit `mmap()`

Üblicherweise werden Dateien unter Unix mit der Standard-I/O Mitteln bearbeitet. Dabei wird mit `open()` die Datei geöffnet. Man erhält einen File-

Deskriptor, der bei allen weiteren Operationen die Datei identifiziert. Mit diesem Deskriptor ist eine I/O-Position innerhalb der Datei assoziert. Bei der Durchführung von I/O-Operationen mit den Funktionen `read()/write()` wird diese Position automatisch entsprechend verschoben. Gezielt kann sie mit Hilfe von `lseek()` gesetzt werden. Damit ist ein wahlfreier Zugriff auf den Datei-Inhalt realisierbar. Am Ende der Bearbeitung kann die Datei mit close() wieder geschlossen werden.

Darüberhinaus stehen noch weitere Funktionen für zusätzliche Manipulationen zur Verfügung. Die wichtigsten sind `ftruncate()` um die Größe der Datei zu verändern, `sync()` um einen Abgleich des externen Inhaltes mit dem Buffer-Cache durchzuführen und `stat()` mit der allgemeine Informationen über die Datei angefordert werden. Der Aufruf `ioctl()` stellt Funktionen für alle Aufgaben, die nicht in das Standard-I/O-Schema passen, zur Verfügung.

Bei der Bearbeitung der Datei stehen zwei Varianten zur Auswahl. Der reine sequentielle Zugriff ist auf alle Typen von Filedeskriptoren möglich. Dabei wird die Datei sequentiell mit `read()` ausgelesen bzw. mit `write()` geschrieben. Beim sequentiellen Zugriff kann eine Position nur einmal gelesne oder geschrieben werden.

Werden reguläre Dateien bearbeitet, so ist ein wahlfreier Zugriff auf den Inhalt möglich. Durch `lseek()` kann die I/O-Position an eine beliebige Stelle innerhalb der Datei gesetzt werden, wodurch eine Position auch mehrmals geschrieben oder ausgelesen werden kann.

Die Ein-/Ausgabe erfolgt dabei mit Blöcken unterschiedlicher Größe. Minimal erfolgen die Operationen `read()/write()` auf einzelne Bytes. Wesentlich dabei ist, daß die Daten zwischen einem Puffer im Adressraum und einem Puffer im I/O-Subsystem kopiert werden. Bei gepufferter I/O (`getc()`, `gets()`, usw.) kommt es sogar zu einer weiteren Kopieroperation, da die Library-Funktionen selbst auch noch einen Puffer verwenden.

Allgemein ist der Ablauf so, daß die Daten zuerst mit `read()` aus der Datei in einen lokalen Puffer eingelesen, dort bearbeitet und danach aus diesem Puffer wieder in eine (andere) Datei mit `write()` geschrieben werden. Es wird also eine Programmstruktur entstehen, bei der iterativ eine Folge von Eingabe, Bearbeitung und Ausgabe der Daten abgearbeitet wird. Folgender Pseudo-C-Code veranschaulicht diese Struktur:

```
 .
 .
char buf[BUFSIZE];
int  infd, outdf;
size_t readcnt;
 .
 .
infd = open ("IN-FILENAME",...);
outfd = open ("OUT-FILENAME",...);
 .
 .
while (read(infd,buf,BUFSIZE) != EOF) {
```

```
        .
        /* Manipulationen der Input-Daten ... */
        .
        write(outfd,buf,BUFSIZE);
    }
     .
     .
    close(infd);
    close(outfd);
     .
     .
```

Handelt es sich um Dateien, die in Relation zur Speichergröße nicht allzu groß sind, ist es mitunter sinnvoll, diese zuerst vollständig in einen lokalen Buffer einzulesen (*Slurping*), um dann die Bearbeitung wahlfrei auf die ganze Datei durchführen zu können. Dabei wird die Situation nachgebildet, die man vorfindet, wenn man die Datei mit `mmap()` in den Adressraum einblendet.

Soll die Bearbeitung der Datei mit `mmap()` erfolgen, muß sie ebenfalls zuerst geöffnet werden. Der `mmap()`-Aufruf benötigt den, von `open()` gelieferten File-Deskriptor. Nun kann die Datei zur Gänze oder nur ein Teil davon in den Adressraum eingeblendet werden. Danach kann der File-Deskriptor bereits mit `close()` geschlossen werden. Nach der Bearbeitung der Daten wird der Bereich mit `munmap()` wieder aus dem Adressraum ausgeblendet.

Wie im folgenden Programmsegment zu sehen ist, erspart man sich bei der Verwendung von `mmap()` eine Schleife.

```
     .
    char *inbuf, *outbuf;
    int  infd, outdf, i;
    size_t isize;
     .
     .
    infd = open ("IN-FILENAME",...);
    outfd = open ("OUT-FILENAME",...);
     .
    isize = ... /* Dateigröße über fstat() oder lseek() */
    ftruncate (outfd, OSIZE); /* Dateigröße über ftruncate() setzen */
     .
    inbuf = mmap (NULL, isize, PROT_READ, MAP_PRIVATE, infd, 0);
    outbuf = mmap (NULL, OSIZE, PROT_WRITE, MAP_PRIVATE, outfd, 0);
    close(infd);
    close(outfd);
     .
        /* Berechnung von Output-Daten aus den Input-Daten mit */
        outbuf[i] = foo(inbuf[i]);/* direkter Adressierung ... */
        .
     .
    munmap(inbuf);
```

```
munmap(outbuf);
  .
```

Der Zugriff auf den Inhalt der Datei ist wahlfrei durch Adressreferenzierung im zugeordneten Adressbereich möglich. Der Inhalt der Datei wurde beim `mmap()` noch nicht in den Speicher geladen. Das geschiet erst wenn tatsächlich darauf zugegriffen wird. Eine Einblendung verbraucht zum Zeitpunkt an dem sie angelegt wird noch keinen Speicher. Daher ist es auch möglich Dateien, die größer sind als der vorhandene (virtuelle) Speicher, einzublenden. Es wirden die Möglichkeiten des *Demand Pagings* der Speicherverwaltung genutzt.

Das ist auch der Unterschied zum `Slurping`. Beim vollständigem Einlesen der Datei wird der ganze Speicher sofort benötigt. Ein Slurping von Dateien, die größer sind als der physische Speicher, ist nicht zu empfehlen, da man damit das Pagingsystem einer Hochlastsituation aussetzt.

War die Datei oder der Teil davon, der eingeblendet wurde kleiner als das ganzzahllige Vielfache der Page-Größe, so ist der Inhalt, der zwischen der Bereichs-Obergrenze und der nächsthöheren Page-Grenze liegt, undefiniert. Zugriffe auf Adressen ausserhalb des Einblendungsbereiches werden natürlich genauso behandelt, wie alle anderen Zugriffe auf ungültige Adressen (Segmentation Fault).

Wird die Größe einer eingeblendeten Datei verändert, so ist das in der Einblendung nicht direkt sichtbar. Ein Vergrößerung ist kein Problem. Wird sie allerdings verkleinert, sodaß dadurch Teile der Einblendung hinter dem neuen Dateiende liegen, so ist das Ergebnis eines Zugriffs auf eine Adresse in diesem Bereich undefiniert. Dieser Umstand sollte beim Arbeiten mit eingeblendeten Dateien immer beachtet werden. Der Bereich der durch die Größenänderung betroffen ist, kann durch ein `munmap()` aus dem Adressraum ausgeblendet werden.

Soll eine Vergrößerung der Datei auch in der Einblendung nachvollzogen werden, kann das nur erreicht werden, indem man sie zuerst mit `munmap()` ausblendet und danach mit dem erhöhten Wert wieder einblendet.

Die Größenänderung an der Datei selbst kann mit Hilfe von `ftruncate()` ausgeführt werden. Genauso werden auch neu zu erzeugende Ausgabedateien behandelt. Mit einem `open()` mit *create*-Mode wird ein File-Deskriptor erzeugt, worauf mit `ftruncate()` die Größe der neuen Datei festgelegt wird. Es ist nicht genug, beim `mmap()`-Aufruf lediglich einen beliebigen Längenwert für die Einblendung anzugeben. Dieser darf maximal das Dateiende erreichen. Daher muß zuvor die Größe entsprechend gesetzt werden.

Die Verwendung von `mmap()` zur Realisierung der File-I/O hat also zwei Konsequenzen. Einerseits kann man durch die andere Zugriffsart eine Effizienzsteigerung gegenüber der Standard-I/O-Methode erreichen. Andererseits führt die Anwendung von `mmap()` zu anderen, mitunter einfacheren Programmstrukturen. Im Kapitel 5 sind diese Konsequenzen an praktischen Beispielen nachvollziehbar.

Ein Aspekt der `mmap()`-Methode wurde noch nicht angesprochen. Dateien können als einfachste Form der Interprozess-Kommunikation (IPC) verwendet werden. Dabei ist aber auch der Aufwand der Standard-I/O-Funktionen in kauf zu nehmen. Wird die Datei, über die mehrere Prozesse miteinander kommunizieren wollen von diesen mit MAP_SHARED eingeblendet, so kann auch in diesem Fall der Aufwand reduziert werden. Die IPC mit den Standard-I/O-Mitteln bietet aber bereits eine gewisse Form der Synchronisierung, da die `read()/write()`-Operationen bis zu einer bestimmten Blockgrösse *atomar* ausgeführt werden.

1.3.4 Systemspezifische Eigenschaften

Der in den Standards beschriebene Funktionsumfang stellt eine Mindestanforderung dar. Reale Implementierungen können erweiterte Möglichkeiten anbieten. Unter vielen Systemen wird zusätzlich das Flag MAP_ANONYMOUS angeboten. Bei einem solchen `mmap()`-Aufruf wird kein Filedeskriptor angegeben, da ja keine Datei eingeblendet werden soll. Es wird ein *anonymer Speicherbereich* eingerichtet. Natürlich ist das eine Leistung, die auch von `malloc()` erbracht werden kann. Dieses bietet auch noch die effiziente Verwaltung von Speicherbereichen, die kleiner als die Pagesize sind, an. Versucht man mit `mmap(,,MAP_ANONYMOUS,)` Speicherbereiche kleiner als Pagesize einzurichten, wird immer eine Page dafür verbraucht.

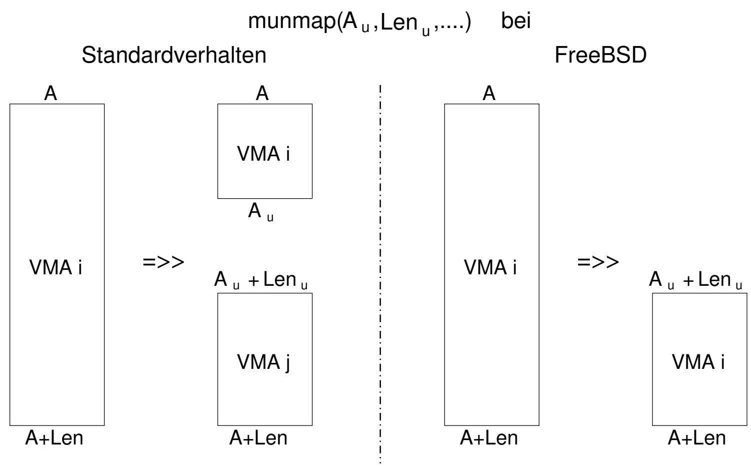

Abbildung 6: Standard-`munmap()` versus FreeBSD

Einen wirklichen Sinn macht ein anonymer Speicherbereich dann, wenn er mit MAP_SHARED erzeugt wurde. Entsprechend der Definition von `fork()` erbt ein Sohnprozess, der durch `fork()` erzeugt wird, auch alle Mappings des Vaters. Daher würde dieser auch die *anonymen, shared Mappings* des Vaters übernehmen. Eine elegante Lösung um *Shared Memory* zwischen verwandten Prozessen zu erzeugen.

Leider ist diese Methode nicht portabel, da sie nicht auf allen Systemen zur Verfügung steht. So erlaubt *Linux* zum Beispiel MAP_ANONYMOUS nur mit MAP_PRIVATE.

Manche Systeme, wie zum Beispiel Linux, bieten einen `mremap()`-Call an, der die Funktion der `munmap() - mmap()`-Kombination erfüllt. Dieser Aufruf ist aber nicht im Standard enthalten und sollte daher für portable Programme nicht verwendet werden.

Unter AIX ist die Anzahl der Einblendungen beschränkt, da sie durch Segmente dargestellt werden. Gemeinsam mit der Anzahl der Shared Memorys, die ebenfalls eigene Segmente belegen, können nur maximal 8 solcher Objekte angelegt werden. Die weitern 8 der 16 vorhandenen Segmente werden durch andere Teile des Prozess-Images belegt ([Kel96]).

Besonders unangenehm ist eine Eigenschaft von FreeBSD. Bei Operationen auf virtuellen Speicherbereichen werden Angaben für den Bereichsanfang, die einen Offset in einem VMA darstellen, einfach ignoriert. Die Operation wird vom Anfang des VMAs beginnend, angewendet. In Abb.: 6 ist dieser Sachverhalt dargestellt. Solche Abweichungen können sich natürlich fatal beim Versuch, Programme auf diese Plattform zu portieren, auswirken.

2 Erweitertes `mmap()`

Die Stärke der Standard-I/O-Methode liegt in der Normierung jeglicher I/O-Vorgänge auf dieses Konzept. Unabhängig von der Art des I/O-Objektes kann mit `read()/write()` darauf zugegriffen werden. Diesen Status auch für die Anwendbarkeit von `mmap()` zu erreichen ist sehr aufwendig aber nicht nötig um die Konkurrenzfähigkeit der Memory-Maping Methode auf ein akzeptables Maß zu heben.

Die Zielgruppe der Anwendungen, wo Memory Mapping sinnvoll verwendet werden könnte, beschränkt sich auf jene, wo Dateien oder dazu äquivalente Objekte manipuliert werden. Die dateiäquivalenten Objekte sind jene, auf welche die üblichen Applikationen zur Dateibearbeitung auch angewendet werden.
Das sind vorallem *Pipes, FIFOs, Sockets* und *Terminals*. Diese Objekte haben eines gemeinsam: sie sind alle character-orientiert. Auf sie ist `lseek()` nicht anwendbar, da sie keinen wahlfreien Zugriff erlauben, sondern einen sequentiellen Datenstrom darstellen. Für einen Zugriff mit `read()/write()` stellt das kein Problem dar, aber mit `mmap()` lassen sie sich nicht erfassen.

Die Zielsetzung besteht jetzt darin, ein Modell zu definieren, welches einen gangbaren Weg für das Memory Mapping dieser Stream-Objekte darstellt. Dabei soll möglichst geringe Änderungen oder Erweiterungen am bestehenden `mmap()`-API vorgenommen werden.

2.1 Einblendbares Stream-Objekt

Betrachtet man den `mmap()`-Aufruf, so können daraus folgende Forderungen an das einzublendende Objekt gestellt werden:

- Es muß durch einen File-Deskriptor beschreibbar sein.
- Es muß möglich sein, einen Offset innerhalb des Objektes festzulegen.
- Es muß möglich sein, dem Objekt eine Länge zuzuordnen.
- Unterschiedliche Zugriffsrechte auf das Objekt müssen definierbar sein.
- Es soll shareable sein.

Diesen Forderungen stehen die typischen Eigenschaften der betroffenen Stream-Objekte gegenüber:

- Geöffnete Stream-Objekte werden durch File-Deskriptoren beschrieben.
- Unterschiedliche Eigenschaften aus Lese- und Schreibperspektive.
- FIFO-Verhalten und daher kein wahlfreier Zugriff; etwas das hineingeschrieben wurde, kann nicht mehr verändert werden; ausgelesene Daten sind nicht mehr enthalten.
- Keine definierte Größe des Objektes. Sicher kann nur die Aussage gemacht werden, ob etwas zum Lesen enthalten ist oder nicht. Es kann beliebig lange hineingeschrieben werden.

- Es kann kein Offset im Objekt angegeben werden.
- Zugriffsrechte können wie bei Dateien angegeben werden.
- Mehrere Prozesse können zugleich darauf zugreifen.

Beim Vergleich der beiden Listen ergeben sich überraschend viele Übereinstimmungen. Lediglich die Forderung nach einer definierten Größe und im Zusammenhang damit, die Möglichkeit einen Offset anzugeben, wird nicht erfüllt.

2.1.1 Das Größenproblem

Die Angabe des Längenwertes beim `mmap()` Aufruf verlangt nicht die tatsächliche Größe des Objektes, sondern nur den Wert, der erfasst werden soll. Man kann einen Wert angeben, der unter oder über der tatsächlichen Größe liegt. Bestimmt wird dadurch vorerst nur das Ausmaß des angelegten VMAs. Ob zu einer virtuellen Adresse in diesem Bereich bei einem Zugriff tatsächlich ein Inhalt zugeordnet werden kann, ist zu diesem Zeitpunkt nicht relevant.
Ist die Größe der zu bearbeitende Datenmenge bekannt, kann dieser Wert verwendet werden. Andernfalls können folgende Überlegungen angestellt werden:

- Es besteht eine gewisse Erwartung über die Datenmenge.
- Man nimmt einfach einen sehr großen Wert.

Wird eine Schätzung angewendet, können sich recht große Werte ergeben. Das Modell geht hier von den Voraussetzungen aus, die auf den mehr und mehr zum Standard werdenden 64 bit Systemen gelten. Dabei stehen virtuelle Adressräume zur Verfügung, die diese zu erwartenden Größen erfassen. Folgendes Beispiel soll das veranschaulichen :
Angenommen man liest Daten über eine 100Mbit Netzwerkverbindung ein, so ergibt sich im Idealfall eine Datenrate von 12.5 MByte/sec. Bei einer Laufzeit von einem Jahr, das sind $31,5 * 10^6$ Sekunden, kann man maximal mit einer Datenmenge von etwa $3 * 2^{47}$ rechnen.

Real zu erwartende Datenmengen werden kaum diese Werte eines Idealfalles erreichen. Es ist also anzunehmen, daß diese Größen auf den 64 bit Architekturen zu erfassen sind.

Sollte eine Längenangabe einmal nicht ausreichend sein, so besteht weiterhin die Möglichkeit, das Objekt neuerlich mit einem geändertem Offset wieder einzublenden. Das muß dann natürlich in einer Iteration erfolgen. Damit hätte man wieder die Programmstruktur des blockweisen `read()` der Standard-I/O-Methode. Wobei bedeutend weniger Schleifendurchläufe durchgeführt würden.

2.1.2 Offset und Pufferung

Um ein *remap* mit geändertem Offset zu ermöglichen, muß ein solcher im Objekt bestimmbar sein. Bei einem normalen Stream-Objekt ist das nicht der Fall. Für die Bestimmung eines Offsets wäre es nötig, den Datendurchlauf mitzuzählen.

Daraus ergibt sich die Forderung nach verschiedenen Zuständen, in denen sich ein solches Objekt befinden kann. Im Normalzustand ist das Verhalten unverändert. Wird das Objekt eingeblendet, so geht es in einen anderen Zustand über. In diesem Modus erfolgt eine Mitzählung des Datendurchlaufs, um einen Offset im Objekt bestimmen zu können. Weiters muß eine Bufferung der Daten in einem gewissen Ausmaß stattfinden, da ansonsten ein Einblenden zum Lesen nicht möglich wäre.

Der Anwender muß Informationen über das Objekt abfragen können. Auf jeden Fall muß er aktuelle Offsets ermitteln können. Der aktuelle Offset (Schreib- oder Lese-Position) ist kein Attribut einer Einblendung, sondern des Objektes (auch das Stream-Objekt im Normalzustand verfügt über solche Offsets: sie sind immer 0). Daher kann für den `ioctl()`-Aufruf eine entsprechende Funktion definiert werden.

Ein vollständige Pufferung der Daten ist praktisch nicht realisierbar, da man dabei an die realen Speichergrenzen stoßen würde. Man wird die ausgelesenen Daten, wie beim normalem Stream-Objekt, ab einem gewissen Punkt verwerfen müssen. Im Normalmodus erfolt das nach jedem einzelnen Zeichen. Für das eingeblendete Objekt sollte der Anwender den Zeitpunkt und damit auch indirekt die Position frei bestimmen können.

Dadurch geht aber die Eigenschaft der Einmaligkeit verloren. Für das eingeblendete Objekt gilt nicht mehr, daß ein Inhalt nur ein einziges Mal ausgelesen werden kann. Er kann solange ausgelesen werden, bis er verworfen wurde.

2.1.3 Datenzustände und Commit

Das explizite Verwerfen der Daten bedeutet, daß der Inhalt des Objektes verschieden Zustände annehmen kann. Es wird zwar der gesamte Datendurchsatz mitgezählt, die Daten selbst aber nur in einem bestimmten Bereich gepuffert. Die Daten die gepuffert sind, sind auch einblendbar und damit lesbar. Sie befinden sich im Zustand *gültig*. Der Inhalt der bereits verworfen wurde, ist somit *abgearbeitet*.

Betrachtet man die Schreibseite, wirft sich eine ähnliche Frage auf. Wann ist eine Information, die in das eingeblendete Objekt geschrieben wurde, auf der Leseseite sichtbar? Beim Ablauf der Standard-I/O-Methode ist eine Synchronisierung implizit enthalten. Ein Datum wird lokal solange bearbeitet, bis es fertig ist. Danach wird es mit `write()` in das Objekt transferiert und damit zum Lesen freigegeben. Das Datum existiert so lange, bis es aus dem Objekt ausgelesen wurde. Dieser Mechanismus muß sich im eingeblendeten Objekt in einer bestimmten Form wiederfinden, auch wenn das keine Entsprechung in den Eigenschaften einer eingeblendeten Datei hat.

Aus der schreibenden Perspektive gibt es ebenfalls zwei Zustände. Solange die Daten noch bearbeitet werden, befinden sie sich im Zustand *unfertig*. Ist die Bearbeitung abgeschlossen und werden sie zum Lesen freigegeben, gehen sie in den Zustand *gültig* über.

Die Zustandsänderung der Daten kann nicht automatisch erfolgen, sondern muß vom Anwendungsprogramm explizit veranlaßt werden. Es wird dafür eine Funktion benötigt, die ein *Commit* auf einem Bereich der Daten ausführt. Im `mmap()`-API ist eine solche Funktion nicht enthalten. Als Lösung gibt es die Möglichkeit das API um eine solche Funktion zu erweitern, oder eine bestehende

Funktion im API entsprechend zu adaptieren.

Die Einführung einer neuen Funktion steht im Widerspruch zur Ausgangsbedingung, das API nicht zu verändern. Für die Adaptierung einer bestehenden Funktion würde der `msync()`-Aufruf in Frage kommen. Funktionen wie z.B. `ioctl(), ftruncate()` scheiden aus, da sie auf den File-Deskriptoren und nicht auf den VMAs operieren.

Der `munmap()`-Aufruf kann wegen der Zweideutigkeit nicht verwendet werden. Der Anwender ist in der Situation, da[ß er keine Information über die zu bearbeitende Datenmenge hat. Daher verwendet er einen geschätzten Wert für den `mmap()`-Aufruf. Wenn die Bearbeitung beendet ist, ist mitunter ein Bereich der Einblendung noch vorhanden. Würde das `munmap()` auch das Commit bewirken, könnte dieser überflüssige Bereich immer nur mit Freigabe ausgeblendet werden.

Der Standard ([Ope98]) definiert den `msync()`-Call für die Anwendung auf eingeblendete Dateien. Die Auswirkung der Anwendung auf andere VMAs ist undefiniert. Es kann also eine Bedeutung für einen neuen Typ von VMAs frei festgelegt werden. Eine solche Maßnahme darf jedoch keine Auswirkung auf das Verhalten bei der Manipulation von normalen Dateien haben. In der Situation, wo das eingeblendete Objekt eine Datei oder ein Stream-Objekt ist, müssen die Schritte, die für die Behandlung des Stream-Objektes benötigt werden, ohne Auswirkungen auch auf eine eingeblendete Datei angewendet werden können.

Belegt man die `msync()`-Funktion mit der commit-Funktionalität bei der Anwendung auf ein Stream-Objekt, so ist diese Bedingung erfüllt. Ein `msync()` auf eine eingeblendete Datei veranlaßt den Abgleich des Speicherinhaltes mit dem Inhalt am externen Medium. Das kostet lediglich Zeit, hat aber weiter keinen Einfluß auf das Objekt oder den VMA.

Die Bedingung, die Daten nach dem Commit entsprechend ihres Modus zu schützen, ist etwas problematischer. Auf der Schreibseite würde es genügen, den Bereich auf den das Commit angewendet wurde, mit einem Schreibschutz zu versehen. Auf der Leseseite ist die Bedeutung des Commit aber etwas anders. Die Daten, die vom Commit betroffen sind, werden in den Zustand *abgearbeitet* übergeführt und können verworfen werden. Wichtig dabei ist, daß der Speicherplatz den die Daten belegen, freigegen wird. Das kann aber nur erfolgen, wenn sie nicht mehr verwendet werden. Solange der Speicher aber noch eingeblendet ist, kann er nicht freigegeben werden.

Dafür müsste nach jedem `msync()` ein `munmap()` auf den Bereich ausgeführt werden. Das ist weiter nicht problematisch aber unbequem. Wenn dieses Unmap selbsttätig beim `msync()` ausgeführt würde, könnte auch eine potentielle Fehlerquelle eliminiert werden. Wenn jemand vergißt, das `munmap()` nach den Commit auszuführen, könnte das zu Speicherproblemen im System führen. Durch das automatische Ausblenden wird das vermieden.

Der Umstand, daß ein `msync()` auf ein Stream-Objekt auch ein Ausblenden des Bereiches bewirkt, hat keinen Einfluß auf die Handhabung einer normalen Datei. Es macht weiter kein Problem, wenn ein `munmap()` auf einen Bereich angewendet wird, in dem gar nichts eingeblendet ist, bzw. wirkt sich ein `munmap()` auch hier positiv auf den Speicherverbrauch aus.

Die Eigenschaft, daß die Daten im eingeblendeten Stream-Objekt unterschiedliche Zustände annehmen, stellt den wesentlichsten Unterschied zum Bild der eingeblendeten Datei dar. Durch die Adaptierung des `msync()`-Aufrufes kann dieses Problem aber gelöst werden. Indem das `msync()` für das Commit - also für den Zustandsübergang - der Daten sorgt, kann diese neue Funktionalität ohne Erweiterung oder Änderung des APIs realisiert werden.

2.1.4 Das Ende

Eine Datei ist ein Objekt mit einer festgelegten Größe. Wird sie eingeblendet, so steht von vornherein fest, wo das Ende des Objektes liegt.

Blendet man ein Stream-Objekt ein, so gibt man eine Länge an, die einer Erwartung entspricht. Das tatsächliche Ende des Datenstroms kann aber an einer beliebigen Stelle liegen. Es stellt sich also das Problem, das Ende in brauchbarer Weise darzustellen.

Beim Lesen aus einem Stream-Objekt mit `read()` wird das Ende des Datenstroms durch ein EOF signalisiert. Das EOF ist aber kein Zeichen im Datenstrom, sondern der Returnwert einer Funktion. Beim Arbeiten mit einem eingeblendeten Objekt erfolgt der Zugriff aber direkt auf eine Adresse. Ein Adresszugriff hat keinen Returnwert. Es besteht auch nicht die Möglichkeit ein Zeichen zu reservieren und damit das Ende des Datenstroms zu kennzeichnen.

Im Normalbetrieb wird aber am Ende eines Datenstromes ein Signal ausgelöst. Wird eine Pipe abgebaut, so erhalten die Prozesse, die darauf zugreifen ein SIGPIPE Signal. Dieses Signal wird auch bei den anderen Streams verwendet. Dieses Signal wird nun auch beim eingeblendeten Objekt dazu verwendet werden, das Ende des Datenstromes zu signalisieren.

2.1.5 Modell

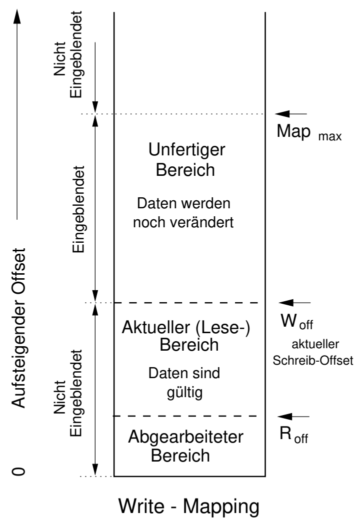

Abbildung 7: Schreib-Einblendung eines Stream Objektes

Zusammenfassend ergibt sich ein allgemeines Bild eines eingeblendeten Stream-Objektes das, wie in den Abbildungen: 7 und 7 zu sehen ist, dargestellt werden kann. Dabei wird die unterschiedliche Sichtweise aus Leser- und Schreiber-Perspektive durch die getrennte Darstellung berücksichtigt.

Diese Trennung ist deshalb sinnvoll, weil sich Stream-Objekte mitunter unterschiedlich für den Leser und den Schreiber präsentiert. Schreibt ein Prozess beispielsweise in eine Pipe und liest daraufhin aus dieser, so erhält er genau diesen Inhalt (außer ein anderer Prozess war schneller). Der gleich Vorgang auf einen Socket ausgeführt, liefert sicher nicht diesen Inhalt.

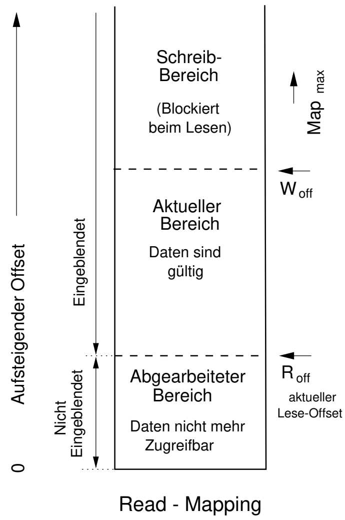

Abbildung 8: Lese-Einblendung eines Stream Objektes

Soll ein Stream-Objekt mit `mmap()` eingeblendet werden, so ist durch den Anwender ein geeigneter Längenwert festzulegen. Beim Einblenden ändert das Objekt seinen Zustand, um in der Folge über eine definierte Größe zu verfügen. Ab dem Zeitpunkt der Einblendung wird der Datendurchlauf mitgezählt. Damit ist es möglich, einen Offset im Objekt zu bestimmen. Zwei solcher Offsets sind die aktuelle Lese- und Schreibposition im Objekt (Werte R_{off} und W_{off} in Abb.: 7 und 8). Einblendungen erfolgen sinnvollerweise mit diesen Offset-Werten.

Durch das Einblenden des Objekts wird es Teil des Prozess-Adressraumes. Manipulationen auf der Schreib-Seite des Objektes dürfen nicht sofort auf die Lese-Seite durchschlagen. Auf der Lese-Seite dürfen die Daten nicht sofort verworfen werden. Daher haben die Daten im Objekt unterschiedliche Zustände: *unfertig, gültig* und *abgearbeitet.* Der Zustandsübergang des Inhaltes wird explizit durch ein Commit veranlaßt. Dabei verliert der Prozess die Zugriffsmöglichkeit auf den angegebenen Bereich.

Ein Schreib-Mapping gibt Zugriff auf den Bereich der *unfertigen* Daten. Eine Lese-Einblendung ermöglicht den Zugriff auf die *gültigen* Daten. Die Daten im Zustand *abgearbeitet* stellen die *Vergangenheit* des Objektes dar. Sie sind nur mehr quantitativ erfasst, aber nicht mehr zugreifbar.

Ein Bereich auf dem ein Commit ausgeführt wird, kann diesem Vorgang tatsächlich erst dann ausgesetzt werden, wenn der Bereichsanfang mit dem aktuellen Wert des entsprechenden Offsets (R_{off} oder W_{off}) übereinstimmt. Damit bleibt der sequentielle Charakter erhalten.

Solange das Stream-Objekt eingeblendet ist, behält es seine Vergangenheit. Wird die letzte Einblendung gelöscht, geht das Objekt wieder in den Normalzustand über und verliert seine Geschichte. Bei einer neuerlichen Einblendung beginnt die Zählung wieder bei Null.

Das Objekt kann von beliebig vielen Prozessen zugleich eingeblendet werden. Ein Commit eines Bereiches wird tatsächlich erst dann ausgeführt, wenn es von allen beteiligten Prozessen verlangt wurde. Das Objekt kann auch weiterhin mit den Standard-I/O-Mitteln manipuliert werden.

2.1.6 Funktionsschema

Wie beim Memory Mapping von normalen Dateien, wird beim `mmap()`-Aufruf vorerst nur ein VMA der angegeben Größe eingerichtet und mit dem entsprechenden Objekt (Sub-System) assoziiert. Der Inhalt wird erst beim ersten Zugriff auf eine Adresse in diesem Bereich zugeordnet. Dabei kommt das *Exception Handling*, welches auf die Paging Mechanismen aufbaut, zum Einsatz.

Das eingeblendete Stream-Objekt stellt einen Speicherbereich dar, der zwischen dem Betriebssystem und den Prozessen geteilt wird. Dabei ergibt sich folgendes Funktionsschema:

- Schreibseite: Beim allerersten Zugriff auf einen Bereich des Objektes stellt das System initialisierten Speicher zur Verfügung. Jeder weitere Zugriff auf diesen Bereich im Rahmen einer anderen Einblendung bekommt genau diesen Speicher zugewiesen. Alle Schreib-Einblendungen bearbeiten den gleichen Speicher. Sie können sich also gegenseitig überschreiben. Allfällige Synchronisation ist Applikationssache. Wird ein Commit auf einem Bereich ausgeführt, so wird der Speicher aus dem Adressraum der Applikation ausgeblendet. Dieser Speicher ist nicht mehr zu bearbeiten, auch nicht durch ein `write()`. Eingaben über Standard-I/O erfolgen immer im *unfertigen* Bereich.

- Leseseite: Ein Zugriff auf eine Adresse in einer Lese-Einblendung blockiert solange, bis der Speicher der dieser Adresse im Objekt entspricht, von der Schreib-Seite freigegeben wurde. Ist der Speicher *gültig*, so kann darauf zugegriffen werden. Zugriffe auf diesen Bereich sind nicht nur lesend möglich. Um wirklich sinnvoll verwendbar zu sein, ist auch ein schreibender Zugriff möglich. Gibt es mehrere Prozesse die lesend einblenden, so ist die eventuell nötige Synchronisation Sache der Applikationen. Speicher, der Daten im *gültigen* Zustand enthält, wird - nachdem von allen Prozessen ein Commit darauf ausgeführt wurde - freigegeben. Der Inhalt geht verloren und ist nicht mehr zugreifbar.

Bisher war immer von Bereichen die Rede, die beliebige Größen haben können. Die realen Speicherverwaltungen arbeiten aber auf Page-Basis. Das bedeutet, daß das beschriebene Modell auf diese Pages abgebildet werden muß. Dabei ergeben sich allerdings Probleme, da reale Speicherverwaltungen mit Methoden arbeiten, die mit dem Modell nicht vereinbar sind. In Kapitel 2.2 wird darauf näher eingegangen.

2.2 Bedingungen für das VMM

Reale Speicherverwaltungen arbeiten mit Einheiten des Speichers, die ein ganzzahliges Vielfaches der zugrundeliegenden, hardwareunterstützten Page-Größe entspricht. Alle Operationen die vom Memory Management auf Speicherbereiche angewendet werden können, müssen minimal auf eine ganze Speicherseite ausgeführt werden. Die Unterteilung des Speichers, bzw. des Adressraumes in Speicherseiten legt einen Raster fest. Eine Page beginnt immer auf einem fixen Rasterpunkt und nicht an einer beliebigen Stelle.

Die Operationen der Speicherverwaltung erfolgen daher immer *pagealigned* mit einer minimalen *Granularität* von einer Page.

Die Abläufe im Modell für das erweiterte `mmap()` wurden so beschrieben, als ob sie mit beliebigen Werten erfolgen könnten. In einer realen Implementierung müssen die benötigten Mechanismen allerdings auf eine reale, page-basierte Speicherverwaltung abgebildet werden.
Dabei ergeben sich zwei Hauptprobleme:

- Bei page-basierten Speicherverwaltungen werden Angaben von beliebigen Offsets auf Page-Grenzen *gerundet*, wobei mitunter Information verloren geht.

- Die Art der *Rundung* muß nicht unbedingt mit dem Modell für erweitertes `mmap()` verträglich sein.

Das erste Problem kann durch die Umstellung der Verwaltung auf *Byte-Granularität* gelöst werden. Die Lösung für das Rundungsproblem ist durch die Realisierung eines *minimierenden Verhaltens* möglich.

2.2.1 Byte-Granularität

Ein Adressbereich wird durch einen Anfangs- und einen End-Wert beschrieben. Dabei kann man entweder zwei Positionswerte, oder einen Positionswert und einen Längenwert verwenden. Ist die Grundmenge einer Rasterung unterworfen, wie es bei einem in Pages geteilten Adressraum der Fall ist, besteht nicht unbedingt die Notwendigkeit, beliebige Offsets für Bereiche zuzulassen. Man kann entweder von vornherein die Angabe von Werten, die nicht auf den Rasterpunkten liegen verbieten, oder die angegebenen Werte selbst auf Rasterpunkte verschieben.

Die virtuellen Speicherbereiche werden durch ein Wertepaar (entweder Anfangs- und Endwert oder Angangswert und Länge), das an den Page-Raster angepasst wurde, beschrieben. Die diversen Operationen auf VMAs erwarten die Angabe des Operationsbereiches in Form eines page-aligned Offsets als Anfangswert und einer beliebigen Längenangabe. Dadurch kann sich eine Obergrenze ergeben, die nicht auf einer Page-Grenze liegt. Dieser Wert wird dann selbsttätig auf die nächsthöhere Seitengrenze gezogen und natürlich auch so abgespeichert. Damit geht mitunter eine Information verloren, da alle Werte des Intervalls $(x * Pagesize, (x+1) * Pagesize]$ auf den Wert $(x+1) * Pagesize$ abgebildet werden.

Die Verwaltung erfolgt also rein mit einer *Page-Granularität.* Für das erweiterte `mmap()` wird aber die Verwaltung der Bereiche mit beliebigen Offsets benötigt. Man kommt natürlich nicht um die Einteilung des Adressraumes in Pages herum. Man kann aber die Beschreibung der VMAs um ein Wertepaar erweitern.

Zu den verwaltungstechnischen Werten $(start, end)$ bzw. $(start, len)$ kann man ein weiteres Wertepaar $(rstart, rend)$ bzw. $(rstart, rlen)$ zur Beschreibung eines VMAs heranziehen. Die r-Werte sind dabei beliebige Offsets, die irgendwo im Bereich $[start, start + Pagesize)$ bzw $(end - Pagesize, end]$ liegen. Auf die Verwaltung der Speicherbereiche selbst, hat diese Erweiterung vorerst keinen Einfluß. Nachwievor werden die, auf die Seitengrenzen gezogenen Werte, als Verwaltungsbasis verwendet. Aber es geht keine Information verloren und man hat die Möglichkeit, die exakten Werte zu den VMAs für zusätzliche Verwaltungsaufgaben zur Simulierung der *Byte-Granularität* zu nutzen.

Allerdings sind die Funktionen, die auf Speicherbereichen angewendet werden können dahingehend zu erweitern, daß sie auch beliebige Offsets als Anfangswerte für Speicherbereiche akzeptieren. Weiters ist festzulegen, wie aus den neuen Werten die Bereiche, auf die die Operation dann tatsächlich anzuwenden ist, ermittelt werden.

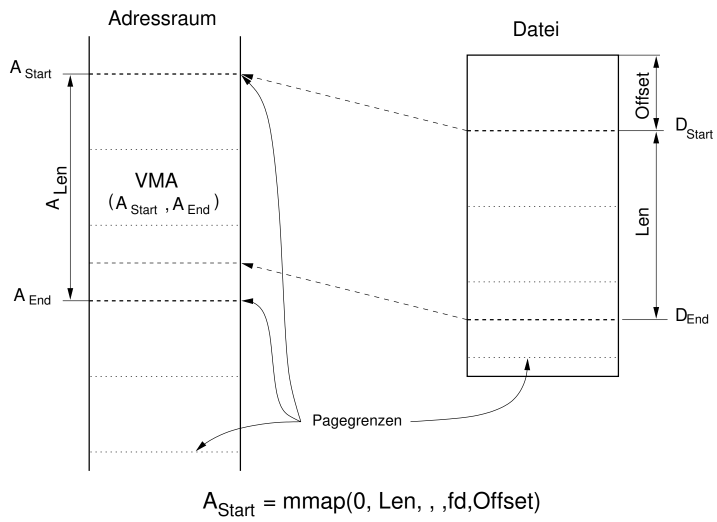

Abbildung 9: `mmap()` mit Page-Granularität

Die betroffenen Funktionen sind `mmap()`, `munmap()`, `msync()`, `mlock()` und `mprotect()`. Vor allem auf `mmap()` hat das großen Einfluß. Der Aufruf erfolgt jetzt mit einem Offset im Objekt (Datei) und einer Länge. Der Offset muß pagealigned sein. Zusätzlich kann eine Adresse angegeben werden, an der die Einblendung erfolgen soll. Diese Adresse muß ebenfalls an einer Pagegrenze liegen. Der Aufruf liefert die Anfangsadresse des Bereiches, an dem die Datei in den Speicher eingeblendet wurde. Gegebenenfalls ist das die Adresse, die dem Aufruf übergeben wurde. (Abb.: 9).

Die erweiterte Form von `mmap()` soll für den Offset und die Adresse beliebige

Werte zulassen. Real wird sich das Ergebnis nicht vom herkömmlichen Aufruf unterscheiden. Der VMA der eingerichtet wird, beginnt und endet natürlich wieder an Pagegrenzen. Allerdings liefert der Aufruf jetzt nicht mehr den pagealigned Anfangswert des Bereiches, sondern genau die Adresse, die dem angegebenen Offset entspricht, zurück (Abb.: 10). Für den Offset und die Wunschadresse ungleich 0 gilt folgende Bedingung:

```
Addr mod Pageesize = Offset mod Pagesize
```

Das bedeutet, wenn eine Adresse angegeben ist, müssen beide Werte in Bezug auf die nächstkleinere Seitengrenze den gleichen Offset haben. Hat der Aufruf freie Hand eine Adresse zu wählen, so kann ein beliebiger Wert für den Objekt-Offset angegeben werden. Mit dem erweiterten `mmap()` auf Basis eines VMM mit Byte-Granularität, kann ein Objekt mit einem beliebigen Offset eingeblendet werden.

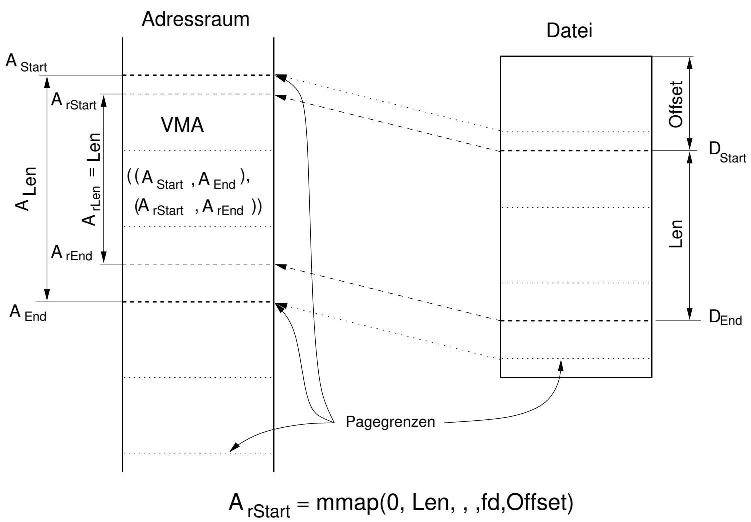

Abbildung 10: `mmap()` mit Byte-Granularität

Diese Änderung im Verhalten von `mmap()` ist völlig transparent für alle bestehenden Anwendungen. Für die Programme, die bisher davon ausgegangen sind, daß sie nur pagealigned Offsets und Adressen verwenden können, ergibt sich keinerlei Veränderung, da alle Requests mit korrekt angegebenen Parametern auch weiterhin - wie gewohnt - erfüllt werden.

Mit `mmap()` wird ein VMA eingerichtet. Mit den anderen Funktionen werden solche Bereiche manipuliert. Diesen Funktionen werden Bereiche angegeben, auf die sie die gewünschte Operation ausführen sollen. Ihr Interface muß natürlich auch so angepasst werden, daß sie die Angabe von beliebigen Offsets akzeptieren. Dabei sind aber neue Regeln festzulegen, mit denen die angegebenen Werte interpretiert bzw. angepasst werden.

2.2.2 Minimierendes Verhalten

In den Definitionen der Operationen auf virtuellen Speicherbereichen wurde ein Verhalten in der Form festgelegt, daß die Operation sicher auf alle Teile des angegebenen Bereiches angewendet wird. Dieses Verhalten wird in der Folge *maximierendes* Verhalten genannt. Wird diese Strategie angewandt, so können auch Teile eines Bereiches betroffen sein, von denen das nicht verlangt wurde. Der Grund dafür ist, daß der Endwert, der sich aus den angegeben Parametern `Anfangsadresse` und `Länge` berechnet, auf die nächsthöhere Pagegrenze gerundet wird.

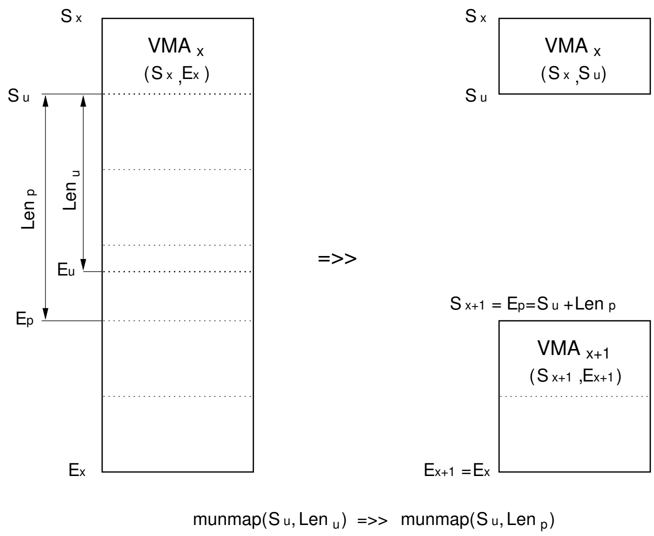

Abbildung 11: *Maximierendes* `munmap()` bei Page-Granularität

Die gegenteilige Strategie würde die Operation sicher nur auf jene Bereiche anwenden, für die das explizit verlangt wurde. Dabei würde der Endwert auf die nächste Page-Grenze abgerundet werden. Der Anwendungsbereich wird minimiert - *minimierendes* Verhalten.

Jede dieser Strategien wäre in Bezug auf bestimmte Fälle besser geeignet. So könnte im Falle eines `munmap()` bei Maximierung des Bereiches versucht werden, auf eine Adresse zuzugreifen, die zwar nicht explizit ausgeblendet wurde, aber durch die Bereichserweiterung bereits ungültig geworden ist. Das hätte eine Adress-Verletzung mit einem SIGSEGV zur Folge.

Ideal wäre also, das Verhalten vom Anwendungsfall abhängig zu machen. Das würde aber das Arbeiten mit VMAs komplizierter gestalten, da man sich immer überlegen müßte was in der Situation das gerade "bessere" oder "schlechtere" Verhalten wäre.

Diese Abhängigkeit von der Semantik der Funktion dürfte einer der Gründe sein, warum man sich für die generelle Bereichsmaximierung entschieden hat. Ein zweiter Grund liegt darin, daß bei der Bereichsminimierung die Gefahr besteht, daß verwaiste VMAs entstehen.

Eine Speicherverwaltung, die eine Byte-Granularität auf einer Page-basierten

Basis implementiert, kann aber nur bei Verwendung der Bereichsminimierung realisiert werden. Prinzipiell dürfen die angegebenen Adressen in keiner Weise mehr gerundet werden, da die genauen Werte für die Verwaltung benötigt werden. Abb.: 11 zeigt wie ein `munmap()` nach dem maximierenden Verfahren auf einen Bereich bei herkömmlicher Speicherverwaltung angewendet wird. Die Anfangsadresse S_u muß pagealigned sein. Der Wert von Len_u wird auf die nächstgrößere Pagegrenze gezogen und ergibt Len_p.

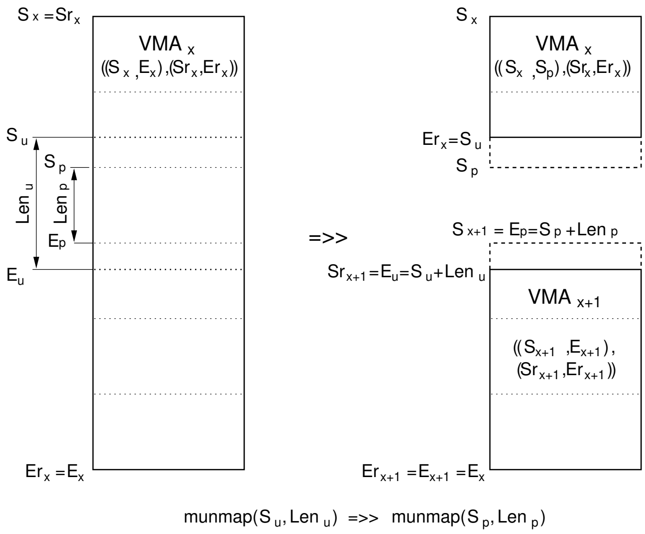

Abbildung 12: *Minimierendes* `munmap()` bei Byte-Granularität

Werden die Speicherbereiche mit beliebigen Offsets verwaltet, so sollte ein vergleichbarer Aufruf von `munmap()` ein Ergebnis liefern, wie in Abb.: 12 dargestellt. Da jetzt auch die Anfangsadresse nicht mehr pagealigned sein muß, wird hier ein Wert mit einem beliebigen Offset angenommen. Ein Aufruf der Form:

`munmap(`S_u`,` Len_u`);`

hat also verwaltungstechnisch dieselbe Wirkung wie ein Aufruf:

`munmap(`S_p`,` Len_p`);`

wobei sich S_p wie folgt berechnet:

`PAGE_MASK = (~(PAGE_SIZE-1))`

S_p `= (`S_u `& ~PAGE_MASK) ? (`S_u `& PAGE_MASK) + PAGE_SIZE:` S_u

Len_p und damit der Endpunkt E_p des Bereiches muß analog dazu berechnet werden. Sei zunächst der Endpunkt des Unmap-Bereiches

E_u `=` S_u `+` Len_u

dann wird der verwaltungstechnische Endpunkt E_p wie folgt berechnet:

E_p `= (`E_u `& PAGE_MASK)`

und die reale Unmap-Länge ist:

$$Len_p = E_p - S_p$$

Dieser Formalismus beschreibt das Gegenteil zum Verfahren, welches jetzt bei den Operationen auf VMAs verwendet wird. Es erfolgt immer eine Rundung der Grenzen zur nächsten, innerhalb des Bereiches liegenden Seitengrenze - Bereichsminimierung, *minimierendes* Verfahren. Würden die Grenzen nach außen gerundet (maximiert), würden die offsetbedingten Teile ebenfalls betroffen.

Für die Realisierung der Speicherverwaltung mit Byte-Granularität, bedarf es noch einiger Zusätze am minimierenden Verfahren. Wesentlich ist, daß es durch die Angabe nicht zu einer Teilung eines Bereiches derart kommt, daß Teile einer Page in verschiedenen VMAs zu liegen kommen würden. Weiters ist festzulegen, wie ein Wert zu interpretieren ist, der zwischen der verwaltungstechnischen Grenze und der tatsächlichen Grenze liegt.

Obige Formeln für die Berechnung der Werte S_p und E_p bzw. Len_p müssen also noch ergänzt werden. Liegt ein Wert S_p im Intervall

$$[(S_{op}\ \texttt{\&}\ \texttt{PAGE\_MASK})\ ,\ S_{op}]$$

dann kann diese Seite in die Operation miteingeschlossen werden. Es wird also:

$$S_p = S_{op}\ \texttt{\&}\ \texttt{PAGE\_MASK}$$

sein. Ist das nicht der Fall, bleibt also ein Teil der Seite von der Operation ausgeschlossen, dann fällt die Seite zur Gänze aus dem Bereich. Der Wert für S_p berechnet sich dann wie in der ersten Formel angegeben durch:

$$S_p = (S_{op}\ \texttt{\&}\ \texttt{PAGE\_MASK}) + \texttt{PAGE\_SIZE}$$

Genauso muß für das Ende des Bereichs vorgegangen werden. Für einen Wert $E_{op} = S_{op} + Len_{op}$ der im Intervall

$$[E_{op}\ ,\ (E_{op}\ \texttt{\&}\ \texttt{PAGE\_MASK}) + \texttt{PAGE\_SIZE}]$$

liegt, wird die letzte Seite des Bereiches in die Operation miteingeschlossen und E_p wird zu:

$$E_p = (E_{op}\ \texttt{\&}\ \texttt{PAGE\_MASK}) + \texttt{PAGE\_SIZE}$$

Andernfalls wird entsprechend dem minimierenden Verhalten die Operation sicher nicht auf einen Teil angewendet, für den das nicht verlangt wurde. Die letzte Seite des Bereiches bleibt ausgeschlossen:

$$E_p = E_{op}\ \texttt{\&}\ \texttt{PAGE\_MASK}$$

Abb.: 13 zeigt, welcher Anwendungsbereich der Operationen sich in den Sonderfällen aus den möglichen Parameterwerten ergeben. Diese Fälle beinhalten immer Randpages und können daher nicht einfach nach den Regeln des minimierenden Verhaltens behandelt werden. Es können also bei allen Operationen immer beliebige Offsets angegeben werden, wobei in allen Fällen verwaltungstechnisch minimierendes Verhalten angewendet wird. Bei allen Operationen außer `munmap()`, insbesondere bei denen, die zu einer Teilung eines VMAs führen können (`mprotect(), mlock()`), wird ein Offset nicht nur verwaltungstechnisch, sondern auch tatsächlich nach dem minimierenden Verfahren gerundet. Das gilt für die Obergrenze genauso, wie für die Untergrenze. Das ist notwendig, da es ansonst zu einer Teilung eines VMAs innerhalb einer Page kommen würde. Die Grenzwerte der neu entstehenden VMAs werden daher nicht auf Basis der angegeben Parameter sondern aus den minimierten Werten berechnet.

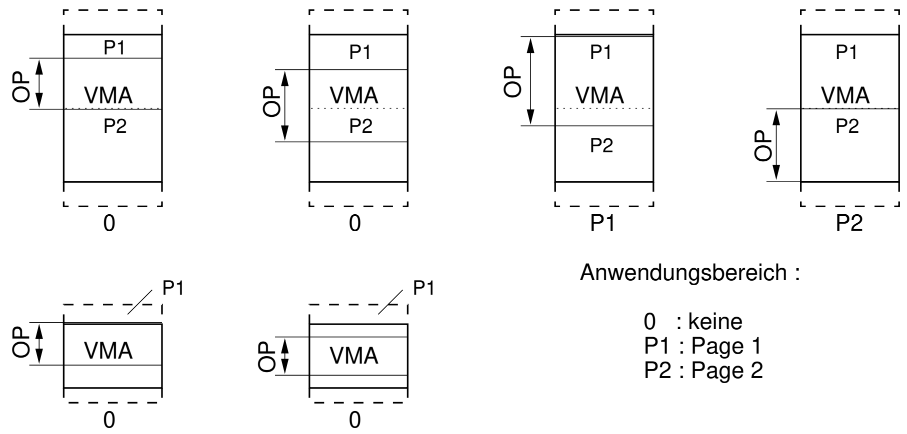

Abbildung 13: Anwendbarkeit von Operations-Bereichen

Bei `munmap()` kommt es nicht zu einer Teilung in dem Sinn, daß einfach ein Bereich an einer Stelle aufgetrennt wird, sondern es wird immer ein Loch erzeugt, da ein gewisser Teil ausgeblendet wird. Hier werden die Grenzwerte (r-Werte) der verbleibenden VMAs aus den echten Offsets ermittelt. Einzige Ausnahme ist der Fall wo der Bereich kleiner als eine Page ist und mit keiner VMA- oder Page-Grenze zusammenfällt. Hier wird ebenfalls entsprechend dem minimierenden Verfahren vorgegangen und die Operation auf keine Page angewendet.

Natürlich trägt der Umstieg auf die Verwaltung mit Byte-Granularität und die Anwendung des minimierenden Verfahrens nicht gerade zur Vereinfachung bei, was aber auch nicht zu erwarten war. Wichtig ist vor allem, daß sich das API und das Verhalten der Funktionen für bestehende Programme nicht ändert. Daß das mit den vorgestellten Modellen möglich ist, zeigt die Implementierung unter Linux, die im Abschnitt 4 beschrieben ist.

3 Erweitertes `mmap()` für Pipes

3.1 Die Pipe

Pipes stellen unter Unix ein oft verwendetes Mittel zur Interprozess-Kommunikation dar. Der Zugang dazu steht auf zwei Arten zur Verfügung. Der System-Call `pipe()` erlaubt es dem Programmierer eine IPC-Verbindung zwischen verwandten Prozessen herzustellen. `pipe()` liefert ein Array von zwei File-Deskriptoren. Jeweils einen als Lese-(`pipe[0]`) und Schreib(`pipe[1]`)-Ende der Pipe. Da ein Filedeskriptor nur ein Index in der prozesseigenen Deskriptoren-Tabelle ist, hat er nur lokal eine Bedeutung. Soll also mit Hilfe einer Pipe eine Interprozess-Kommunikation möglich sein, muß sie vor einem Fork erzeugt werden. Beim Fork erbt der Sohn die geöffneten Dateien und damit auch die Deskriptortabellen vom Vater-Prozess. Damit macht ein Index in der Deskriptortabelle auch im neuen Prozess-Image Sinn. Beide Prozesse können auf die Pipe mit Hilfe der Standard-I/O-Funktionen `read()/write()` zugreifen, wobei das jeweilige Ende für die entsprechende Funktion zugänglich ist.

Auch wenn sich die Schnittstelle, über die eine geöffnete Pipe angesprochen wird genauso präsentiert wie eine Datei, so unterscheidet sie sich doch wesentlich von einer solchen. Auf eine Datei kann wahlfrei zugegriffen werden. Eine Pipe zeigt ein *First In - First Out (FIFO)*-Verhalten. Es kann also nichts an der Reihenfolge des Inhaltes geändert werden. Schreiben ist nur im Anschluß an das zuletzt Geschriebene möglich, gelesen kann nur das jeweils nächste Zeichen werden.

Genauso, wie bei einer Datei, ist ein Inhalt, der in die Pipe geschrieben wurde, für alle sichtbar. Also kann der Inhalt auch von dem Prozess ausgelesen werden, von dem er geschrieben wurde. Allerdings kann er nur einmal ausgelesen werden, da die gelesenen Zeichen dabei aus der Pipe entfernt werden. Eine Datei hingegen kann beliebig oft ausgelesen werden.

3.1.1 Named Pipes (FIFOs)

Um eine Pipe auch für die Kommunikation zwischen beliebigen Prozessen verwenden zu können, kann sie im Filesystem repräsentiert werden. Dafür steht auf Kommandoebene der Befehl `mkfifo` zur Verfügung. Dieses Kommando greift seinerseits auch auf die Library-Funktion `mkfifo()` zurück, die dem Programmierer das Anlegen einer Named Pipe erlaubt. Eine *Named Pipe* (FIFO) wird genauso wie eine normale Datei über einen Namen angesprochen. Prozesse die miteinander kommunizieren wollen, brauchen nur den Namen der zu verwendenden FIFO miteinander zu vereinbaren. Sie können die FIFO entsprechend den Datei-Zugriffsrechten öffnen und verwenden. Die Art wie eine geöffnete FIFO verwendet wird, gleicht der von offenen Dateien mehr, als der von Pipes. Bei einer Pipe erfolgt die Ein- und die Ausgabe über verschiedene Deskriptoren. Wird eine Named Pipe O_RDWR geöffnet, so kann ein und derselbe Deskriptor für Ein- und Ausgabe verwendet werden. Natürlich kann eine Anpassung an das Pipe-Interface erfolgen. Dazu braucht ein Prozess eine FIFO zur zweimal zu öffnen. Einmal mit O_RDONLY und einmal mit O_WRONLY.

Das wird aber davon abhängen, aus welcher Perspektive die FIFO betrachtet wird. Hier ist eine FIFO die Repräsentation einer Pipe im Filesystem. Alternativ dazu kann eine FIFO auch als effizientere Variante des IPC-Mittels Datei gesehen werden.

Die einfachste Möglichkeit für Prozesse miteinander Daten auszutauschen besteht darin, gemeinsam auf eine Datei zuzugreifen. Was aber möglicherweise die Kosten von tatsächlichen Lese- und Schreib-Operationen am externen Speichermedium mit sich bringt. Zusätzlich müssen sich die Prozesse um eine Synchronisation kümmern. Eine FIFO kann auch als eine spezielle Datei zur Interprozesskommunikation mit einem einfachen Synchronisationsmechanismus (FIFO) gesehen werden. Dieser Sicht kommt wiederum der Zugang über einen einzigen Read/Write-Deskriptor entgegen, da sie dann genauso wie eine normale Datei behandelt werden kann.

3.1.2 Anwendung von Pipes

Am häufigsten werden Pipes beim Aufbau von Filterketten die auf einen Datenstrom angewendet werden sollen, verwendet. Unter Unix sind Kommandoaufrufe die solche Strukturen realisieren, sehr gebräuchlich. Diese Aufrufe haben etwa folgende Form:

```
Filter_1 < Input | Filter_X >  Output
```

Die Filterprogramme können im Allgemeinen auf zwei Varianten mit den Eingabedaten versorgt werden. Entweder man übergibt ihnen einen Dateinamen, oder sie lesen einfach von `stdin` (der Standard-Eingabe). Ebenso verhält es sich mit der Ausgabe. Wird ein Dateiname dafür angegeben, so öffnet sie das Programm oder die Ausgabedaten werden direkt nach `stdout` (Standard-Ausgabe) geschrieben. Die Programme arbeiten in beiden Fällen nach demselben Prinzip. Der Datei-Deskriptor der für die Ein- oder Ausgabe verwendet wird, hat entweder den Wert der geöffneten Datei, oder den des entsprechenden Standard-Deskriptor (`stdin, stdout`) zugewiesen bekommen. Auf diesen Deskriptoren wird mit den Standard-I/O-Funktionen `read()` und `write()` gearbeitet. Wenn dem Programm einer der Standard-I/O-Kanäle zur Verwendung angegeben wurde, ist es für dieses nicht unbedingt ersichtlich, um welche Art von Objekt es sich handelt. Daher ist es kaum möglich festzustellen, ob darauf mit Memory Mapping gearbeitet werden könnte, oder nicht. Darüberhinaus müsste das Programm zwei unterschiedliche Methoden der Datenbearbeitung implementieren. Für einblendbare Dateien erfolgt die Bearbeitung der Daten direkt im Speicher. Wenn es sich um Pipes oder ein undefinierbares Objekt handelt, werden die Methoden unter Verwendung der Standard-I/O-Methoden herangezogen. Im Sinne der einfacheren Programmierung ist es natürlich klar, daß daher in allen Fällen nur mit den Standard-I/O-Methoden gearbeitet wird.

Werden solche Programme in Filterketten der zuvor angegeben Form verwendet, so sind die I/O-Kanäle, mit denen sie arbeiten, Pipes. Für die Programme ist das aber nicht direkt erkennbar, da sie nur angewiesen werden die Standard-I/O-Kanäle zu verwenden. Die *Shell*, die diese Kommandos interpretiert und ausführt, arbeitet zu diesem Zweck mit der Möglichkeit der Umlenkung der Standard-I/O-Kanäle. Mit Hilfe von `fork()` werden einzelne Prozesse erzeugt, die über `exec()` die angegebene Programme aufrufen. Deren Standard-I/O-Kanäle werden zuvor auf, durch die Shell angelegte Pipes umgelenkt, sodaß eine Verknüpfung der einzelnen Prozesse über diese Pipes entsteht.

Die Form, in der die Pipes hier verwendet werden, ist eigentlich ein Einbahnverkehr. Es wird nicht das volle Potential der Pipes genutzt, da eigentlich kein Bedarf nach einer Rückmeldung vom empfangenden zum sendenden Prozess besteht. Die Pipe wird dabei eher auf die Eigenschaften, die ein Stream Objekt beschreiben, reduziert. Auf dieser Art des Einsatzes lag auch das Hauptaugenmerk bei der Entwicklung des Modells für das erweiterte `mmap()`. Auch wenn das Modell in Situationen, wo Pipes in ihrem vollen Funktionsumfang verwendet werden angewendet werden kann, so wird der meiste Nutzen in den Anwendungsfällen der obigen Struktur zu erzielen sein.

3.2 Modell des Pipe-Mappings

Wie kann also das Modell für das erweiterte `mmap()` auf Pipes angewendet werden? - Wie könnte in der Praxis damit gearbeitet werden?

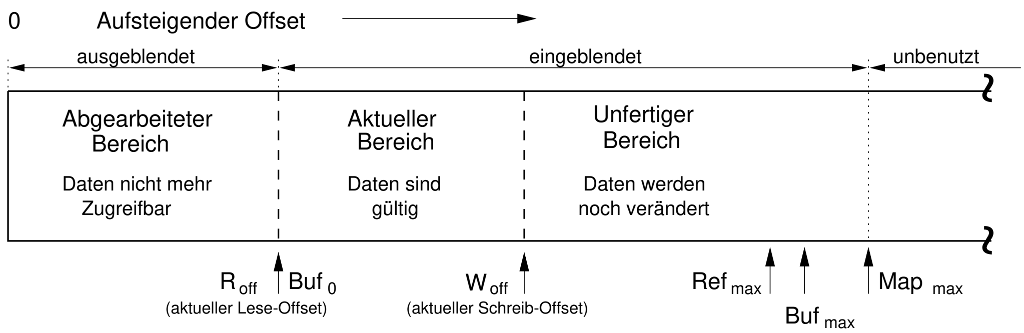

Abbildung 14: Modell einer eingeblendeten Pipe

In Abb.: 7 und 8 wurde das Modell in allgemeiner Form dargestellt. Die Trennung der Lese- und Schreib-Sicht ist für die Pipe nicht nötig, da ein Wert, der von einem Prozess hineingeschrieben wurde, von diesem auch sofort wieder ausgelesen werden kann. Daher kann man sie als ein Objekt auffassen, bei dem beide Perspektiven aneinandergekoppelt sind, wie in Abb.: 14 dargestellt.

Unabhängig von Zustand, kann eine Pipe immer wie in Abb.: 14 betrachtet werden. Ist sie im Normalzustand (d. h. von keinem Prozess eingeblendet), dann ist der Wertebereich in dem sich die Offsets bewegen können, auf die Größe des internen Puffers beschränkt. R_{off} klebt sozusagen immer an der Basis, bei Offset 0, fest. Wird in die Pipe geschrieben, so kann sich W_{off} maximal bis zur Puffergröße erhöhen. Einen Lesevorgang kann man sich in der Art vorstellen, daß R_{off} um die Anzahl der gelesenen Zeichen erhöht und danach beide Werte - R_{off} und W_{off} um diesen Betrag wieder reduziert werden. Damit bleibt R_{off} immer gleich 0 und W_{off} kann den Betrag der Puffergröße nicht übersteigen.

Wird eine Pipe mit `mmap()` in den Adressraum eines Prozesses eingeblendet, ändern sich zwei Bedingungen. Die Begrenzung durch die interne Puffergröße fällt weg und die Fixierung von R_{off} am Nullpunkt wird gelöst. Durch diese Loslösung wird ein absoluter Referenzpunkt geschaffen, über den die Zuordnung von Bereichen in der Pipe und den VMAs der Einblendungen hergestellt werden kann. Beim Einblenden wird die Anfangsadresse des VMAs dem angegebenen Offset im Pipebuffer zugeordnet. Die Mitzählung des Datendurchflußes beginnt und damit sind auch Offsets bestimmbar.

Offsets werden benötigt um die Pipe in die unterschiedlichen Bereiche zu teilen. Alles was sich zwischen dem Nullpunkt und dem Wert von R_{off} befindet, hat die Pipe bereits durchlaufen und kann, entsprechend den Pipe-Eigenschaften, auch nicht mehr angesprochen werden. Daher macht es keinen Sinn, bei einem `mmap()` der Pipe, einen Offset in dieser Zone als Basiswert anzugeben. Ein Aufruf mit dieser Parametrisierung schlägt fehl.

Die Zone zwischen R_{off} und W_{off} legt zu jedem Zeitpunkt den zugreifbaren Inhalt der Pipe fest. Prozesse, die die Lese-Pipe mit einem passenden Offset einblenden, erhalten Zugriff auf diese Inhalte. Sie können solange zugegriffen werden, bis sie mittels `msync()` explizit als abgearbeitet erklärt werden (Commit).

Ab W_{off} beginnt der Schreibbereich der Pipe. Versuche, die Schreib-Pipe mit einem Offset kleiner als diesem Wert einzublenden, schlagen ebenfalls fehl. Prozesse, die die Schreib-Pipe einblenden, können in diesem Bereich beliebig Lesen und Schreiben. Diese Inhalte sind solange für die Leser unzugänglich, bis sie über ein `msync()` explizit freigegeben werden.

Die Größe des Schreibbereiches, und damit der gesamten zugreifbaren Pipe ist natürlich einer Begrenzung unterworfen. Beim Versuch eine neue Adresse im Schreibbereich anzusprechen, die über diese Grenze (Buf_{max}) hinausgeht, wird ein Prozess solange warten müssen, bis am anderen Ende, auf der Leseseite, die untere Grenze (R_{off}) verschoben wird. Die Größe der Zugriffszone (Buf_0, Buf_{max}) sollte änderbar sein.
Damit ist das Verhalten einer eingeblendenten Pipe in den Grundzügen festgelegt. Bei der genaueren Betrachtung zeigen sich aber noch einige Probleme.

3.2.1 `msync()` und `munmap()`

Die Werte R_{off} und W_{off} werden durch einen `sync()`-Aufruf (Commit) verändert. Dabei erfolgt auch ein `munmap()` des betroffenen Bereichs.

Mit einem `munmap()` soll ein Ausblenden der Pipe ohne eine Bestätigung der Daten möglich sein. Das `munmap()` kann auf Teilbereiche oder auf den gesamten Einblendungsbereich angewendet werden. In beiden Fällen ist die Wirkung identisch. Das `msync()` hat aber eine unterschiedliche Wirkung. Diese und die Unterschiede bei der Anwendung auf eine Read- oder eine Write-Pipe werden in der folgenden Liste dargestellt.

Pro Operation (`msync(), munmap()`) müssen 4 Fälle betrachtet werden, die sich aus den beiden Kriterien Read- oder Write-Pipe und Teilbereich oder gesamte Einblendung, ergeben:

- `msync()`: Für die Anwendung auf einen Teilbereich einer Write-Pipe ist die Wirkung genau so, wie sie für das Commit definiert ist. W_{off} wird angepasst und der Teilbereich wird ausgeblendet. Auf die ganze Einblendung angewendet, wird das Commit auf jenem Bereich ausgeführt, der tatsächlich referenziert wurde. W_{off} wird also bis zur letzten, tatsächlich zugegriffenen Adresse nachgezogen. Für den Rest des Bereiches wird nur ein `munmap()` ausgeführt.

  Bei einer Read-Pipe ist die Wirkung immer gleich. R_{off} wird entsprechend angepasst und der angegebene Bereich wird ausgeblendet.

- `munmap()`: Hier ist das Verhalten konsistent. Egal ob Teil- oder Gesamtbereich, ob Read- oder Write-Bereich, es erfolgt keine Anpassung von W_{off} und der Bereich wird ausgeblendet.

3.2.2 Deskriptor-Identifikation und Zugriffsmodus

Intern ist die Pipe durch ein einzelnes Objekt repräsentiert. Von außen wird die Pipe aber durch zwei Deskriptoren beschrieben. Dadurch ist bei einem `mmap()` eindeutig festgelegt, welche Ausprägung der Pipe eingeblendet werden soll. Wird eine *named Pipe* (FIFO) behandelt, ist die Sachlage nicht immer eindeutig. Eine FIFO wird auf Filesystemebene wie eine normale Datei behandelt. Sie kann auf drei Arten geöffnet werden. Nur zum Lesen (O_RDONLY), nur zum Schreiben (O_WRONLY) oder für beide Operationen (O_RDWR). Wird dem `mmap()`-Aufruf einer der beiden eindeutigen Deskriptoren übergben, ist die Situation klar. Was aber im Falle des Read/Write-Deskriptors angenommen werden soll, ist nicht von vornherein festgelegt. Eine Lösung für diesen Fall ist gemeinsam mit der, für das nächste Problem gefundenen, möglich.

Wie gerade gezeigt, wird durch die Zugriffsrechte des File-Deskriptors eindeutig festgelegt, welches Pipe-Interface angesprochen wird. Die Lese-Seite wird durch einen *readonly*- und die Schreibseite durch einen *writeonly*-Deskriptor beschrieben. Entsprechend der Definition für den `mmap()`-Aufruf ([Ope98], Kap. 1.3.1), wird das solange problemlos funktionieren, solange man eine Read-Pipe tatsächlich nur *readonly* und eine Write-Pipe nur *writeonly* einblenden will. Diese Einschränkung ist aber nicht sinnvoll, da man dadurch in dem Speicher, der die Einblendung darstellt, nicht arbeiten kann.

Das Problem lässt sich leider nicht ohne eine Erweiterung des APIs lösen. Die Menge der, für den `mmap()`-Aufruf festgelegten Flags, wird um zwei Werte erweitert. Durch die neuen Flag-Werte MAP_RDSTREAM und MAP_WRSTREAM hat der Anwender die Möglichkeit, bei Mehrdeutigkeit des Deskriptors festzulegen, wie dieser zu interpretieren ist.

Diese Erweiterung der Parameter am `mmap()`-Interface ist noch nicht problematisch. Eine FIFO kann jetzt O_RDWR geöffnet und über das Flag deklariert werden. Der Mode eines Pipe Deskriptors kann aber nicht verändert werden. Es muß vom `mmap()`-Interface verlangt werden, daß in den Fällen, wo eines der beiden neuen Flags gesetzt ist, auf eine genaue Überprüfung der Zugriffsrechte verzichtet wird. Nur so ist es möglich, daß eine Read-Pipe die durch einen *readonly*-Deskriptor identifiziert wird, auch Read/Write eingeblendet werden kann. Entsprechendes gilt natürlich für eine Write-Pipe.

Die STREAM-Flags müssen aber nicht unbedingt angegeben werden. Fehlen sie, wird der `mmap()`-Aufruf das gewohnte Verhalten zeigen. Ohne diese Flags kann eine FIFO, die readonly oder writeonly geöffnet wurde, mit eben diesen Zugriffsrechten eingeblendet werden. Genauso verhält es sich bei einer Pipe.

Wird ein Read/Write-FIFO-Deskriptor ohne STREAM-Flags angegeben, liefert `mmap()` einen Fehler (EACCES). Derselbe Fehler wird angezeigt, wenn eine Pipe oder ein readonly/writeonly FIFO-Deskriptor mit Lese- und Schreib-Recht eingeblendet werden soll. Das angegebene STREAM-Flag muß es zum angegebenen Deskriptor passen. Der Versuch eine Lese-Pipe mit dem Flag MAP_WRSTREAM anzugeben, wird durch den Fehler EACCES angezeigt. Analog dazu das Verhalten für die Schreib-Pipe und entsprechende FIFO-Deskriptoren. Bei Verwendung eines STREAM-Flags in Kombination mit einem Deskriptor, der kein Stream-Objekt identifiziert, wird der Fehler EINVAL geliefert.

Die beiden neuen Flags dienen also dazu, dem `mmap()` bei Angabe mit einem O_RDONLY oder O_WRONLY Deskriptor mitzuteilen, auf die exakte Überprüfung der Zugriffsrechte zu verzichte. Bei einem O_RDWR-FIFO-Deskriptor wird damit mitgeteilt, welche Ausprägung des Objekts angesprochen ist.

Diese Erweiterung der Menge der MAP-Flags hat keinerlei Einfluß auf das API in Bezug auf die gewohnte Funktionalität. Genauso beschränkt sich der Verstoß gegen die Konvention bei der Kontrolle der Zugriffsrechte auf den Fall, wo dies explizit durch die Angabe eines STREAM-Flags verlangt wird. Dadurch ist sichergestellt, daß keine Veränderung am API für bestehende Applikationen erfolgt.

3.2.3 Mehrere Einblendungen

Bisher wurde das Verhalten nur für den Fall, bei dem jeweils ein Prozess die Lese- und die Schreib-Pipe einblendet, betrachtet. Das beschreibt auch die Struktur der Anwendungen, die als Hauptzielgruppe für das erweiterte `mmap()` betrachtet wurden. Das Modell muß aber auch den Fall abdecken, wo mehrere Prozesse die Pipe einblenden.

Jeder Prozess der die Pipe einblendet, hat eine Sicht des Objektes, wie in Abb.: 14 dargestellt. Die interne Verwaltung des Objektes speichert zu jeder Einblendung der Pipe zumindest die charakteristischen Werte (Prozess-ID, R_{off}, W_{off}) ab. Daher gibt es für jede Pipe eine Menge von Read- und Write-Offsets (Abb.: 15). Aus diesen prozessbezogenen Werten muß ein Wert generiert werden, der für das ganze Objekt gilt. Entsprechend der Bedeutung der Offset-Werte, ergibt sich der Gesamtwert als jener Offset, bis zu dem die Operation von allen beteiligten Prozessen angewendet wurde.

Damit erhält man den Schreib-Offset W_{off} für eine Pipe als das Minimum aller prozessbezogenen Schreib-Offsets Wn_{off}. Analog dazu ist R_{off} das Minimum aller Werte Rn_{off} (Abb.: 15).

Führt ein Prozess ein Commit (`msync()`) auf einen Bereich aus, so wird sein eigener Offset-Wert (je nach Mapping) auf den Endwert des angegebenen

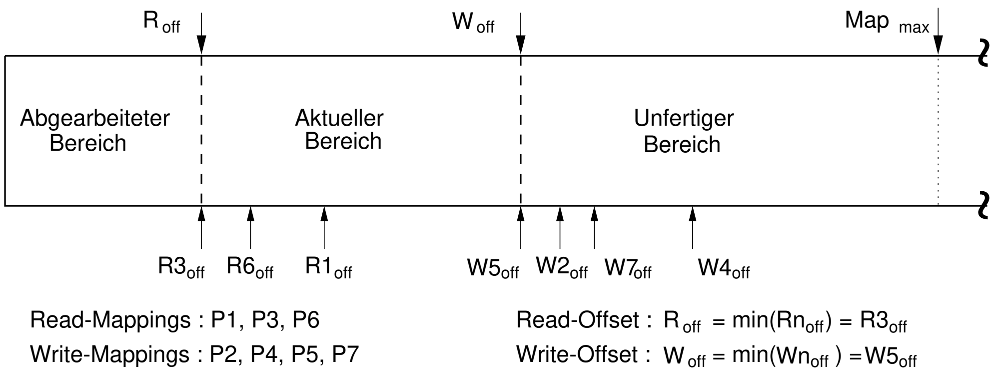

Abbildung 15: Mehrfach eingeblendete Pipe

Bereiches gezogen. Der Gesamt-Offset wird aber nur bis zum nächsten Privat-Offset gezogen. War der ursprüngliche, prozessbezogene Offset bereits größer als der kleinste prozessbezogene Wert, erfolgt keine Änderung am Gesamt-Offset. Dieser Wert wird tatsächlich nur dann verändert, wenn der betreffende Prozess zuletzt den minimalen Offset geliefert hat. Im Fall, wo mehrere Prozesse eine Schreib-Pipe eingeblendet haben, bedeutet also ein Commit auf einen Bereich nicht unbedingt, daß die Daten des gesamten Bereichs tatsächlich abgeschlossen und zum Lesen freigegeben wurden. Die Zone die auch noch von anderen Prozessen eingeblendet ist, kann daher noch von diesen verändert werden. Diese Eigenschaft erbt das Pipe-Mapping vom File-Mapping. Beim Commit einer Read-Pipe spielt diese Eigenschaft keine Rolle.

Schwieriger ist die Situation, wenn ein Prozess ein `munmap()` durchführt. Ursprünglich wurde festgelegt, daß bei einem `munmap()` der Bereich aus dem Adressraum des Prozesses ausgeblendet wird und der entsprechende Offset unverändert bleibt. Das kann vorerst in allen Fällen tatsächlich auf diese Art erfolgen. Der prozessbezogene Offset wird angepasst, bzw. wenn der gesamte Bereich ausgeblendet wird, wird er überhaupt verworfen. Der Gesamt-Offset (Bereichsanfang) bleibt dabei unverändert.

Dabei darf aber nicht vergessen werden, was mit einem `munmap()` im Gegensatz zu einem `msync()` bezweckt wird: Beim `msync()` soll eine Zustandsänderung des angegebenen Bereiches erfolgen. Ein `munmap()` bedeutet aber, daß der betroffene Bereich nicht weiter bearbeitet werden soll.

Für die Leseseite stellt das kein besonderes Problem dar. Die Wirkung ist gleich wie beim `msync()` - die Daten werden verworfen. Für die Write-Pipe bedeutet das `munmap()` aber, daß die Inhalte des Bereiches (aus der Sicht des ausführenden Prozesses) nicht zum Lesen freigegeben werden sollen. Der Prozess hat lediglich das Interesse an dem Bereich verloren. Solange die Prozesse, die mit Schreib-Mappings der Pipe arbeiten, nur ein `munmap()` ausführen, funktioniert das auch, da ja nie der Bereichsanfang verschoben wird. Ein Problem entsteht in dem Augenblick, wo einer der Prozesse ein `msync()` ausführt. Dabei gibt es zwei Fälle zu unterscheiden.

1. Der Write-Offset des Prozesses war bestimmend für den Bereichsanfang

W_{off}. Da der Bereich von diesem Prozess als gültig erklärt wird, kann keine Beurteilung erfolgen, von wem die Daten des Bereiches tatsächlich stammen. Der neue Wert für W_{off} wird wie gewohnt ermittelt. Der angegebene Bereich wird zum Lesen freigegeben.

2. Der prozessbezogene Offset, an dem der `msync()`-Bereich beginnt, ist größer als W_{off}. Damit stellt sich die Frage, was mit dem Bereich der zwischen W_{off} und Wp_{off} (prozesspezogener Offset) liegt, geschehen soll. Auf diesen Bereich wurden lediglich `munmap()`-Operationen angewendet, womit der Inhalt verworfen werden soll. Der Wert von W_{off} muß aber über diese Inhalte hinwegwandern, was die Freigabe dieser Daten zur Folge hätte. Man kann diesen Bereich auch nicht einfach verschwinden lassen, da die Offsets nicht beliebig verändert werden können. Hier verhält sich die eingeblendete Pipe wie eine Datei. Die Bereiche müssen an die Lesezone weitergereicht werden. Die einzige Möglichkeit besteht darin, die Inhalte zu vernichten, indem der Bereich mit 0 initialisiert wird. Um zu erkennen ob die Inhalte eines Bereiches bereits einmal einem Commit ausgesetzt waren, werden die Speicherseiten des Bereiches mit einem weiteren Attribut versehen. Wird ein `msync()` auf die Seite angewendet, wird das *sync-Flag* gesetzt. Bei der Operation, die eine Speicherseite von einem Bearbeitungszustand in den Lesezustand bringt, wird dieses Flag abgefragt. Ist es nicht gesetzt, wird die Seite initialisiert.

Diese Erweiterung die aus der `munmap()`-Problematik entstanden ist, erweist sich auch für die nächste Anforderung als nützlich.

3.2.4 Standard-I/O und gemischter Betrieb

Analog zu den Dateien soll es möglich sein, auf eine Pipe, die von Prozessen eingeblendet ist, auch mit den Standard-I/O-Funktionen `read()` und `write()` zuzugreifen. Natürlich soll sich für den normalen Zugriff möglichst keine Veränderung des Verhaltens der Pipe einstellen. Ein wesentliche Unterschied, zwischen Pipe und Datei besteht darin, daß bei der Pipe die Lese-/Schreib-Position für alle Prozesse gleich ist. Diese Eigenschaft muß erhalten bleiben.

Der Zugriff auf die eingeblendete Pipe über `read()/write()` wird durch die Verwaltung eines weiteren Offset-Paares ermöglicht. Für den jeweiligen Bereich kommt also noch ein zusätzlicher Offset dazu, der im Prinzip einem der prozessbezogenen Read-/Write-Offsets entspricht. Allerdings ist dieser Wert (Wrw_{off} bzw. Rrw_{off}) den anderen prozessbezogenen Offset-Werten untergeordnet. Das bedeutet, daß er bei der Ermittlung des Bereichsanfanges nur in einigen Sonderfällen mitbestimmend ist.
In Abb.: 16 sind einige Szenarien für eine Schreib-Pipe dargestellt.

- Abb.: 16.A zeigt die Wirkung eines `write()`-Aufrufes. Die Position von Wrw_{off} wrid dabei einfach um die angegebene Anzahl von Zeichen (len) verschoben. Jede Schreib-Operation erfolgt an der aktuellen Position von Wrw_{off}. Unabhängig welcher Prozess das `write()` ausführt.

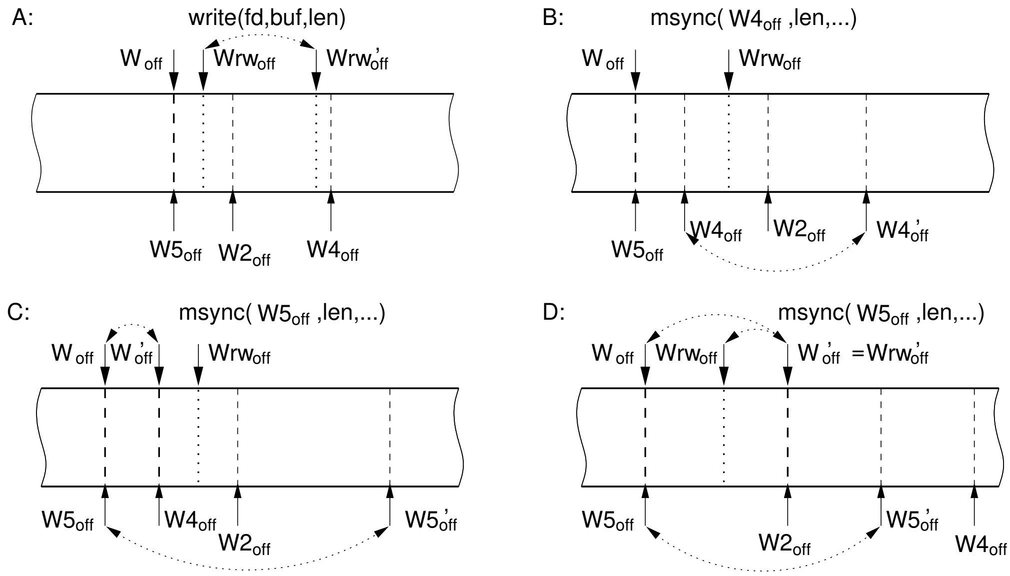

Abbildung 16: Eingeblendete Write-Pipe mit `write()`- und `msync()`-Szenarien

- Ein `msync()` eines Prozesses, der keinen bestimmenden Einfluß auf den Bereichsanfang hat, führt lediglich zur Veränderung des prozessbezogenen Offsets ($W4_{off}$). Wie in Abb.: 16.B zu sehen ist, beeinflußt so ein `msync()` auch die aktuelle Read/Write-Position Wrw_{off} nicht.
- Ein allgemeines `msync()`-Szenario, wie es zuvor beschrieben wurde, ist in Abb.: 16.C zu sehen. Der tatsächliche Commit-Bereich erstreckt sich nur bis zum nächstkleineren Offset $W4_{off}$. Auf Wrw_{off} hat dieses `msync()` ebso wie Abb.: 16.B keinen Einfluß.
- Die Sonderstellung von Wrw_{off} ist bei einem `msync()` wie in Abb.: 16.D zu erkennen. Anders als ein prozessbezogener Offset, stellt Wrw_{off} keine Begrenzung beim Verschieben des Bereichsanfanges dar. Er wird bei der Neuberechnung von W_{off} nicht brücksichtigt und hat keinen Einfluß. Die Read/Write-Position wird ihrerseits von W_{off} mitgezogen. Bedenkt man die Bedeutung eines `msync()`, ist dieses Verhalten auch logisch. Das `msync()` entspricht einem Schreibvorgang in der Pipe. Daher ist es selbstverständlich, daß die Read/Write-Position mitgezogen wird.

Es gibt aber Ausnahmesituationen, wo der Wert von Wrw_{off} ebenfalls die Bereichsgrenze W_{off} beeinflussen kann. In der Situation, in der es keinen Prozess gibt, der die Schreib-Pipe eingeblendet hat, kann die Position W_{off} natürlich nur durch die Read/Write-Position Wrw_{off} bestimmt werden. Auch in Situationen mit bestehenden Einblendungen der Write-Pipe kann Wrw_{off} einen Einfluß auf W_{off} haben.

In Abb.: 17 ist ein Szenarion wie in Abb.: 16.D dargestellt. Jetzt wird aber nicht ein `msync()` sondern ein `munmap()` ab $W5_{off}$ ausgeführt. Als Randbedingungen werden folgende Annahmen über die anderen Prozesse getroffen: $W2_{off}$

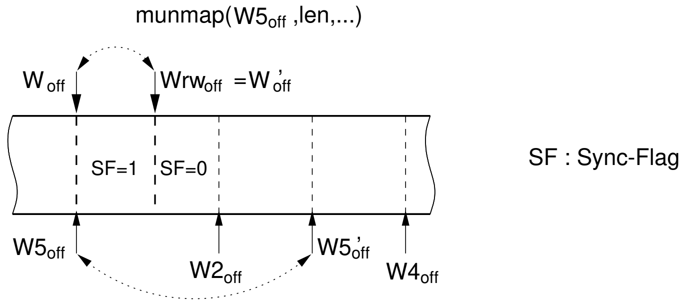

Abbildung 17: `munmap()` mit bestimmender Read-/Write-Position

wurde ebenfalls durch ein `munmap()` an die gegenwärtige Position gelegt. Prozess 4 hat die Pipe mit dem Wert $W4_{off}$ als Offset beim `mmap()` eingeblendet. Damit kann für den Bereich zwischen W_{off} (bzw. $W5_{off}$) und Wrw_{off} gesagt werden, daß das Sync-Flag gesetzt ist (durch eine `write()`-Operation, die Wrw_{off} an die gegenwärtige Position verschoben hat). Für den Bereich zwischen Wrw_{off} und $W2_{off}$ ist das Sync-Flag aber noch nicht gesetzt.
Beim `msync()` in Abb.: 16.D wird für diese Zone auf jeden Fall auch das Sync-Flag gesetzt und der Wert Wrw_{off} wird bis zur Position $W2_{off}$ mitgezogen. In Abb.: 17 soll aber möglichst verhindert werden, daß die bei einem `munmap()` verworfene Inhalte zum Lesen freigegeben werden. Hier besteht noch die Möglichkeit, daß durch ein `write()` auf die Pipe dieser Bereich mit gültigen Daten beschrieben wird. Daher wird der Bereichsanfang jetzt nur bis zur Position von Wrw_{off} nachgezogen.

Eines muß in einer solchen Situation allerdings bedacht werden: Ein nachfolgendes `msync()` ab einer der anderen Positionen ($W2_{off}$ oder $W4_{off}$) muß den Bereichsanfang auf alle Fälle auch über die Zone Wrw_{off} - $W2_{off}$ ziehen. Ansonsten würde ein Commit erst dann möglich werden, wenn tatsächlich ein entsprechender `write()`-Aufruf erfolgt wäre. Davon kann aber nicht ausgegangen werden. Die Inhalte, die noch immer kein Sync-Flag aufweisen, werden entsprechend den angesprochenen Regeln mit 0 initialisiert.

Für die Leseseite einer Pipe gelten die vorgestellten Prinzipien in analoger Weise. Wobei die Situation mit einem `munmap()` ebenso - wie schon zuvor bei der Erweiterung auf mehrere Einblendungen - weniger Probleme macht, als bei der Write-Pipe. Das betrifft jene Bereiche, auf denen ein `munmap()`, aber kein `msync()` ausgeführt wurde. Mit einem `munmap()` auf einer Lese-Pipe wird etwas anderes als auf der Schreib-Seite zum Ausdruck gebracht. Bei einem Schreib-Bereich erklärt der Prozess, daß dieser Bereich verworfen werden soll. Er stellt also eine ganz bestimmte Forderung. Auf der Lese-Seite kann ein `munmap()` in der Form interpretiert werden, daß der Prozess eine neutrale Haltung zu diesem Bereich einnimmt. Es soll kein Einfluß auf den Zustand des Bereiches genommen werden. Vertreten alle Prozesse eine solche Haltung einem Bereich gegenüber, so wird in der Speicherverwaltung eine Entscheidung getroffen. Das ist in diesem Fall die Gleichbehandlung mit Bereichen auf denen ein Commit (`msync()`) erfolgte. Das ist deshalb sinnvoll, da es ansonsten auch hier - wie zuvor auf der Schreib-Seite schon dargestellt - dazu kommen könnte, daß der Bereichsanfang R_{off} nur mehr durch ein `read()` nachgezogen wird.

Die einzige Situation in der sich mitunter eine Änderung des gewohnten Verhaltens ergibt ist dann, wenn aus der Pipe mit `read()` gelesen wird. Im Normalfall können die Daten, die mit `write()` in die Pipe geschrieben werden, sofort mit `read()` ausgelesen werden - auch wenn es sich um eine geringere Anzahl von Zeichen als die Puffer- (Page-) größe handelt. Wird beim Lesen mit `read()` aus einer eingeblendeten Pipe das obere Ende des Lesebereiches erreicht, zeigt die Abbildung des Modells auf die Page-Struktur eine Wirkung. Weitere Daten können erst dann ausgelesen werden, wenn eine ganze Page auf der Schreibseite abgeschlossen wurde.

Die Änderung beim Lesen mit `read()` spielt dann eine Rolle, wenn es um den Transfer sehr kleiner Datenmengen geht. Die entstehenden Probleme entsprechen denen bei gepufferter I/O.

3.2.5 Automatisches Commit

Um den Ablauf des Datentransfers möglichst flüssig zu gestalten, wird ein *Auto-Commit* angeboten. Durch diese Funktion können sich die Prozesse den expliziten Aufruf von `msync()` ersparen. Es kann für jede Einblendung ein eigener Auto-Commit-Wert festgelegt werden. Dieser gibt die Größe des zugegriffenen Bereiches an, bei dessen Überschreitung automatisch ein `msync()` durchgeführt wird. Der zugegriffene Bereich ist jener Bereich, der zwischen der Anfangsadresse der Einblendung und der höchsten, referenzierten Adresse liegt. Durch die Abbildung der Pipe-Verwaltung auf die Page-basierte Speicherverwaltung macht es keinen Sinn, eine Commit-Größe kleiner als eine Speicherseite anzugeben. Die Entscheidung, ob eine Auto-Commit durchgeführt wird, kann immer nur dann gefällt werden, wenn das Betriebssystem die Kontrolle erhält. Das ist immer dann der Fall, wenn der Prozess beim Zugriff auf die eingeblendete Pipe eine Page-Exception auslöst, also immer dann, wenn er eine neue Speicherseite referenziert.

Durch das Auto-Commit wird auch die Gefahr eines Deadlocks durch einen davonlaufenden Schreib-Prozess verhindert. Ein solcher Prozess wird durch die Größe des Fensters, welches den aktuell bearbeiteten Bereich der Pipe angibt, gestoppt. Er blockiert dann in einem Speicherzugriff und kann keine Daten zum Lesen freigeben um den Leseprozess weiterlaufen zu lassen. Dieses Fenster ist jener Bereich, der vom Wert R_{off} bis zum Wert Ref_{max} (höchsten Adresse, die auf Schreibseite referenziert wurde; Werte in Abb.: 14) reicht.

Die Festlegung der Auto-Commit-Größe (je Einblendung) und der realen Gesamtgröße der Pipe (pro Pipe) erfolgt über entsprechende `ioctl()`-Aufrufe. Der Auto-Commit-Wert kann durch jeden Prozess selbst verändert werden.

3.2.6 Das Ende einer Pipe

Am schwierigsten ist die Anpassung des Modells an die Erfordernisse der Page-basierten Speicherverwaltung im Bezug auf das Ende einer Pipe. Es stellen sich hier im Wesentlichen zwei Probleme: wie wird das Ende einer Pipe erkannt und wie kann der verbliebene Inhalt zugänglich gemacht werden?

Im Gegensatz zu einem Prozess, der sich in einem `read()`- oder `write()`-Aufruf befindet, greifen die Prozesse die eine Pipe eingeblendet haben, direkt auf den Speicher zu. Steht ein Inhalt nicht zur Verfügung, wird der Prozess solange blockiert, bis der geforderte Inhalt zugänglich ist. Dabei ist es vorerst unerheblich, ob der Prozess in einem System-Call oder im direkten Speicherzugriff blockiert wird. Ein Unterschied ergibt sich dann, wenn der Zustand nicht mit dem erwarteten Ergebnis beendet werden kann.

Befindet sich ein Prozess in einem (blockierenden) `read()` auf einer Pipe und wird diese vom letzten schreibenden Prozess geschlossen (`close()`), dann wird durch ein Signal (SIGPIPE) der System-Call beendet. Dieser liefert einen Fehlerindikator als Returnwert und die Fehlervariable `errno` wird mit einer Fehlernummer belegt. Der Prozess kann eine ganz normale Fehlerbehandlung durchführen.

Wird der Prozess in einem direkten Speicherzugriff festgehalten, ist die Situation bedeutend schwieriger. Normalerweise bekommt ein Prozess, wenn die gesuchte Speicherseite nicht geliefert werden kann, ein Signal (SIGSEGV, SIGBUS; SIGSEGV stellt hier das Standard-Verhalten dar - Linux verwendet in diesen Fällen aber SIGBUS. Daher werden hier beide Signale angesprochen). Die Default-Reaktion auf dieses Signal ist die Beendigung des Prozesses. Man kann also nicht einfach aus der Referenzierung zurückkehren und eine Fehlerbehandlung durchführen.

Die Lösung besteht darin sicherzustellen, daß der Prozess eine Speicherseite erhält um das SIGSEGV/SIGBUS zu verhindern und eine Fehlerbehandlung im Rahmen eines Signalhandlers am SIGPIPE-Signal durchzuführen. Erstere Maßnahme ist Sache der Implementierung. Der Signalhandler muß auf Applikationsebene erstellt werden. Für seinen Aufbau können nur Hinweise gegeben werden.

Wird eine Pipe auf einer Seite endgültig geschlossen - egal ob Lese- oder Schreib-Pipe - so bekommen die Prozesse die darauf noch zugreifen, ein SIGPIPE. Das gilt für die normale wie für die eingeblendete Pipe.

Wird das Signal an die Prozesse auf der Schreib-Seite ausgeliefert bedeutet das für den Prozess, daß es keinen Sinn mehr macht witerhin etwas in die Pipe zu schreiben. Im Signalhandler sind keine besonderen Aktionen nötig. Die Applikation muß ihre normale Fehlerbehandlung durchführen.

Interessant ist die Situation auf der Lese-Seite. Das Signal kann den Prozessen nur dann zugestellt werden, wenn sie eine neue Seite im Lesebereich anfordern. Dabei ist, je nach Situation, unterschiedlich vorzugehen. Der Zugriff erfolgte auf:

1. eine Seite die noch vor dem Pipe-Ende liegt und nicht die letzte Seite ist. Dem Prozess braucht das Signal noch nicht gesendet werden, da er in der Folge bei weiteren Zugriffen in einen der nächsten Fälle behandelt wird.

2. die letzte Seite und diese ist:

(a) vollständig beschrieben.
Der Prozess bekommt SIGPIPE und die Seite wird retourniert.

(b) nur teilweise voll.
Der Prozess bekommt SIGPIPE, der Inhalt zwischen dem Pipe-Ende und dem Seiten-Ende wird mit 0 initialisiert und die Seite wird retourniert.

3. eine Seite die hinter dem Pipe-Ende liegt.
Der Prozess bekommt SIGPIPE gesendet. Da jetzt keine Seite geliefert werden kann, würde der Prozess in der Folge ein SIGBUS/SIGSEGV bekommen. Um das zu verhindern, wird eine neue, leere Seite retourniert.

Dabei wird davon ausgegangen, daß das Signal SIGPIPE durch einen Signal-Handler abgefangen wird. Die Information, die die Applikation jetzt benötigt, ist die Anzahl der noch in der Pipe verbliebenen Bytes. Dieser Wert kann mit einem `ioctl()`-Aufruf abgefragt werden. Mit Hilfe von PM_GETSTAT (Siehe Kap.: 3.3) kann die Länge bis zum Pipe-Ende in Bezug auf eine angegebene Adresse ermittelt werden. Damit kann der Prozess das Pipe-Ende auch in dem Fall, wo es irgendwo in der letzten Seite liegt (Fall 2(b)), genau bestimmen. Im Abschnitt 5.2 wird an einem Beispiel gezeigt, wie die Zusammenarbeit zwischen Programm und Signal-Handler ausehen könnte.

Die Vorgehensweise in 2(b) und 3 bedarf noch einer Erklärung. In beiden Fällen erhält der Prozess einen Speicherbereich, der noch nicht im Zustand *gültig* ist. Das ist nur denn der Fall, wenn es keine Schreib-Referenz mehr auf die Pipe gibt. Bei einer Pipe ist das akzeptabel, da es keine Möglichkeit mehr gibt sie neuerlich zu öffnen und zu beschreiben. Hier kann sicher gesagt werden, daß das Objekt in der Folge vernichtet wird. Handelt es sich aber um eine FIFO, so ist ein neuerliches Open zum Schreiben sehr wohl möglich. Vor allem im Fall 3 hätte ein Leseprozess bereits eine ganze Speicherseite, die eigentlich erst beschrieben werden sollte.

Dieses Problem kann nur gelöst werden, indem man ein neuerliches Schreib-Open solange verhindert, bis das Objekt durch ein vollständiges Ausblenden durch alle beteiligten Prozesse in den Normal-Modus zurückversetzt wurde. Das entspricht einer Reinitialisierung des Objektes.

Hat ein Prozess mehrere Pipes eingeblendet, so kann er über einen `ioctl()`-Aufruf abfragen, welche Pipe das Signal ausgelöst hat. In der Statusinformation ist ein entsprechendes Flag enthalten.

In der Behandlung des Endes des Datenstroms liegt sicherlich der größte Unterschied zum Umgang mit eingeblendeten Dateien. Allerdings stellt es für die Behandlung einer regulären Datei auch kein Problem dar, wenn ein Signal-Handler an SIGPIPE gebunden wird.

3.3 `ioctl()` für eingeblendete Pipes

Per Definition dient die `ioctl()`-Funktion dem Zweck, alle Aufgaben zu erledigen, die nicht in das I/O-Schema aus `open()`, `close()`, `read()` und `write()` passen. Das erweiterte `mmap()` stellt einige solcher Anforderungen, die am einfachsten durch spezielle `ioctl()`-Aufrufe erledigt werden können.

Der `ioctl()`-Aufruf hat folgende Form:

```
int ioctl(int fd, int request, <type> *argp)
```

wobei `fd` ein gültiger Filedeskriptor sein muß. `request` ist die angesprochene Funktion. Der dritte Parameter gibt die Argumente der betreffenden Funktion an. Es kann sich um Eingabe- (in) oder um Ausgabe- (out) Parameter handeln. Hat man eindeutige Deskriptoren (O_RDONLY, O_WRONLY), kann die Pipe (Read-/Write-Pipe) im `ioctl()`-Aufruf eindeutig identifiziert werden. O_RDWR Deskriptoren machen Probleme beim Ermitteln der Pipe-Offsets.
Folgende Funktionen werden definiert:

- PM_PMODE : Parameter (out) `char *`
  Liefer den Modus in dem sich die Pipe aktuell befindet. Es werden die beiden Konstanten NORM_PIPE und MAP_PIPE definiert und zur Verfügung gestellt.

- PM_BUFOFF: Parameter (out) `unsigned long *`
  Liefert den aktuellen Offset (R_{off}, W_{off}) in der Read- oder Write-Pipe. Welcher Wert geliefert wird, hängt vom angegebenen Deskriptor ab. Die Prozesse brauchen diesen Wert als Parameter beim `mmap()`-Aufruf. Hier ist es notwendig, über eindeutige Deskriptoren zu verfügen.

- PM_GETSTAT: Parameter (in/out) `struct pstat_struct *`
  Mit dieser Funktion können alle wichtigen Daten zu einer Pipe ausgelesen werden. Dazu wird ein eigener Daten-Typ `struct pstat_struct` definiert.

```
        struct pstat_struct {
                unsigned long addr;
                char mode;
                unsigned long sec_rbuf;
                unsigned long tot_rbuf;
                unsigned long bsize;
                unsigned long vmstart;
                unsigned long io_pos;
                unsigned long updsize;
                int sigflag;
                int readers;
                int writers;
        };
```

  - `addr`: Adresse im VMA.
    Da ein Prozess eine Pipe auch öfters eingeblendet haben kann, muß die angesprochene Einblendung eindeutig identifiziert werden. Das erfolgt durch Angabe der Anfangsadresse oder einer beliebige Adresse innerhalb der Einblendung. Zusätzlich kann dadurch auch eindeutig die Art der Einblendung identifiziert werden. Wird auf Basis eines

Read/Write-Deskriptors gearbeitet, kann durch den Filedeskriptor alleine nicht zwischen Read- oder Write-Pipe unterschieden werden. Ist ein reiner Eingabewert. Beim Abruf allgemeiner Informationen über die Pipe kann er auch fehlen (ist 0).

- `mode`: Modus der Pipe.
  Ist der Wert der auch über PM_MODE abgefragt werden kann. Ist die Pipe im Normal-Modus: NORM_PIPE, ist sie eingeblendet: MAP_PIPE.
- `sec_rbuf`: Länge der Pipe (secure).
  Die Anzahl der Bytes in der Pipe, die sicher - also ohne daß der Prozess blockiert - ausgelesen werden können. Wurde ein Wert für `addr` angegeben, so wird die Länge in Bezug auf diesen Wert berechnet. Für `addr` = 0 erhält man die Länge ab R_{off}.
- `tot_rbuf`: Länge der Pipe (total).
  Die Länge der Pipe inclusive des Teils, der in der letzten Page noch nicht erreichbar ist. Bedeutung von `addr` wie bei `sec_rbuf`.
- `bsize` Maximale Größe, die die Pipe erreichen kann. Soviel Speicherplatz wird vom Betriebssystem maximal für die Pipe zur Verfügung gestellt.
- `vmstart`: Aktuelle Startadresse der Einblendung. Wird durch ein Auto-Commit verändert und kann so wieder abgefragt werden.
- `io_pos`: Aktuelle Read- oder Write-Position im Puffer. Wird durch ein Auto-Commit verändert und kann so wieder abgefragt werden.
- `updsize`: Auto-Commit Größe.
  Gibt die maximale Differenz zwischen Bereichsanfang und größter, zugegriffener Adresse an. Wird dieser Wert überschritten, erfolgt automatisch ein `msync()`. Dieser Wert ist einblendungsspezifisch. Er wird nur angezeigt, wenn eine Einblendung durch die Angabe eines Wertes für `addr` identifiziert wurde.
- `sigflag` Zeigt an, daß diese Pipe ein SIGPIPE ausgelöst hat.
- `readers`: Anzahl der Read-Handles.
  Zeigt an, wieviele Read-Handles auf diese Pipe bestehen. Ein Read-Handle wird durch `open()`, `pipe()` oder `mmap()` angelegt.
- `writers`: Anzahl der Write-Handles.
  Zeigt an, wieviele Write-Handles auf diese Pipe bestehen. Write-Handles werden durch `open()`, `pipe()` und `mmap()` angelegt.

Über `addr` wird die Art der Abfrage spezifiziert. Wird dafür 0 angegeben, erhält man die allgemeinen Werte. Für `updsize` kann kein Wert geliefert werden. Wird eine Adresse innerhalb einer Pipe-Einblendung angegeben, so wird eine spezielle Einblendung identifiziert. Damit kann ein Wert für das Auto-Commit ausgelesen werden. Die beiden Längenwerte (`sec_rbuf`, `tot_rbuf`) werden in Bezug auf den angegebenen Adress-Wert berechnet.

Damit kann ein Prozess zu jeder Position in seiner Einblendung die auslesbaren Zeichen ermitteln. Das gilt natürlich nur für eine Einblendung einer Lese-Pipe.

- PM_SETUPDSIZE: Parameter (in) `struct pstat_struct *`
  Setzt die Größe für das Auto-Commit. Jeder Prozess kann für jede Einblendung einer Pipe den Grenzwert für das automatische `msync()` selbst festlegen. Sinnvolle Werte liegen zwischen der Pagesize und der Gesamt-Puffergröße (`bsize`). Die Werte werden in einer Variablen vom Typ `struct pstat_struct` übergeben. Durch `addr` wird die Einblendung identifiziert (beliebige Adresse innerhalb der Einblendung), in `updsize` wird der zu setzende Wert übergeben.

3.4 Zusammenfassung

Die Handhabung einer eingeblendeten Pipe folgt im Wesentlichen den Vorgaben durch das Verhalten einer eingeblendeten Datei. Die charakteristischen Eigenschaften des FIFO-Verhaltens einer Pipe können damit aber nicht vollständig abgedeckt werden. Das führt zu folgenden Unterschieden:

1. Das `mmap()` braucht eine Längenangabe. Dafür ist eine, dem Anwendungsfall angepasste, Annahme zu treffen.

2. Beim Einblenden gibt es einen Minimalwert für den Offset. Dieser kann mit einem `ioctl()`-Aufruf abgefragt werden.

3. Nur ein gewisser Teil der durchgelaufenen Daten werden tatsächlich gepuffert und sind damit zu einem Zeitpunkt zugreifbar.

4. Die Zustandsänderung der Daten muß explizit mit `msync()` vorgenommen werden. Beim Schreiben werden die Daten dadurch zum Lesen freigegeben, beim Lesen werden sie als abgearbeitet erklärt. Nach dem `msync()` ist der angegebene Bereich nicht mehr zugänglich. Dieses Commit kann auch automatisch erfolgen. Der Auto-Commit-Wert kann über einen `ioctl()`-Aufruf gesetzt werden (0 bedeutet kein Auto-Commit).

5. Für die Behandlung des Endes des Datenstroms ist ein Handler an SIGPIPE nötig. Darin kann mit einem `ioctl()`-Aufruf das Pipe-Ende in Bezug auf eine angegebene Adresse ermittelt werden.

6. Zusätzliche Schleife um ein Remap durchzuführen, wenn die Annahme für den Längenwert beim `mmap()` nicht ausreichend war.

Im Abschnit 5.2 wird die praktische Anwendung dieser Änderungen an einem Beispiel gezeigt.

4 Pipe Mapping für Linux

Die Anforderungen die aus dem Modell für das erweiterte `mmap()` entstehen, betreffen wesentliche Teile des zugrundeliegenden Betriebssystems. Aus diesem Grund kann eine beispielhafte Implementierung nur auf einer Plattform erfolgen, zu der auch der Source-Code zur Verfügung steht. Aus den Möglichkeiten, die diese Bedingung erfüllen, wurde Linux gewählt, da es aufgrund seiner Popularität einfach verfügbar und gut dokumentiert ist.

Die Basis für das Projekt ist ein Standard-Kernel der Version 2.0.36 ([Ker00]). Die Tauglichkeit der vorgenommenen Änderungen wurde im Rahmen einer Installation auf Basis einer SuSE 5.3 Distribution getestet. Ein Kernel mit den Änderungen am Speichermanagement läuft auf meinem Rechner in einer typischen Workstation-Installation ohne jegliche Probleme.

Die Möglichkeit, den Kernel zur Laufzeit um nachladbare Kernelmodule zu erweitern, bietet sich im Normalfall für die Entwicklung neuer Kernel--Komponenten an ([Pom99]). Da aber die notwendigen Eingriffe am Speicher-Management sehr tiefgehend waren, hätte die Verwendung von Kernel-Modulen eine weitere Komplexitätssteigerung mit sich gezogen. Darum habe ich darauf verzichtet und den vorhanden Code direkt verändert.

Alle Angaben von Sourcefiles, die im folgenden Text gemacht werden, beziehen sich auf die Struktur der Kernel-Sourcen. Standardmässig befindet sich der Linux-Kernel Code in `/usr/src/linux`. Die Kernelquellen können aber auch an beliebiger anderer Stelle liegen. Daher werden alle Pfadangaben von Soure-Dateien relativ in Bezug auf das Basisverzeichnis `..../linux` angegeben.
Folgende Liste gibt alle Source-Dateien an, in denen Änderungen vorgenommen wurden:

C-Files:
```
arch/i386/kernel/ptrace.c
fs/buffer.c
fs/exec.c
fs/fifo.c
fs/inode.c
fs/pipe.c
fs/proc/array.c
ipc/shm.c
kernel/exit.c
kernel/fork.c
mm/filemap.c
mm/memory.c
mm/mlock.c
mm/mmap.c
mm/mprotect.c
mm/mremap.c
```
Header-Files:
```
include/asm-i386/pgtable.h
include/asm-i386/mman.h
```

```
include/linux/mm.h
include/linux/mman.h
include/linux/pipe_fs_i.h
include/linux/version.h
```

4.1 Speichermanagement unter Linux

Das Speichermanagement von Linux basiert auf einem *architekturunabhängigem Speichermodell* ([BBD+97], [Tor94], [Rus99], [Joh97]). Dieses Speichermodell bedient sich einer abstrakten Schnittstelle zur darunterliegenden Hardware. Diese Schnittstelle folgt den Regeln einer Speicherverwaltung auf Page-Basis mit Hardwareunterstützung durch eine MMU (Memory Management Unit), wobei angenommen wird, daß bei der Adressübersetzung ein TLB (Translation Look-aside Buffer) zum Einsatz kommt.

Durch diese Trennung in eine hardwareabhängige Ebene und die abstrakte Ebene des Linux-Speichermodells, wird die Portierung auf neue Plattformen erleichtert.

Der virtuelle Adressraum wird in ein Kernel- und ein User-Segment geteilt. Im Kernelsegment liegt der Kernel-Code und die Kernel-Daten, im User-Segment liegt der Programm-Code und die Nutzer-Daten. Eine virtuelle Adresse besteht daher aus einem *Segmentselektor* und dem *Offset* innerhalb des Segmentes. Innerhalb der Segmente wird mit Adressen operiert, die nur aus den Offsets bestehen. Lediglich der Kernel muß den Segmentselektor manipulieren können, um auch auf Daten im Nutzer-Segment zugreifen zu können.

Die virtuellen Adressen (Segmentselektor + Offset) müssen in *lineare* Adressen übersetzt werden. Die linearen Adressen ihrerseits werden endgültig auf *physikalische* Adressen abgebildet. Diese Adressübersetzung wird durch die MMU erledigt. Auf Systemen die keine Segmentierung erlauben, entsprechen die virtuellen, den linearen Adressen. Die Segmente werden durch Offsets im linearen Adressbereich nachgebildet.

Auf 32-bit Systemen kann ein linearer Adressraum von 4 GByte angesprochen werden. Linux auf der Intel x86 Plattform teilt den linearen Adressraum in ein Kernelsegment von 1 GByte und das Nutzersegment mit 3 GByte auf. Neuere Kernelversionen (der Serie 2.2.x) unterstützen auch User-Space-Größen von 4 GByte.

Im architekturunabhängigen Speichermodell ist ein dreistufiger Übersetzungsvorgang für die Abbildung der linearen auf die physikalischen Adressen vorgesehen (Abb.: 18).

Die dreistufige Übersetzung wurde bereits unter Berücksichtigung der möglicherweise großen Anzahl von Pages auf 64-bit Architekturen gewählt. So ist sichergestellt, daß sich die Größe der Pagetabellen in Grenzen hält. Die MMU der i386-Plattform unterstützt allerdings nur eine zweistufige Übersetzung ([V.89]). Um die 3. Stufe an der abstrakten Schnittstelle anbieten zu können, wird diese im hardwareabhängigen Layer für die i386-Architektur simuliert. Dabei wird ein Trick verwendet: Das Mittlere Page-Directory hat eine Größe von 1. Die Details der Adressübersetzung folgen der allgemeinen Methode, wie sie in der

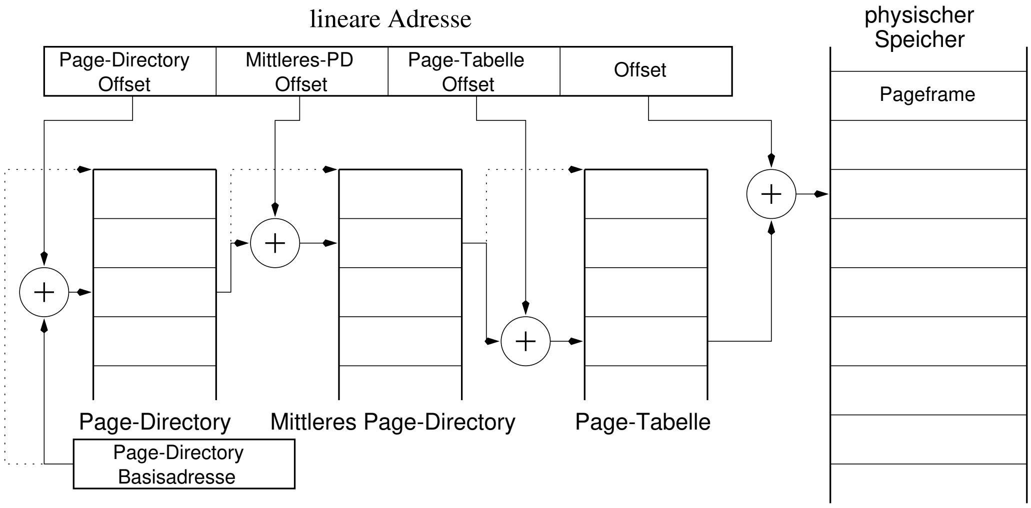

Abbildung 18: 3-stufige Adressübersetzung unter Linux

Einführung bereits dargestellt wurde.

Das Linux-Speichermanagement stellt eine Vielzahl von Funktionen zur Manipulation von Page-Directorys, Mittleren Page-Directory, Page-Tabellen und den jeweiligen Einträgen zur Verfügung. Die Definitionen dieser Funktionen und Macros finden sich - architekturabhängig - in den Dateien `include/asm/page.h` und `include/asm/pgtable.h`.

Jeder Prozess hat seinen eigenen, privaten, virtuellen Adressraum. Daher unterscheiden sich die Page-Directories oder zumindest gewisse Teile der Page-Tabellen der verschiedenen Prozesse. Wird ein Prozess erzeugt (`fork()`, `exec()`, so erhält der neue Prozess eine eigene Kopie der Page-Directories und -Tabellen. Eine Ausnahme bilden die Pagetabellen des Kernelsegments, diese werden von allen Prozessen geteilt.

Die Prozesse werden in der Task-Liste verwaltet. Die Einträge dieser Liste sind vom Typ `struct task_struct` (`include/linux/sched.h`)) und beschreiben einen Task (Prozess) eindeutig. Das grundlegende Layout des Adressraumes wird durch die Task-Komponente `mm` dargestellt. Diese Datenstruktur ist vom Typ `struct mm_struct` (`include/linux/sched.h`).

Alles was ein Prozess in seinem Adressraum referenzieren kann, wird einem *virtuellen Speicherbereich* (Virtual Memory Area - VMA) zugeordnet. Der Adressraum eines Prozesses kann als die Menge seiner VMAs aufgefasst werden. Dieses abstrakte Konzept erinnert an das Speichermodell von SVR4. Ein VMA wird durch den Datentyp `struct vm_area_struct` (`include/linux/mm.h`) dargestellt. Die Menge der VMAs eines Prozesses wird in der Memory-Management Struktur `mm` auf zwei Arten erfasst. `mmap` ist der Einstiegspunkt in die einfach verkettete Liste der VMAs. Darüberhinaus ist diese Menge zur effizienteren Bearbeitung auch noch in Form eines AVL-Baumes verfügbar. `mmap_avl` ist die Wurzel dieses Baumes.

Der Darstellung durch VMAs liegt ein objektorientiertes Konzept zugrunde.

Die Datenstruktur `struct vm_area_struct` fasst die Attribute des Objekts zusammen. Die Methoden auf dem Objekt werden in Form eines Operationsvektors `struct vm_operation_struct` (`include/linux/mm.h`) dargestellt. Diese Darstellungsform ist elementar im Design von Linux. Alle wesentlichen Konzepte - wie `inode`, `file`, usw - werden auf diese Art repräsentiert.

Diese VMAs spielen eine zentrale Rolle im Memory Management. Jeder VMA wird duch einen Aufruf von `do_mmap()` (`mm/mmap.h`) eingerichtet (Ausnahme sind jene VMAs die Einblendungen von *SysV Shared Memory* darstellen; diese werden mit `sys_shmat()` - `ipc/shm.c` erzeugt). Im Operationsvektor für VMAs werden Funktionen für alle wesentlichen Aufgaben des Memory Managements definiert.

- `open()`: Wird aufgerufen, wenn ein VMA neu eingerichtet wird.
- `close()`: Fasst die spezifischen Aufgaben beim Entfernen eines VMAs zusammen.
- `unmap()`: Aufgaben bei der teilweisen Ausblendung des Bereiches. Bei einer vollständigen Ausblendung wird die `close()`-Komponente auch noch ausgeführt.
- `protect()`: Implementiert die Konsequenzen einer Änderung der Zugriffsrechte am VMA.
- `sync()`: Führt im Falle eines File-Mappings eine Synchronisation des Speicherinhaltes mit dem Platteninhalt durch. Für alle Objekte ohne Backing Store undefiniert. Wird beim Pipe-Mapping für das Commit verwendet.
- `nopage()`: Im Falle eines Page-Faults hat diese Funktion die Aufgabe einen Inhalt für die referenzierte Adresse zu liefern.
- `swapout()`: Wird eine Speicherseite des VMAs ausgelagert, können objektspezifische Maßnahmen erforderlich sein. Insbesondere wenn es sich um ein Shared Objekt handelt, ist diese Komponente von Bedeutung.
- `swapin()`: Aktionen, die beim Zurückladen einer Page des Bereiches in den Hauptspeicher erforderlich werden.

Das Memory Management besteht aus Funktionen, welche die allgemeinen Teile dieser Aufgaben erfüllen. Die speziellen Aspekte, die vom konkret betroffenen Objekt abhängen, werden einfach dadurch berücksichtigt, indem die objektspezifische Funktion aus obigem Operationsvektor aufgerufen wird. Damit entfällt die Notwendigkeit von komplizierten Programmstrukturen, in denen Fallunterscheidungen getroffen werden müssten.

Auf dieser Basis lassen sich alle Formen von Memory-Mapping und Memory-Sharing, wie sie für ein modernes und effizientes Memmory Management notwendig sind, realisieren.

Die zentrale Rolle spielt dabei das *Exception Handling*. Jede Einblendung eines Objekts in den Adressraum stellt eine Art von Speicheranforderung dar.

Um nicht durch die Größe des vorhandenen virtuellen Speichers (Summe aus Hauptspeicher und Speicher im Auslagerungsbereich) beschränkt zu sein, werden Anforderungen an physischen Speicher erst dann erfüllt, wenn tatsächlich darauf zugegriffen wird. Daher wird grundsätzlich jede Anforderung nach Speicher - in welcher Form auch immer - erfüllt. So kann ein Prozess eine Datei, deren Größe die des vorhandenen virtuellen Speichers bei weitem übersteigt, trotzdem in den Adressraum einblenden. Zu diesem Zeitpunkt wird lediglich ein entsprechender VMA erzeugt, aber noch kein einziger Pageframe dafür belegt. Erst wenn der Prozess auf eine Adresse in diesem Bereich zugreift, wird ein Pageframe mit dem entsprechenden Inhalt aus der Datei beschrieben und an der entsprechenden Stelle eingeblendet. Speicher wird also immer *on demand* zur Verfügung gestellt.

Anhand dieses Beispiels lässt sich der prinzipielle Ablauf des *Exception Handlings* darstellen. Greift der Prozess auf besagte Adresse in diesem VMA zu, wird durch die MMU festgestellt, daß es für diese Adresse keinen gültigen PTE (Page-Table Entry) gibt. Der Inhalt befindet sich also nicht im Hauptspeicher. Der Prozessor löst eine *Page Exception* aus und übergibt die Kontrolle an den *Page-Fault Handler* des Betriebssystems (genauso reagiert die MMU wenn die Zugriffsrechte eines Pageframes verletzt werden). Der Page-Fault Handler (`do_page_fault()` für Intel in `arch/i386/mm/fault.c`) nimmt eine erste Analyse des Problems vor. Da unter Linux per Definition Kernel-Pages nicht auslagerbar sind, darf der Page-Fault keinesfalls eine solche betreffen. Dies kann nur im Falle eines elementaren Fehlers auftreten. Die adequate Reaktion darauf besteht in einer *kernel panic*. Liegt die Adresse in keinem VMA des User-Space, so liegt eine allgemeiner Zugriffsfehler vor. Der Prozess wird mit SIGBUS (Linux verwendet *nicht* SIGSEGV) beendet. Kann ein VMA ermittelt werden, wird nur nach zwei Ursachen untersucht: ist der Adresse noch kein Pageframe zugeordnet, so wird zur weiteren Behandlung die Funktion `do_no_page()` (`mm/memory.c`) aufgerufen. Ein Verstoß gegen die Zugriffsrechte (zb wegen Copy on Write - CoW) wird von der Funktion `do_wp_page()` (`mm/memory.c`) weiterbehandelt.

Hier ist auch die genaue Funktionsweise von CoW und damit von Private Mapping ersichtlich. Private Mappings werden über Copy on Write realisiert. Änderungen in solchen Seiten sind solange für alle beteiligten Prozesse sichtbar, bis der Privat-Mapping Prozess das erste Mal schreibend auf die Seite zugreift. Genau dann wird eine neue, private Kopie der Seite erzeugt. Im Falle einer Protection-Verletzung wird keine opjektspezifische Funktion aufgerufen.

Nachdem beim Einrichten eines VMAs noch in keiner Weise Speicher reserviert wird, führt ein Zugriff darauf sicher an diese Stelle. `do_no_no_page()` spielt also die zentrale Rolle beim Einlesen der Inhalte in den Speicher. In der Funktion selbst, werden die Anforderungen betreffend *Anonymer Mappings* erledigt. Solche Einblendungen haben kein Backing Store. Sie werden alle vorerst mit einem speziellen Pageframe - der ZERO_PAGE - initialisiert und als CoW markiert. Erst beim ersten schreibenden Zugriff wird ein echter Pageframe an die Stelle eingeblendet. Ein möglicher Grund, warum an der betroffenen Stelle kein Pageframe gefunden werden konnte ist der, daß der Inhalt ausgelagert wurde. Zum Einlesen einer Seite aus dem Auslagerungsbereich wird die Funktion `do_swap_page()` (`mm/memory.c`) verwendet. In allen anderen Fällen wird die objektspezifische Komponente `nopage()` aus dem Operationsvektor des VMAs aufgerufen. Diese Funktion liefert die Adresse eines Pageframes mit dem gesuchten Inhalt. Dieser wird an der betroffenen Stelle in den Speicher eingeblendet und damit ein gültiger PTE erzeugt. Der Prozess kann an der Stelle an der unterbrochen wurde, fortgesetzt werden.

4.1.1 Speichermanagement im Kernel-Segment

Anders als im User-Segment, steht im Kernel-Segment virtueller Speicher nur in stark eingeschränkter Form zur Verfügung. Der physische Speicher wird ab Adresse 0 auf den Kernel-Adressraum abgebildet. Der Bereich zwischen der höchsten physischen Adresse und dem oberen Ende des Kernel-Adressraumes (bei i386 ist das 1 GByte) steht für virtuelle Speicherbereich zur Verfügung. Die einzige Möglichkeit, einen virtuellen Speicherbereich im Kernel-Segment anzulegen, besteht in der Verwendung von `vmalloc()` (`mm/vmalloc.c`). Diese Funktion dient dazu, größere Mengen an Speicher im Kernel anzufordern. Für den Bedarf an Speicherbereichen kleiner als PAGE_SIZE, steht die Funktion `kmalloc()` (`mm/kmalloc.c`) zur Verfügung. Da Kernel-Pages per Definition nicht auslagerbar sind, wird Speicher der mit `vmalloc()` angefordert wurde, tatsächlich sofort belegt. Im Kernel besteht keine Möglichkeit, Speicher *on demand* in Anspruch zu nehmen. Diese Einschränkung ist leider mit den Anforderungen für das Pipe-Mapping nicht vereinbar, da die Speicherseiten des Pipe-Buffers auslagerbar sein müssen, zugleich aber auch vom Kernel, im Rahmen der Standard-I/O-Funktionen zugegriffen werden.

4.2 Anpassung des Linux-VMM

Für die Realisierung des erweiterten `mmap()` wurden einige Bedingungen genannt, die das Memory-Management erfüllen muß. Da diese Bedingungen von Linux nicht erfüllt werden, müssen entsprechende Änderungen und Erweiterungen an der Speicherverwaltung vorgenommen werden. Dabei handelt es sich um die Erweiterung der Verwaltung auf *Byte-Granularität* und die Änderung der Handhabung der Bereichsgrenzen im Sinne eines *minimierenden Verhaltens*.

4.2.1 Byte-Granularität

Die Verwaltung der VMAs muß, so wie in Kap. 2.2.1 dargestellt, auf die Basis von beliebigen Offsets, erweitert werden. Die Darstellung von VMAs erfolgt mit Hilfe der Datenstruktur `struct vm_area_struct` (`include/linux/mm.h`).

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        pgprot_t vm_page_prot;
        unsigned short vm_flags;
        short vm_avl_height;
```

```
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;
        struct vm_area_struct * vm_next;
        struct vm_area_struct * vm_next_share;
        struct vm_area_struct * vm_prev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_offset;
        struct inode * vm_inode;
        unsigned long vm_pte;
};
```

Es wird hier nur auf jene Komponenten näher eingegangen, die in diesem Kontext von Interesse sind. Das sind die beiden Werte, die die Grenzen des dargestellten Bereiches beschreiben: `vm_start` und `vm_end`. Diese beiden Werte bilden auch weiterhin die bestimmenden Grenzen für die Verwaltung des VMAs. Um beliebige Offsets auch erfassen zu können, wird diese Struktur um zwei Komponenten erweitert: `vm_rstart` und `vm_rend`.

Bei Erweiterungen dieser Struktur muß man allerdings vorsichtig sein, da an einigen Stellen im Code eine direkte Initialisierung einer Variablen dieses Typs vorgenommen wird (= a,b,....). Bei Änderungen der Datenstruktur müssen auch die Initialisierungen angepasst werden. Das stellt eine potentielle Fehlerquelle dar. Der saubere Weg würde eine Initialisierung der Komponenten über ihre symbolischen Referenzen beinhalten. Die beiden neuen Komponenten wurden unmittelbar hinter ihrem pagealigned Partner in die Struktur eingefügt:

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_rstart;
        unsigned long vm_end;
        unsigned long vm_rend;
        ...........
```

Die entscheidenden Punkte für die Mitverwaltung beliebiger Offsets finden sich in der Parameter-Aufbereitung der Funktionen zur Berarbeitung von VMAs. Allenvoran ist das `do_mmap()` in `mm/mmap.c`. In derselben Datei findet sich auch der Code der Unmap-Funktion `do_munmap()`. Weiters sind davon die Funktionen `sys_msync()` (`mm/filemap.c`), `sys_mremap()` (`mm/mremap.c`), `sys_mprotect()` (`mm/mprotect.c`), `sys_mlock()` und `sys_munlock()`, beide in (`mm/mlock.c`), betroffen. In diesen Funktionen mußten die Rundungen der Adressen und die Tests auf das Pagealignemant der Werte entfernt bzw. angepasst werden.

Aber auch die Aufbereitung des VMAs in `do_shmat()` wurde genauso wie in `setup_arg_pages()` (`fd/exec.c`) um die Initialisierungen der beiden neuen Komponenten erweitert. Darüberhinaus war es auch nötig, die Rundung des Adresswertes in `do_no_page()` zu entfernen. Einen Einfluß hat die Verwaltung von beliebigen Bereichsgrenzen auf die Funktion `merge_segments()`

(mm/mmap.c), die benachbarte VMAs mit gleichen Eigenschaften zu einem gemeinsamen Bereich zusammenfasst. Ein weiterer Test in dieser Funktion berücksichtigt auch die neuen Anfangswerte der VMAs.

Der Returnwert von mmap() wird durch den angegebenen Offset im Objekt bestimmt. Da der Offset beim normalen mmap() auf einer Page-Grenze zu liegen hat, ist auch die retournierte Adresse page-alligned. Analog dazu richtet sich der Returnwert auch beim erweiterten mmap() nach dem angegebenen Offset. Es ist daher nicht der Wert von vm_start, sondern der von vm_rstart der als Anfangsadresse des VMAs geliefert wird.

Die Darstellung der Memory-Map eines Prozesses im proc-Dateisystem (/proc/*PID*/maps) wurde um die Anzeige der realen Bereichsgrenzen erweitert (Änderungen in fs/proc/array.c). Dabei wurde ein Fehler in der Offset-Berechnung für den Stack-VMA in arch/i386/kernel/ptrace.c auch behoben.

4.2.2 Minimierendes Verhalten

Die Maßnahmen für die Implementierung des *Minimierenden* Verhaltens, wie es in Kap. 2.2.2 definiert wurde, gehen Hand in Hand mit den Änderungen zur Realisierung der Byte-Granularität. Betroffen ist der Code zur Aufbereitung der Bereichs-Grenzen in den Funktionen für die Manipulation von VMAs.

Stellvertretend für alle betroffenen Funktionen werden die Änderungen für den Aufruf der Funktion do_munmap().
Der ursprüngliche Code der Funktion hatte folgendes Aussehen:

```
int do_munmap(unsigned long addr, size_t len)
{
     struct vm_area_struct *mpnt, *prev, *next, **npp, *free;

     if ((addr & ~PAGE_MASK) || addr > MAX_USER_ADDR || len > MAX_USER_ADDR-addr)
             return -EINVAL;
     if ((len = PAGE_ALIGN(len)) == 0)
             return 0;
     mpnt = find_vma(current->mm, addr);
     if (!mpnt)
             return 0;
     avl_neighbours(mpnt, current->mm->mmap_avl, &prev, &next);
     npp = (prev ? &prev->vm_next : &current->mm->mmap);
     free = NULL;
     for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
             *npp = mpnt->vm_next;
             mpnt->vm_next = free;
             free = mpnt;
             avl_remove(mpnt, &current->mm->mmap_avl);
     }
     if (free == NULL)
            return 0;
```

```
    do {
            unsigned long st, end;

            mpnt = free;
            free = free->vm_next;
            remove_shared_vm_struct(mpnt);
            st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
            end = addr+len;
            end = end > mpnt->vm_end ? mpnt->vm_end : end;
            if (mpnt->vm_ops && mpnt->vm_ops->unmap)
                    mpnt->vm_ops->unmap(mpnt, st, end-st);
            zap_page_range(current->mm, st, end-st);
            unmap_fixup(mpnt, st, end-st);
            kfree(mpnt);
    } while (free);
    return 0;
}
```

Schon in der ersten Abfrage wird abgebrochen, wenn der angegebene Wert für `addr` nicht auf einer Seitengrenze liegt. In der nächsten Anweisung wird das Bereichsende durch das Macro `PAGE_ALIGN(len)` auf die nächsthöhere Pagegrenze aufgerundet (maximierendes Verhalten). Die eigentliche `unmap`-Arbeit wird durch die beiden Funktionen `zap_page_range()` und `unmap_fixup()` erledigt.

Diese Funktion hat nach den Änderungen für Byte-Granularität und minimierendem Verhalten folgendes Aussehen:

```
int do_munmap(unsigned long addr, size_t len)
{
    struct vm_area_struct *mpnt, *prev, *next, **npp, *free;
    struct vm_area_struct *first;
    unsigned long raddr = addr,
                  rlen = len;

    addr &= PAGE_MASK;
    len = PAGE_ALIGN(len + (raddr & ~PAGE_MASK));

    if (addr > MAX_USER_ADDR || len > MAX_USER_ADDR-addr)
            return -EINVAL;
    if ((len = PAGE_ALIGN(len)) == 0)
            return 0;
    mpnt = find_vma(current->mm, addr);
    if (!mpnt)
            return 0;
    free = find_vma(current->mm, raddr + rlen);
    if (!free)
            return 0;
```

```
if ((((raddr & ~PAGE_MASK) + rlen) < PAGE_SIZE) &&
    (raddr > mpnt->vm_rstart) && (raddr+rlen < free->vm_rend)) {
        printk("[%d] unmap : invalid area start %lx len %lx\n",
               current->pid,raddr,rlen);
        return -EINVAL;
}
first = mpnt;
avl_neighbours(mpnt, current->mm->mmap_avl, &prev, &next);
npp = (prev ? &prev->vm_next : &current->mm->mmap);
free = NULL;
for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
        *npp = mpnt->vm_next;
        mpnt->vm_next = free;
        free = mpnt;
        avl_remove(mpnt, &current->mm->mmap_avl);
}
if (free == NULL)
        return 0;
do {
        unsigned long st, end;
        unsigned long rst, rend;

        mpnt = free;
        free = free->vm_next;
        remove_shared_vm_struct(mpnt);
        /* Wenn die reale Adresse kleiner ist als der VMA-Start, kann
         * dieser vom Anfang an ausgeblendet werden. Ansonsten handelt
         * es sich um den Anfangsbereich. Hier muss der reale Anfang
         * beruecksichtigt werden. Ist die Adresse kleiner oder gleich
         * diesem, kann von Anfang weg ausgeblendet werden. Ansonst
         * muss das konservative Verhalten gezeigt werden. Die Seite
         * in der die Adresse liegt darf nicht ausgeblendet werden.
         * Daher die Trennung in rst und st. */
        st = (raddr <= mpnt->vm_rstart) ? mpnt->vm_start :\
                     ((raddr - addr) ? addr + PAGE_SIZE : addr);
       /* Am anderen Bereichsende gilt dasselbe : die Seite in der
         * das Ende des auszublendenden Bereiches, liegt darf nicht
         * ausgeblendet werden. */
        end = ((raddr + rlen) >= mpnt->vm_rend) ? \
                     mpnt->vm_end : (((addr+len)-(raddr+rlen)) ?\
                     (addr + len - PAGE_SIZE) : (addr + len));
        rst = (raddr <= mpnt->vm_rstart) ? mpnt->vm_start : raddr;
        rend = ((raddr + rlen) >= mpnt->vm_rend) ? mpnt->vm_end :\
                                                   (raddr+rlen);
         /* Dieser Fall sollte nie auftreten! */
        if (end < st) {
                printk("munmap : error : end < st (%lx <= %lx)\n",
```

```
                        end,st);
                printk("addr0x%lx, len0x%lx, raddr0x%lx, rlen0x%lx\n",
                        addr,(unsigned long)len,raddr,rlen);
                return -EINVAL;
        }
        /* Aufruf der unmap()-Funktion muss mit den realen Bereichs-
         * Werten erfolgen! Ist weiter kein Problem, da es z.Z. nur
         * zwei solcher Funktionen gibt: fuer das allgemeine File-
         * mmap in mm/filemap.c; wird sofort auf ein sync weiter-
         * geleitet, in dem die Adress-Auswertung nocheinmal enthalten
         * ist. In fs/pipe.c: hier muessen auf alle Faelle die
         * echten Offset-Werte verfuegbar sein ==>> rst und rend-rst */
        if (mpnt->vm_ops && mpnt->vm_ops->unmap)
                mpnt->vm_ops->unmap(mpnt, rst, rend-rst,current);
        zap_page_range(current->mm, st, end-st);
        unmap_fixup(mpnt, rst, rend-rst);
        kfree(mpnt);
} while (free);
return 0;
}
```

Die Prüfung, ob der Wert von `addr` auf eine Pagegrenze fällt, wurde entfernt. Die Werte für `addr` und `len` werden sofort auf die umschließenden Pagegrenzen gezogen. Die Original-Werte werden in den neuen Variablen `raddr` und `rlen` abgelegt.

Auch hier wird die aufgerundete, obere Bereichsgrenze (`PAGE_ALIGN(len)`) gebraucht. Allerdings nur zur Bestimmung der betroffenen Bereiche. Das minimierende Verhalten wird durch die Übergabe der realen Grenzwerte an die Funktion `unmap_fixup()` berücksichtigt.

Da sich beim minimierenden Verhalten der angegebene Bereich immer reduziert, muß schon früher ein Test mit dem VMA in dem der Endpunkt des `unmap`-Bereiches liegt, erfolgen. `mpnt` verweist auf den Anfangs-VMA, `free` auf den VMA des Endpunktes (die Bereichsangabe beim Aufruf kann sich auch übere mehrere VMAs erstrecken).

Der nächste Test prüft, ob bei einer Bereichsangabe innerhalb einer Speicherseite, eine der Grenzen mit einer Grenze des VMAs zusammenfällt (Vgl. Kap. 2.2.2). Wenn dies nicht der Fall ist würde versucht werden, eine Teilung eines VMAs in der Form vorzunehmen, daß danach eine Seite in zwei verschiedenen VMAs enthalten wäre. Daher wird ein Fehler angezeigt. Dieser Test stellt den Einfluß des minimierenden Verhaltens auf dieser Ebene dar. Weitere Auswertungen gemäß minimierendem Verhalten erfolgen in den, von hier aus aufgerufenen Funktionen `zap_page_range()` und `unmap_fixup()`.

Die folgenden Operationen liefern eine Liste der betroffenen VMAs. Bei diesem Vorgang wurde nichts verändert. Die folgende `do`-Schleife wendet die `unmap`-Operation auf alle VMAs an. Zum besseren Verständnis wurde der Kommentar im neuen Programmsegment belassen. Die Regeln die hier für die Bereichsgrenzen angewendet werden, wurden in Kap. 2.2.2 erläutert.

Die Anpassungen, die in den anderen Funktionen (`sys_msync()`, `sys_mremap()`, `sys_mprotect()`, `sys_mlock()` und `sys_munlock()`) vorgenommen wurden, folgen dem hier angewendeten Schema.

Erfreulicherweise zeigten sich keinerlei negative Auswirkungen durch die vorgenommenen Änderungen auf bestehende Applikationen. Das Test-System (Workstation mit graphischer Oberfläche im täglichen Einsatz) zeigte keine Anzeichen von Systemstörungen. Allerdings wurde noch keine Erweiterung des VMM des Kernelsegments vorgenommen, da dies einen sehr tiefgehenden Eingriff in das Linux-System-Design bedeuten würde. Andererseits kann im Rahmen einer beispielhaften Implementierung des Pipe-Mappings auf die Swapping-Komponente durchaus verzichtet werden.

4.3 Pipe-Darstellung im Linux-Kernel

In Unix sind die Daten in einem hierarchischen Dateisystem untergebracht, das Dateien unterschiedlichsten Types enthält. Dies umfasst nicht nur reguläre Dateien und Verzeichnisse, sondern auch Gerätedateien (`/dev`), FIFOs, symbolische Links und Sockets. Es kann dadurch auf alle Ressourcen des Systems über Dateien zugegriffen werden.

Zur internen Darstellung von Dateien im Kernel wird die *Inode*-Struktur verwendet ([BBD+97]). Sie enthält Informationen wie den Eigentümer, die Zugriffsrechte oder die Zuordnung der Datei zu einem Gerät. In Linux dient die Datenstruktur `struct inode` (`include/linux/fs.h`) zur Darstellung eines Inodes. Die Sicht eines Prozesses auf eine Datei wird durch eine *File*-Struktur beschrieben. Zu dieser Sicht auf eine Datei gehören Attribute, wie etwa der Modus, in dem die Datei geöffnet wurde (O_RDONLY, O_WRONLY, O_RDWR), oder die aktuelle Position der nächsten I/O-Operation. Im Kernel wird dafür der Datentyp `struct file` (`include/linux/fs.h`) verwendet.

Wie schon bei der Darstellung der VMAs gezeigt wurde, erfolgt auch hier die Beschreibung auf eine objektorientierte Art. Die genannten Datenstrukturen fassen die Attribute der Objekte zusammen. Die Methoden liegen in Form eines Operationsvektors vor. Dazu sind die Typen `struct inode_operations` und `struct file_operations` (beide in `include/linux/fs.h`) definiert.

Da FIFOs Objekte im Dateisystem sind ist es klar, daß sie in Form von Inodes repräsentiert werden. Pipes selbst sind zwar keine Objekte des Dateisystems, werden aber genauso wie Dateien verwendet. Nachdem FIFOs nichts anderes als Pipes mit Repräsentation im Dateisystem sind, werden sie auch intern gleich behandelt. Die Darstellung erfolgt in beiden Fällen als Inode.

Da Inodes die unterschiedlichsten Arten von Dateien beschreiben, haben sie auch eine objektspezifische Attribut-Komponente. Dieses Element `u` ist in Form einer `union` implementiert. Beschreibt der Inode eine FIFO/Pipe, so ist `u` eine Struktur vom Typ `struct pipe_inode_info` (`include/linux/pipe_fs_i.h`).

```
struct pipe_inode_info {
    struct wait_queue * wait;     /* Die Warteschlange der Pipe */
    char * base;            /* Basisadresse der Ringpuffer-Page */
    unsigned int start;  /* Anfang des Inhaltes (READ-Position) */
```

```
    unsigned int len;  /* Laenge und damit Ende (WRITE-Position */
    unsigned int lock;                          /* Sperre der Pipe */
    unsigned int rd_openers;       /* Anzahl der Leser-Prozesse */
    unsigned int wr_openers;   /* Anzahl der Schreiber-Prozesse */
    unsigned int readers;          /* Aktuelle Anzahl der Leser */
    unsigned int writers;      /* Aktuelle Anzahl der Schreiber */
};
```

Diese Datenstruktur beschreibt einen Ringpuffer mit der Größe von einer Speicherseite (Abb.: 19). In diesem Puffer werden zwei Positionen verwaltet: die aktuelle Lese- und Schreib-Position. Erstere ist direkt durch `start` dargestellt, die zweite wird aus `start` und `len` berechnet.

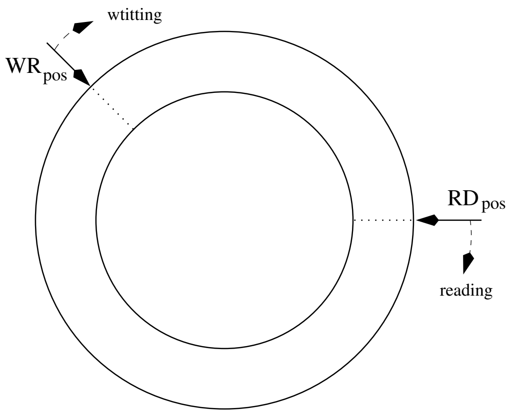

Abbildung 19: Ringpuffer einer normalen Pipe

Der System-Call `pipe()` erzeugt über die Funktion `do_pipe()` (`fs/pipe.c`) eine Pipe. Dazu wird ein temporärer Inode eingerichtet (`get_pipe_inode()` in `fs/inode.c`) und mit `get_empty_filp()` (`fs/file_table.c`) werden zwei Filedeskriptoren, je einer für die Lese- und die Schreib-Pipe, erzeugt.

Das Äquivalent bei den FIFOs ist ein `open()`-Aufruf. Dabei werden mit Hilfe der Funktion `fifo_open()` (`fs/fifo.c`) die zuvor erzeugten Inode- und File-Strukturen entsprechend den Pipe-Erfordernissen initialisiert. Weiters werden die Operationsvektoren von File und Inode FIFO-spezifisch belegt. Die Vektoren die dabei verwendet werden, unterscheiden sich nur in der `select()`-Komponente. Der Grund dafür liegt im Unterschied zwischen dem normalen Select und der Art, wie das unter SunOS implementiert wurde. Aus Kompatibilitätsgründen sind beide Varianten implementiert.

In der Folge unterscheidet die internen Verwaltung nicht mehr zwischen Pipe und FIFO. Beide verwenden dieselben Lese- und Schreib-Operationen (`pipe_read()`/`pipe_write()` in `fs/pipe.c`). Dabei wird der zur Pipe gehörige Speicher (`base`) als Ringpuffer interpretiert, in den ab `start len` Bytes geschrieben worden sind, ohne daß sie bereits wieder ausgelesen wurden. Dabei wird immer unterschieden, ob für den Dateideskriptor das O_NONBLOCK Flag gesetzt wurde. Ist es gesetzt, dürfen die I/O-Operationen nicht blockieren. Wenn

	blockierend	nicht blockierend
gesperrte Pipe	Blockieren des aufrufenden Prozesses	Fehler EAGAIN
leere Pipe	Blockieren des rufenden Prozesses, wenn schreibende Prozesse vorhanden sind, sonst 0 zurückgeben	Fehler EAGAIN, wenn schreibende Prozesse existieren, sonst 0
ansonst	max. Anzahl von Zeichen bis zur angeforderten Position lesen	wie bei der blockierenden Operation

Tabelle 1: Semantik der Pipe-Leseoperation

die Anzahl der zu schreibenden Bytes nicht die interne Puffergröße übersteigt, muß die Schreiboperation *atomar* ausgeführt werden. Das heißt, wenn mehrere Prozesse in die Pipe schreiben, werden die Bytefolgen der einzelnen Schreiboperationen nicht zerstört. Die Semantik der Lese- und Schreib-Operationen ist in den Tabellen 1 und 2 zusammengefasst. Da Prozesse beim Zugriff auf Pipes häufig blockieren, müssen die Schreib- und Leseoperationen dementsprechend oft Prozesse der, zum Inode gehörenden Warteschlange (`wait`) wecken. Dabei werden alle Prozesse in einer einzigen Warteschlange verwaltet, obwohl sie auf unterschiedliche Ereignisse warten.

4.4 Modifikation der Linux-Pipe

Die Tatsache, daß die Pipes im Normalbetrieb in einem laufenden System sehr intensiv verwendet werden, hat die Art der Entwicklung mitbestimmt. Um den normalen Betrieb einer Pipe nicht zu beeinflußen, kann sie zwei Zustände annehmen: NORM_PIPE und MAP_PIPE. Im Normalzustand läuft die Funktion nachwievor genauso wie beschrieben ab (Ringpuffer). Wird eine Pipe mit `mmap()` eingeblendet, geht sie in einen neuen Zustand (MAP_PIPE) über. Dadurch konnte der Normalbetrieb des Systems während der Entwicklung des Pipe-Mappings aufrechterhalten werden. Genaugenommen gibt es drei Zustände. Wenn alle Einblendungen einer Pipe bereits entfernt wurden, die Pipe aber noch mehr als eine Speicherseite an Inhalt hat, befindet sie sich in einem Übergangszustand (MTON_PIPE).

4.4.1 Datenstrukturen

Für die Darstellung einer eingeblendeten Pipe, werden einige zusätzlich Informationen benötigt. Daher wurde die Datenstruktur `struct pipe_inode_info` erweitert:

	blockierend	nicht blockierend
kein Leser	SIGPIPE an den Write-Prozesses; Fehler EPIPE	wie blockierend
gesperrte Pipe	Blockieren des aufrufenden Prozesses	Fehler EAGAIN
atomares Write möglich, zuwenig Platz im Puffer	Blockieren des aufrufenden Prozesses	Fehler EAGAIN
Puffer für atomares Write ausreichend	Schreiben in den Puffer	wie blockierend
ansonsten	Abwechselnd Write und Blockieren bis geforderte Zahl geschrieben	soviel wie freier Puffer auf einmal schreiben

Tabelle 2: Semantik der Pipe-Schreiboperation

```
struct pipe_inode_info {
    unsigned char mode;                          /* Modus der Pipe */
    struct wait_queue * wait;                     /* Warteschlange */
    char * base;             /* Basisadresse der Ringpuffer-Page */
    unsigned long start;  /* Aktueller Bereich Anfang; (normal) */
    unsigned long wstart;      /* Schreibbereich Anfang (write) */
    unsigned long len;              /* Aktueller Bereich Laenge */
    unsigned int lock;                         /* Sperre der Pipe */
    unsigned int rd_openers;       /* Anzahl der Leser-Prozesse */
    unsigned int wr_openers;   /* Anzahl der Schreiber-Prozesse */
    unsigned int readers;          /* Aktuelle Anzahl der Leser */
    unsigned int writers;      /* Aktuelle Anzahl der Schreiber */
    unsigned long bufsize;          /* Groesse des Pipe-Puffers */
    unsigned long map_base;          /* Offset der Puffer-Basis */
    unsigned long map_start;               /* Pipe-Start-Offset */
    struct pbpage *page_list;        /* Liste der Puffer-Seiten */
    struct pipe_mapper * rmap;         /* Liste der Read-Mapper */
    struct pipe_mapper * wmap;      /* Liste der Schreib-Mapper */
    unsigned int dlevel;  /* Diagnoselevel der Kernel-Meldungen */
};
```

Neu sind dabei die Komponenten `mode`, `wstart`, `bufsize`, `map_base`, `map_start`, `page_list`, `rmap`, `wmap` und `dlevel`.

- `dlevel` dient nur zum Testen. Damit kann das Ausmaß an Kernel-Meldungen gesteuert werden.

- `mode` ist der Indikator für den aktuellen Zustand der Pipe.

- Der Wert `wstart` entspricht im Prinzip der Summe von `start` und `len`, wird aber in einigen Sonderfällen, wo `len` bereits verändert wurde, noch gebraucht. Außerdem gibt es zwei Längenwerte bei der eingeblendeten Pipe: die Länge aus der Sicht der I/O-Operationen (Wrw_{off} - Rrw_{off} bzw NPIPE_WSTART - NPIPE_RSTART) und die Länge die beim Einblenden verfügbar ist (W_{off} - R_{off} bzw MPIPE_WSTART - MPIPE_RSTART)

- Die Größe der Zugriffszone (Buf_0 bis Buf_{max}, Abb.: 14 bzw. Abb.: 20) wird in `bufsize` abgelegt. Bei einer Pipe im Normalzustand ist dieser Wert gleich PAGE_SIZE. DEFAULT_BUF_SIZE (`include/linux/pipe_fs_i.h`) ist der Wert mit dem diese Variable beim Einblenden der Pipe initialisiert wird. Er ist ein ganzzahliges Vielfaches der PAGE_SIZE. Dieser Faktor legt damit auch die Länge der Liste der Puffer-Pages (`page_list`) fest. Er wird unter dem Namen DEFAULT_BUF_PAGES ebenfalls in `include/linux/pipe_fs_i.h` definiert.

- Der Offset innerhalb der Pipe, der den Anfang der ersten Page der Zugriffszone angibt (Abb.: 20), wird in `map_base` gespeichert. `map_start` ist der Offset des Anfangs der Zugriffszone innerhalb dieser ersten Seite.

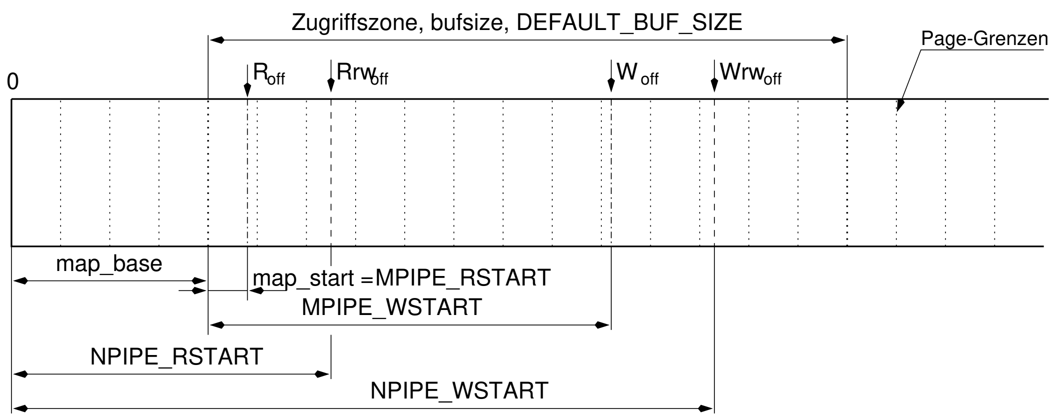

Abbildung 20: Kenngrößen der eingeblendeten Pipe

- Der Pipe-Puffer wird durch die Liste `page_list` repräsentiert. Ihre Elemente sind nicht nur Page-Frames sondern, da zusätzliche Informationen zu den Speicherseiten gebraucht werden, vom Typ `struct pbpage`. Dieser Typ ist ebenfalls in `include/linux/pipe_fs_i.h` definiert:

```
struct pbpage {
    unsigned int ppflags;
    pte_t ppte;
};
```

- Die einzigen, zur Zeit verwendeten Flag-Werte für `ppflags` sind PBF_INIT und PBF_SYNC. Ist zu einer Puffer-Page PBF_SYNC gesetzt, wurde bereits ein `msync()` darauf ausgeführt. PBF_INIT signalisiert, daß die Page mit 0 initialisiert werden muß. Da zu den Seiten des Puffers sämtliche Paging-Informationen benötigt werden, sind die Pageframes nicht direkt in dieser Struktur eingetragen, sondern ganze Page-Tabellen-Einträge (PTE). Damit können die Funktionen des Memory-Managements selbst verwendet werden, um festzustellen, ob eine Seite bereits vorhanden, oder ausgelagert ist. Genauso können die Erfordernisse für MAP_PRIVATE Einblendungen erfüllt werden (CoW).

- Die Einblendungen einer Pipe werden in den beiden Mapper-Listen `rmap` (Lese-Einblendungen) und `wmap` (Schreib-Einblendungen) erfasst. Die Kenndaten einer Einblendung werden in der Struktur `struct pipe_mapper` zusammengefasst (`include/linux/pipe_fs_i.h`):

```
struct pipe_mapper {
    struct task_struct * task;              /* Prozess-ID */
    int mode;           /* Lese- oder Schreib-Einblendung */
    unsigned long vma_start;      /* Startadresse des VMA's */
    unsigned long act_pos;      /* RD/WR-Offset in der Pipe */
    unsigned long rel_len;           /* Auto-Commit-Laenge */
    struct pipe_mapper * next;        /* Listen-Verkettung */
};
```

Über diese Informationen wird der Zusammenhang zwischen den Adressen eines VMAs im Adressraum des User-Prozesses und den Offsets im Pipe-Puffer hergestellt. Die Schlüsselwerte für diesen Zusammenhang sind die Startadresse des VMAs im virtuellen Adressraum des Prozesses `vma_start` und die der Offset - `act_pos` - im Pipe-Buffer, ab dem der Prozess die Pipe eingeblendet hat. Abb.: 21 veranschaulicht das Verhältnis der Werte zueinander.

Jeder dieser Mapper-Entries ist über `vma_start` eindeutig einem VMA zugeordnet. Umgekehrt könnte der Mapper-Entry zu einem VMA über die VMA-Komponente `vm_inode` und einer Suche in den Mapper-Listen ermittelt werden. Dieser Vorgang ist aber sehr aufwendig. Für die Imple-

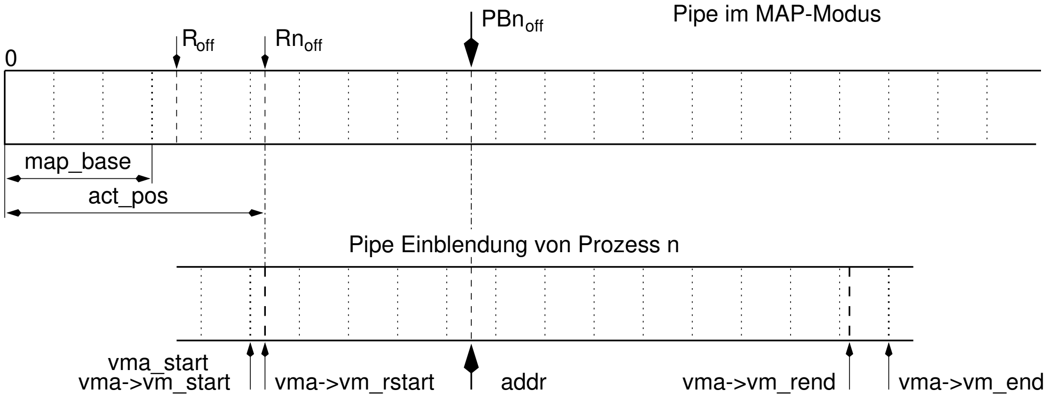

Abbildung 21: Zusammenhang zwischen VMA-Adresse und Pipe-Offset

mentierung von SysV-Shared-Memory ist eine eigene Komponente in der VMA-Struktur vorhanden, die ansonsten ungenutzt ist (`unsigned long vm_pte`). Mit Hilfe dieses Elementes wird eine direkte Verbindung zwischen dem VMA und dem Mapper-Entry hergestellt. `vm_pte` wird beim Einblenden mit der Adresse des neuen Mapper-Entries initialisiert. Um den Zugriff auf diesen Wert verständlicher zu machen, wird das Macro VMA_PMAP() in `include/linux/mm.h` definiert.

Soll zu einer virtuellen Adresse `addr` im VMA die entsprechende Position in der Pipe PBn_{off} ermittelt werden, so wird zuerst der Offset der Adresse in Bezug auf den realen Anfang des VMAs (`vm_rstart`) benötigt. Dazu wurde das Macro `VMA_OFF(vma, addr)` (`include/linux/mm.h`) definiert. Die Anfangsposition des Prozesses in der Pipe ist im Mapper-Entry in der Komponente `act_pos` abgelegt. Dieser Wert ist das Äquivalent zum realen Anfang des VMA `vm_rstart`. Hat man den Offset einer Adresse in Bezug auf `vm_rstart`, kann der Offset in der Pipe ganz einfach ermittelt werden, indem dieser Wert zur Anfangsposition Rn_{off} bzw. `act_pos` (aus dem Mapper-Entry) addiert wird. Das Ergebnis PBn_{off} ist der Offset in Bezug auf den Nullpunkt in der Pipe. Um aber die Speicherseite, in der die gesuchte Adresse liegt, zu erhalten, muß der Offset relativ zum Anfang der Zugriffszone ermittelt werden. Der Nullpunkt der Zugriffszone ist der Wert `map_base`. `map_base` vom zuvor ermittelten Offset PBn_{off} subtrahiert, ergibt den Offset in der Zugriffszone. Daraus kann durch Division mit PAGE_SIZE der Index der Page in der Pipe-Page-Liste errechnet werden. In `include/linux/pipe_fs_i.h` sind Macros für all diese Berechnungen definiert. Die hier beschriebene Berechnung wird z.B. im Falle des ersten Zugriffs auf eine Adresse der eingeblendeten Pipe benötigt. Der Page-Fault-Handler versucht über die objektspezifische `nopage()`-Funktion einen Pageframe zu ermitteln. In dieser Funktion wird mit Hilfe dieser Berechnung die gesuchte Page in der Pipe-Page-Liste bestimmt.

4.4.2 Funktionen

Der Ablauf beim Einblenden eines Objektes in den Adressraum mittels `mmap()` teilt sich entsprechend dem unter Linux angewendetem Konzept ebenfalls wieder in zwei Teile. Die allgemeinen Aufgaben werden in der Funktion `do_mmap()` (`mm/mmap.c`) erledigt. Die speziellen Aufgaben, die von der Art des eingeblendeten Objektes abhängen, werden in einer objektspezifischen Funktion ausgeführt. Das einzublendende Objekt wird über einen Filedeskriptor dem `mmap()`-Aufruf angegeben. Damit steht die File-Struktur und die Inode-Struktur des Objektes zur Verfügung. Der Gegenstand der Einblendung ist aber die File-Struktur, zu der auch eine Methode für das Einblenden definiert ist. Der File-Operationsvektor verfügt daher über eine `mmap()`-Komponente, die für Pipes mit NULL belegt war.

Für die Erweiterung von `mmap()` auf Pipes wurde eine neue objektspezifische Funktion `pipe/mmap()` (`fs/pipe.c`) benötigt. Ihre Aufgabe ist es, eine Pipe die sich im Normalzustand befindet, in den MAP-Modus überzuführen. Dabei gilt es, einige Fälle, den Pipe-Zustand betreffend, zu unterscheiden:

- Die Pipe ist leer.
  Dieser Fall macht weiter keine Probleme. Die Pipe im MAP-Modus wird damit durch die Situation $R_{off} = W_{off} = 0$ charakterisiert.

- Die Pipe ist voll.
  Das Ergebnis muß darin bestehen, daß eine Speicherseite bereits zum Lesen zur Verfügung steht: $R_{off} = 0$, $W_{off} =$ PAGE_SIZE. Wobei allerdings berücksichtigt werden muß, daß der Inhalt des Pipe-Puffers nicht unbedingt kontinuierlich ist. Der Lese-Anfang im Ring-Puffer kann an einer beliebigen Stelle liegen. In diesem Fall wird eine Linearisierung des Puffer-Inhalts durchgeführt.

- Die Pipe ist nicht ganz gefüllt.
  Da noch keine Speicherseite voll beschrieben ist, kann sie auch noch nicht zum Lesen freigegeben werden. Der Puffer-Inhalt muß aber linearisiert werden, sodaß die Daten ab Offset 0 in der ersten Page stehen. Damit wird $R_{off} = 0$ und $W_{off} =$ `len`.

Weiters wird die Repräsentation der erzeugten Einblendung in Form eines neuen Elementes der Mapper-Liste initialisiert und entsprechend der Art der Einblendung in eine der beiden Listen `rmap` oder `wmap` eingetragen. Auch wird die VMA-Struktur mit den objektspezifischen Werten initialisiert. Das betrifft vor allem den Operationsvektor der VMA-Struktur. Dieser wird mit den neuen Funktionen zur Manipulation einer Pipe-Einblendung belegt:

```
static struct vm_operations_struct pipe_mmap_ops = {
    pipemap_open,               /* new : open */
    NULL,                      /* new : close */
    pipemap_unmap,             /* new : unmap */
    NULL,               /* no special protect */
    pipemap_sync,      /* new : sync (commit) */
```

```
    NULL,                /* not used in 2.0.x */
    pipemap_nopage,           /* new : nopage */
    NULL,                /* not used in 2.0.x */
    pipemap_swapout,         /* new : swapout */
    pipemap_swapin,           /* new : swapin */
};
```

Diese Funktionen stellen das Kernstück am Pipe-Mapping dar (alle in `fs/pipe.c`). Die Funktion `pipemap_nopage()` (`fs/pipe.c`) ist die zentrale Routine, die immer dann aufgerufen wird, wenn durch eine Adressreferenzierung ein Page-Fault ausgelöst wurde. Dabei ergeben sich je nach Art der Einblendung unterschiedliche Aufgaben. Bei einer Schreib-Einblendung braucht lediglich eine neue Speicherseite angefordert werden. Dieser Page-Frame wird in den PTE in der Page-Liste eingetragen. Lediglich wenn die referenzierte Adresse über der Obergrenze der Zugriffszone liegt, muß der betroffene Prozess solange blockiert werden, bis durch eine Freigabe am unteren Zonenende die Erweiterung am oberen Ende möglich wird. Tritt der Page-Fault in einer Lese-Einblendung auf, so wird - wenn die gesuchte Seite bereits zum Lesen freigegeben wurde - der Page-Frame aus der Page-Liste ermittelt und an den Page-Fault-Handler retourniert. Kann die gesuchte Seite noch nicht gelesen werden, wird der betroffene Prozess blockiert. Das Blockieren erfolgt durch den Aufruf von `interruptible_sleep_on()`. Dabei wird der Prozess in die angegebene Warteschlange (`wait`-Komponente der Pipe) eingetragen und danach wird die Kontrolle an den Scheduler übergeben. Die Prozesse, die auf einer Warteschlange schlafen, können mit dem Aufruf von `wake_up_interruptible()` wieder geweckt werden.

Eine weitere wichtige Aufgabe, die in `pipemap_nopage()` erledigt wird, ist die Kontrolle, ob eine Seite des Puffers ausgelagert wurde. Im Fall, wo ein Prozess das erste Mal auf eine Seite des Puffers zugreift kann es sein, daß diese Seite gar nicht im Speicher ist. Dies ist am Zustand des betreffenden PTEs in der Page-Liste erkennbar (`pte_present()`). Sollte die Seite ausgelagert sein, so muß die Einlagerung über `do_swap_page()` (`mm/memory.c`) veranlaßt werden.

`pipemap_open()` wird immer dann aufgerufen, wenn ein VMA neu erzeugt wird. Das ist z.B. dann der Fall, wenn ein VMA durch eine Operation wie `munmap()` oder `mprotect()` geteilt wird. Auch auch, wenn bei einem `fork()` ein Prozess neu erzeugt wird und eine Pipe-Einblendung erbt. Genau in diesem Fall sind aber die Möglichkeiten, die in der Linux-Implementierung zur Verfügung stehen, unvollständig. Die Informationen zum aktuellen Prozess stehen in Form einer globalen Variablen `current` (Pointer auf eine `struct task_struct` Variable) bereit. Im Falle eines `fork()` ist das aber nicht richtig. Zu einem Zeitpunkt, zu dem eigentlich Datenstrukturen des neu erzeugten Prozesses bearbeitet werden, verweist `current` aber auf den Vater-Prozess. In `pipemap_open()` muß aber ein neuer Mapper-Eintrag für den Sohn-Prozess erzeugt werden, dessen Informationen aber nicht verfügbar sind, da `current` den Vater referenziert. Daher wurde das Interface der `open()`-Komponente um zusätzliche Parameter erweitert: Ein `int`-Wert `mode`, mit dem Informationen über die aufrufende Funktion angegeben und ein `void *addpara` über den - je nach Situation -

beliebige Werte übergeben werden können. Für `mode` sind zwei Werte definiert (`include/linux/mm.h`):

- VM_OC_NOOP: bietet in einigen Sonderfällen die einfache Möglichkeit die Funktion von `pipemap_open()` zu unterbinden. Ein zusätzlicher Parameter wird in diesem Fall nicht übergeben.
- VM_OC_FORK: Wird `pipemap_open()` im Zuge eines `fork()` aufgerufen, so wird als zusätzlicher Parameter ein Pointer auf die Task-Struktur des Sohn-Prozesses übergeben. Jetzt können die, für den neuen Mapper-Eintrag benötigten Informationen aus der Task-Struktur entnommen werden.

`pipemap_unmap()` sorgt für die korrekte Behandlung der Situationen im gemischten Betrieb (siehe Kapitel 3.2.3). Wegen dem zuvor angesprochenen `fork()`-Problem und dem notwendigen Zusammenspiel mit der `sync()`-Komponente, mußte auch hier das Interface erweitert werden. Zu den ursprünglichen drei Parametern (`struct vm_area_struct * unsigned long` und `size_t`) kommt noch die Information zum betroffenen Task: `struct task_struct *`.

`pipemap_sync()` übernimmt bei einer eingeblendeten Pipe die Aufgabe eines Commit des angegebenen Bereiches. Die Semantik dieser Funktion wurde bereits in den Kapiteln 2.1.3, 3.2.1 und 3.2.3 erläutert.

Aufgrund der Einschränkungen im VMM im Kernelsegment (siehe 4.1.1) stellt sich das Problem, daß Pages, die zwischen dem User- und dem Kernel-Segment geteilt werden sollen, als unauslagerbar markiert werden müssten. Um aber in der Praxis verwendbar zu sein, müssen die Pipe-Einblendungen aber auf jeden Fall im *Demand Paging*-Pool enthalten sein. Nur so kann der, mitunter sehr große Speicherbedarf einer Pipe-Einblendung erfüllt werden. Abgesehen davon sollen ja in einem System beliebig viele solcher Einblendungen zur selben Zeit existieren können.

Für eine endgültige Implementierung ist natürlich eine Änderung im Kernel-VMM vorzunehmen. Diese Erweiterung würde aber den Rahmen der vorliegenden Arbeit sprengen. Die prinzipielle Funktion des Modells läßt sich auch ohne diese Funktionalität beurteilen.

Das ist auch der Grund, warum vorerst auf eine Implementierung der beiden Swap-Funktionen `pipemap_swapout()` und `pipemap_swapin()` verzichtet wurde. Die grundsätzliche Funktion dieser beiden Operationen soll aber trotzdem dargestellt werden.

`pipemap_swapout()` wird über die Swap-Funktion `swap_out()` (`mm/vmscan.c`) angesprochen. Beim Auslagern einer Speicherseite, die Teil des Pipe-Puffers ist, ist es notwendig, die PTEs aller Prozesse, die diese Pipe eingeblendet haben, anzupassen. Ansonsten könnte es vorkommen, daß ein Prozess auf einen Pageframe zugreift, dessen Inhalt inzwischen ausgelagert und durch einen anderen Inhalt ersetzt wurde. Um die Änderungen in den Page-Tabellen durchführen zu können, müssen alle Prozesse ermittelt werden, die diese Pipe eingeblendet haben. Diese Information ist in der Mapper-Liste enthalten. Allerdings ist über diesen Weg kein direkter Zugriff auf die betroffenen VMAs möglich. Einfacher erhält man die betroffenen VMAs über die Memory-Map Liste des Inodes. Die Inode-Komponente `struct vma_struct *i_mmap` ist der Einstiegspunkt in eine Ringliste, in der alle VMAs, die Einblendungen dieses Inodes repräsentieren, enthalten sind. Mit Hilfe der Komponenten `vm_next_share` und `vm_next_prev` der VMA-Struktur wird eine doppelt verkettete Ringliste - der *Inode i_mmap Ring* - aufgebaut. Der Eintrag in dieser Liste erfolgt beim Einblenden durch `do_mmap()`.

`pipemap_swapin()` wird aufgerufen, um die gewünschte Page wieder aus dem Auslagerungsbereich zurückzuholen. Hier gibt es allerdings keine allgemeine `swapin()`-Funktion, die ihrerseits auf die objektspezifische Komponente zurückgreift. Im Falle eins Page-Fault, bei dem durch `pte_present()` signalisiert wird, daß die betreffende Seite ausgelagert ist, wird `do_swap_page()` (`mm/memory.c`) aufgerufen. In dieser Funktion wird entweder die allgemeine Swapin-Funktion `swap_in()` (`mm/page_alloc.c`), oder die objektspezifische Komponente `swapin()` des VMA-Operationsvktors aufgerufen. Die Aufgabe von `pipemap_swapin()` besteht darin, festzustellen, ob die Seite inzwischen von jemand anderem bereits eingelagert wurde. Ist das der Fall, ist die Seite wieder in der Page-Liste der Pipe vorhanden. `pte_present()` am betroffenen PTE läßt das erkennen. Es braucht nur die Adresse des Pageframes retourniert zu werden. Ist die Seite noch nicht eingelagert, greift `pipemap_swapin()` selbst auf die allgemeine Funktion `swap_in()` zurück. Nachdem `swap_in()` die gesuchte Seite wieder eingelagert hat, muß diese von `pipemap_swapin()` in den PTE in der Pipe-Liste eingetragen werden. `pipemap_swapin()` gibt seinerseits die Page als Returnwert zurück. Die Eintragung der Page in den PTE des Prozesses erfolgt in der nächsthöheren Ebene.

5 Praktische Anwendungen und Benchmarks

Anhand von zwei einfachen Applikationen, die aber genau in die definierte Zielgruppe passen, wird die praktische Anwendung von Memory Mapping gezeigt. Dabei handelt es sich um den *Word-Count* `wc` und den Standard-Filter `grep`. Weiters wurde für eine spezielle Testanordnung eine Minimalvariante von `cat` implementiert.
Die Sourcen von `wc` wurden dem GNU-Paket textutils-1.22 entnommen. `grep` stammt aus dem GNU-Paket grep-2.0.

Ziel ist es, zwei Gruppen von Tests durchzuführen. Erstens wird ein Vergleich der Programm-Varianten unter Verwendung von Standard-I/O und Datei-Mapping auf einigen unterschiedlichen Plattformen durchgeführt.
Zweitens werden die Applikationen auf Verwendung des erweiterten `mmap()` für Pipes umgestellt. Dabei wird einerseits die Anwendung der Methode präsentiert, andererseits werden ebenfalls Vergleichsmessungen durchgeführt. Dieser Teil wird natürlich nur unter Linux ausgeführt.

Mit Hilfe des Shell-Scripts `cmd_time` ([Syr00]) wurden die Messtabellen erstellt. Das Script führt ein Kommando, das als Parameter angegeben wird mit der, ebenfalls angegebenen, Wiederholungszahl aus. Aus dem angegeben Kommando wird intern wieder ein Shell-Script erzeugt, das mit `time` aufgerufen wird. Dabei werden die Zeit-Werte im BSD-Style ermittelt und durch eine Folge von `grep`- und `sed`-Aufrufen die Tabelleneinträge aufgebaut.

5.1 Vergleich von Standard-I/O und Datei-Mapping

Für die Durchführung der Testreihe, wurde das vorhandene `wc` um eine Variante ergänzt, die mit `mmap()` die zu bearbeitende Datei in den Speicher einblendet. In der Originalvariante wird die Eingabe blockweise in einer Schleife eingelesen. Die Blockgröße beträgt dabei 16kByte.

In der Version unter Verwendung von `mmap()` wird die Schleife nur mehr ein einziges Mal durchlaufen, da der Block jetzt die gesamte Datei umfasst.

Das Programm `grep` ist bereits in einer Variante vorgelegen, die `mmap()` verwendet hat. Die Art der Verwendung war allerdings nicht wirklich sinnvoll. Die Standard-I/O-Variante arbeitet blockweise, wobei die Blockgröße architekturabhängig ermittelt wird. Auf den verwendeten Systemen ergab sich jeweils ein Größe von 32kByte. Die vorgegebene `mmap()`-Variante arbeitet genauso wie die Standard-I/O-Version. Es wurde lediglich das `read()` durch ein `mmap()` ersetzt. Da der `mmap()`-Aufruf selbst aber relativ teuer ist, kann auf diese Weise keine Effizientszunahme erreicht werden.

Das Programm wurde auf zwei Arten verändert. Eine Version verwendet weiterhin die Standard-I/O-Methode, liest aber die Datei in einem Stück in den Speicher ein (*Slurping*. Die angepasste `mmap()`-Variante blendet die Datei vollständig in den Speicher ein. Dadurch ist der Vergleich zwischen den Kosten durch das Kopieren bei der Standard-I/O-Methode mit dem *On Demand*-Zugriff bei `mmap()` besser möglich. Beim Slurping von großen Dateien (Größe im Bereich des freien Speichers und darüber) erhält man auch einen Eindruck von der Qualität des betroffenen Memory Managements.

Die Messungen umfassen daher zwei Programm-Varianten von `wc` und drei Programm-Varianten von `grep`. Alle Messungen wurden mit Dateien mit Größen zwischen 1 MByte und 55 MByte durchgefürt. Sie erfolgten auf folgenden Plattformen:

- FreeBSD 2.2.6
  AMD 486/133, 64 MByte Hauptspeicher.
- Linux 2.0.36
  AMD 486/133, 64 MByte Hauptspeicher.
- SunSolaris
  Sun Ultra Sparc 5, 64 MByte-Hauptspeicher.
- SCO 5.0
  Pentium 133, 48 MByte-Hauptspeicher (kein L2-Cache).

Es wurden folgende Kommando-Aufrufe durchgeführt:

```
wc <Dateiname>
grep "zoneinfo"  <Dateiname>
```

Der Wort-Count `wc` hat daher die Werte für die Anzahl der Zeichen, Worte und Zeilen ermittelt.

Bei den Messungen wurden drei Werte erfasst. Die User-Zeit, die System-Zeit und die Gesamtzeit. Aus dem User- und dem System-Wert wurde die CPU-Zeit durch Addition ermittelt. Jede Messung wurde 10 mal wiederholt, als repräsentativer Wert wurde der angenäherte Median (5. Wert) herangezogen. Die vollständigen Tabellen sind dem Dokument aus Platzgründen nicht angeschlossen. Im Anhang A finden sich die reduzierten Tabellen der Median-Werte. Sie sind die Basis der grafischen Darstellungen.

5.1.1 Ergebnisse

Die erhaltenen Messwerte (Abbildungen 22 bis 29) beschreiben ein Verhalten das der Erwartung entspricht. Unabhängig von der Plattform zeigt die Programmvariante unter Verwendung der `mmap()`-Methode den günstigeren Verlauf bei der Systemzeit. Trotz der leichten Erhöhung der User-Time ergibt sich damit eine deutliche Reduktion der CPU-Zeit. Die Veränderung der User-Time hat die Ursache vermutlich in erhöhten Cache-Misses. Bei der Standard-I/O Variante wird in einem Speicherbereich (der I/O-Puffer) gearbeitet, der schon durch die `read()`-Operation zur System-Zeit in den Cache geladen wurde. Die Kopieroperation des `read()` kostet Zeit, die durch den hohne System-Zeit-Wert zu sehen ist, sorgt aber dafür, daß die nachfolgende Referenzierung der Daten im User-Prozess sofort über den (L2) Cache erfolgt.

Im `mmap()`-Fall wird die Zeit für die Kopieroperation gespart, daher die geringere System-Zeit. Werden die Daten aber referenziert, liegt der letzte Zugriff bereits relativ lange zurück, sie werden daher tatsächlich erst aus dem Hauptspeicher gelesen. Dieser Aufwand wird durch die User-Zeit erfasst.

In beiden Fällen wird mit einem "warmen" Buffer-Cache gearbeitet - die Inhalte sind bereits von der Platte eingelesen. Der Aufwand dafür ist jeweils im ersten Wert der Messwiederholungsreihen zu sehen (nicht in den Tabellen im Anhang, da das schon die Tabellen der Median-Werte sind; vollständige Tabellen bei [Syr00] zu finden).

Erreicht die bearbeitete Datei die Größenordnung des, im System freien Speichers, beginnen die Aktivitäten des Pagings. Das ist durch einen deutlichen Anstieg der Kurven im oberen Bereich zu erkennen.

Direkt vergleichbar sind die Messungen unter FreeBSD und Linux (Abb. 22 mit Abb. 25 und Abb. 26 mit Abb. 29), da sie auf dem gleichen Gerät durchgeführt wurden.

Das unterschiedliche Verhältniss von User- zu System-Zeit bei `grep` und `wc` resultiert daher, daß bei `wc` für die Ermittlung der Byte-Anzahl jedes Zeichen einzelnen betrachtet wird. Bei `grep` kann durch die Verwendung einer *Boyer-Moore-Gosper*-basierten Suche (siehe Datein AUTHORS und README im GNU-grep2.0-Paket) verhindert werden, daß jedes Zeichen untersucht werden muß.

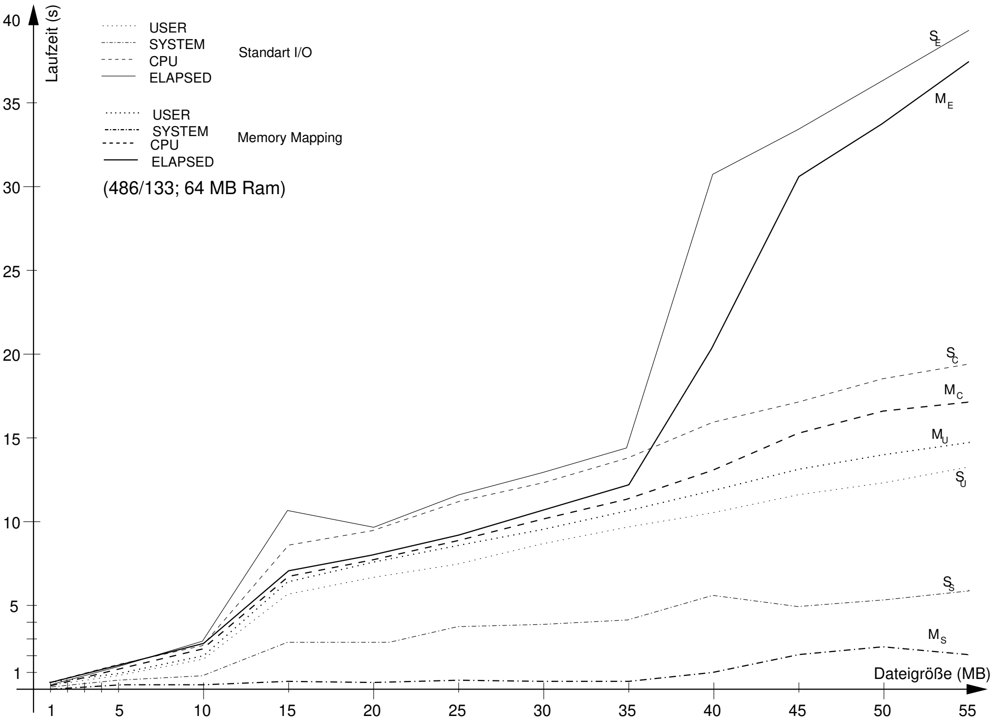

Abbildung 22: wc mit read() oder mmap() unter BSD

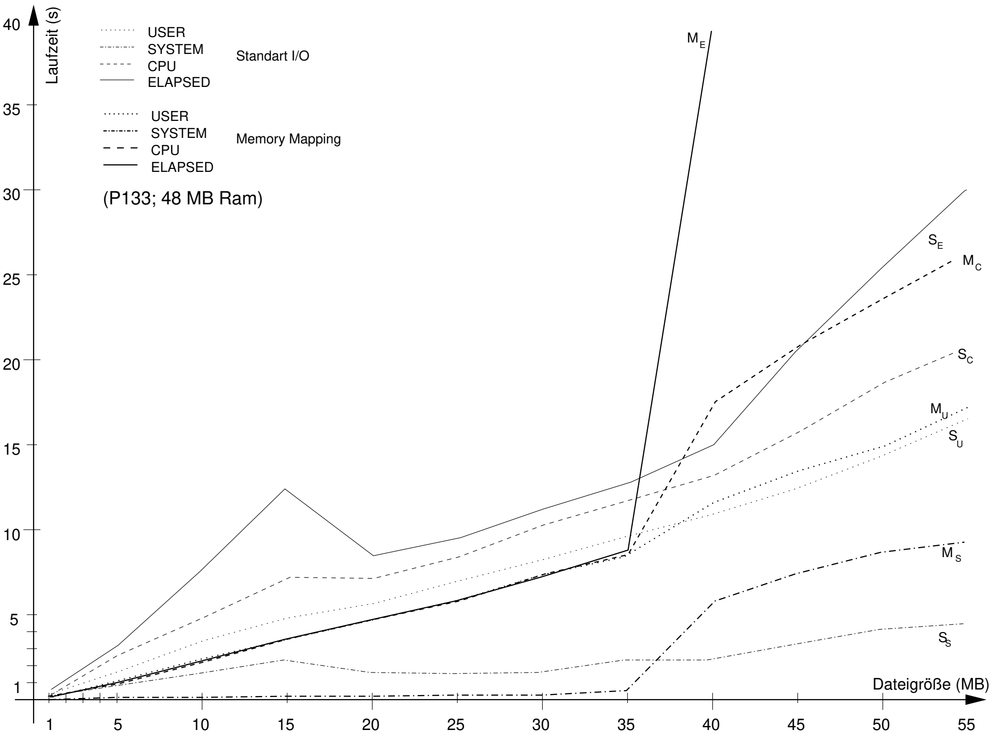

Abbildung 23: wc mit read() oder mmap() unter SCO

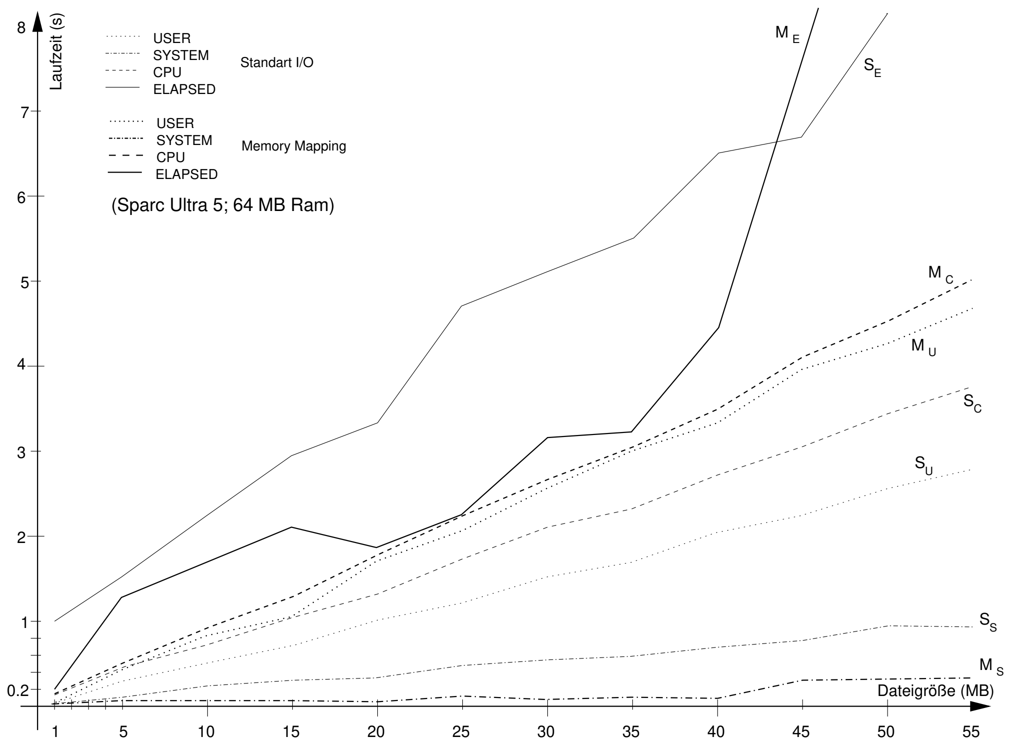

Abbildung 24: wc mit read() oder mmap() unter Sun Solaris

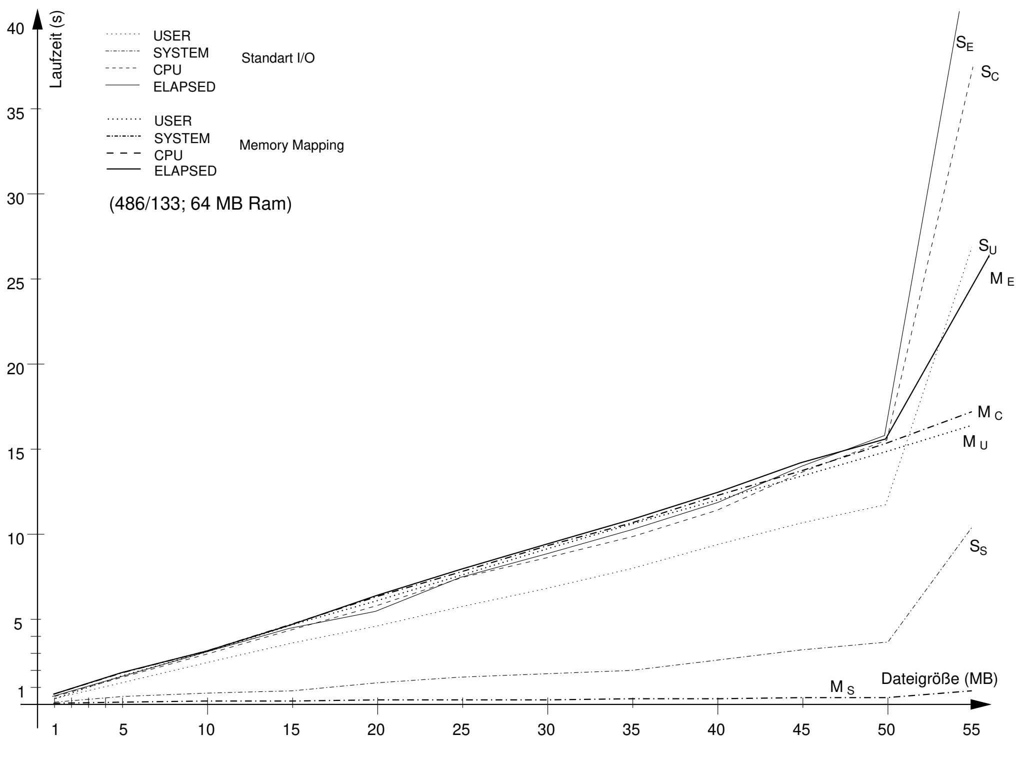

Abbildung 25: wc mit read() oder mmap() unter Linux

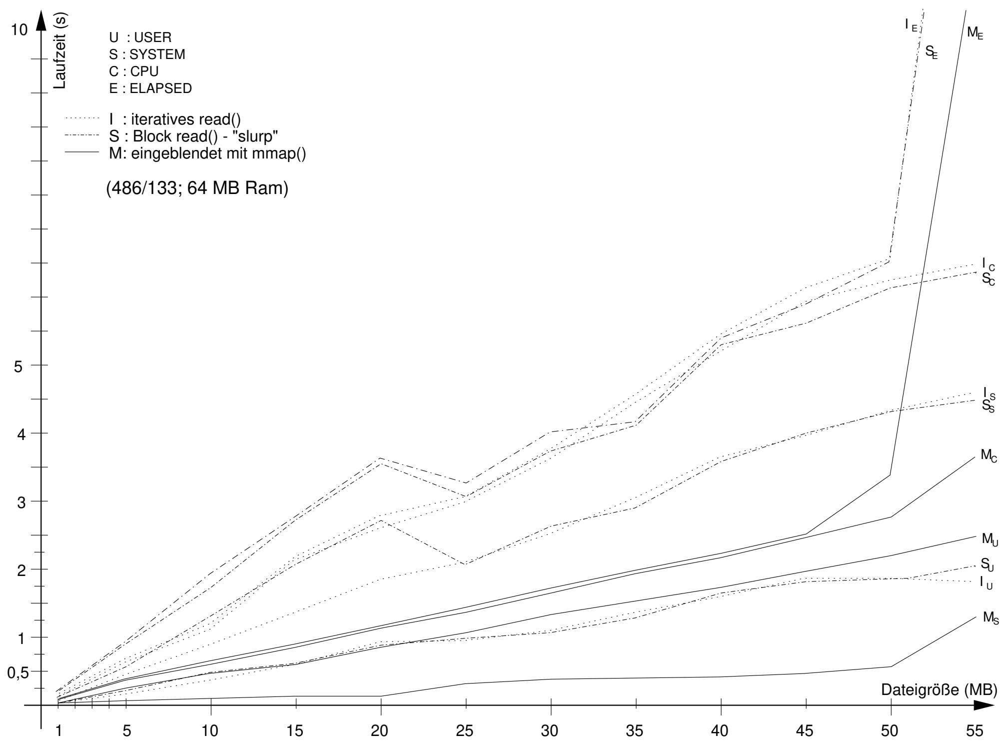

Abbildung 26: grep mit read() (iterativ/slurp) oder mmap() unter BSD

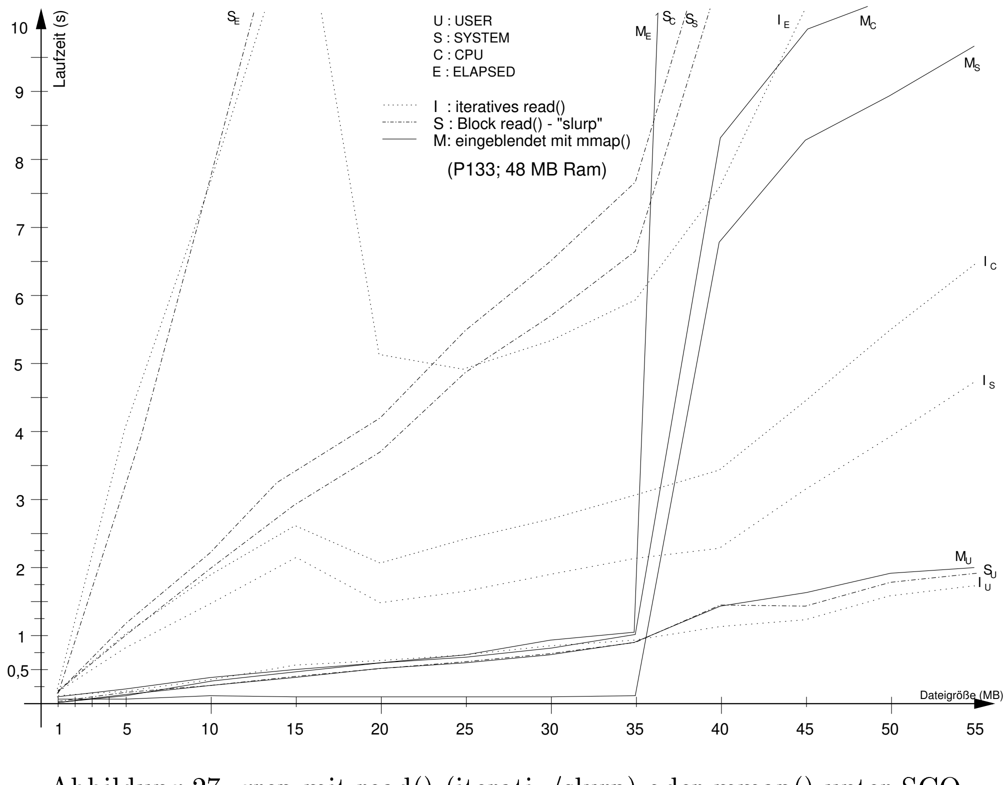

Abbildung 27: grep mit read() (iterativ/slurp) oder mmap() unter SCO

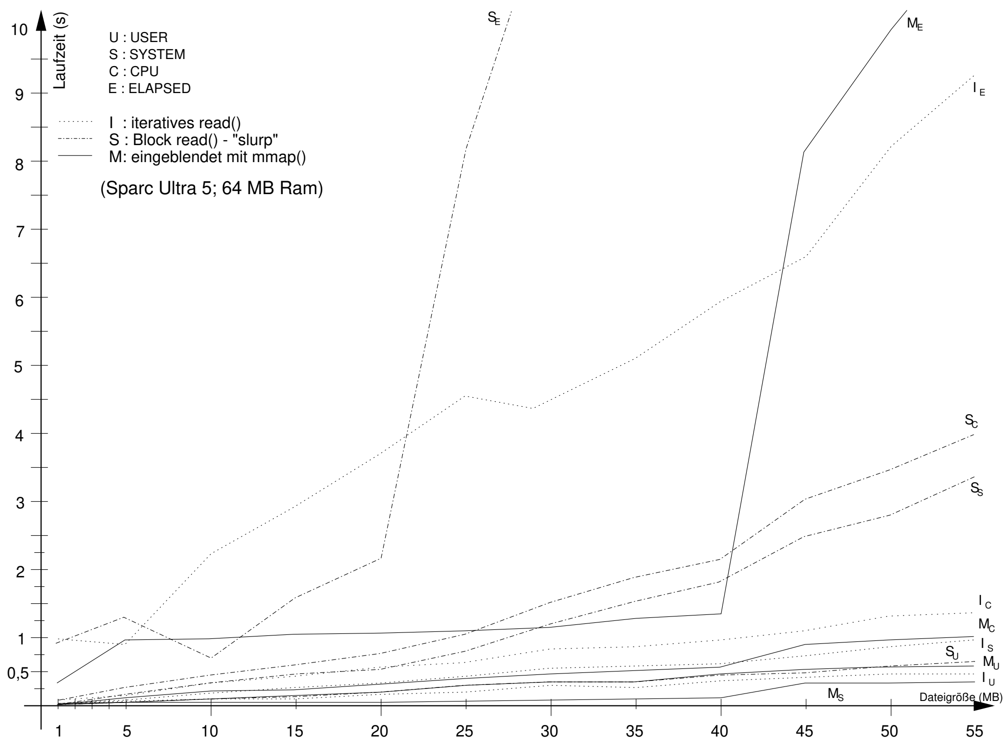

Abbildung 28: grep mit read() (iterativ/slurp) oder mmap() unter Sun Solaris

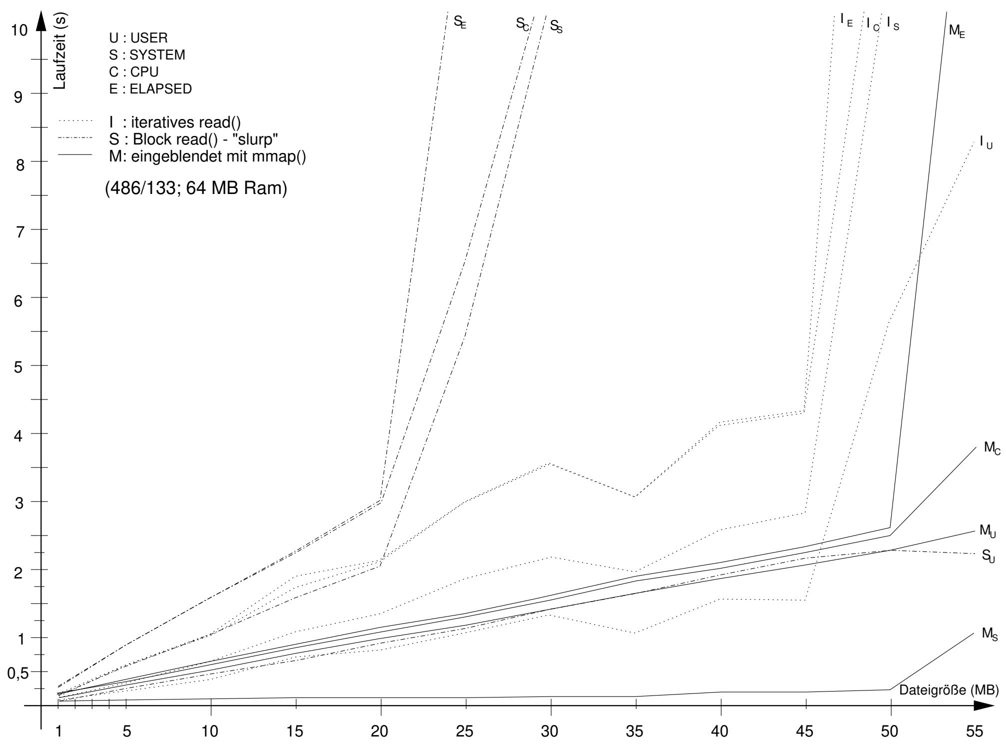

Abbildung 29: grep mit read() (iterativ/slurp) oder mmap() unter Linux

5.2 Anwendung des Pipe-Mappings

Ausgehend von den Änderungen wurden die Programme `wc` und `grep` auf die Verwendung des erweiterten `mmap()` angepasst. Dafür sind im Wesentlichen zwei Schritte notwendig:

1. Installation eines Signal-Handlers um einen Endwert zu setzen.
2. Zusätzliche Schleife um die Pipe gegebenenfalls mit einem geänderten Offset neu einzublenden.

Am Beispiel von `wc` werden die Änderungen im Detail gezeigt. Die Schleife, in der die Zeichen, Wort und Zeilenzählung erfolgt, hat folgendens Aussehen:

```
if ((buf = mmap (NULL, chars, PROT_READ, MAP_PRIVATE, fd, 0))
    != (char *) MAP_FAILED) { /* gemapptes File counten ... */
   register int i;
   fprintf(stderr,"%s : mapping !!!\n",program_name);
   if (!print_words) /* Worte werden nicht gezaehlt ... */
      {
      for (i = 0; i < chars; i++)
          if (buf[i] == '\n') ++lines;
      }
   else       /* Count lines, words and bytes ... */
      {
      for (i = 0; i < chars; i++)
         switch (buf[i])
            {
            case '\n':
               lines++;
               /* Fall through. */
            case '\r':
            case '\f':
            case '\t':
            case '\v':
            case ' ':
               if (in_word)
                  {
                  in_word = 0;
                  words++;
                  }
               break;
            default:
               in_word = 1;
               break;
            }
      }
   munmap(buf, chars);
```

```
}
```

Die Variable `chars` wird anfangs mit der Dateigröße initialisiert. Die Schleife wird solange durchlaufen, bis der Zähler `i` gleich `chars` ist. Wird eine Pipe eingeblendet, ist kein Wert vorhanden um als Schleifenbegrenzung zu dienen. Genauso steht kein Wert für die Längenangabe beim `mmap()` fest.

Man geht jetzt einfach so vor, daß man einen Längenwert frei wählt und damit die Variable `chars` initialisiert. Damit wird die Pipe auch mit diesem Wert eingeblendet. Es gibt zwei Aktionen, die zusätzlich ausgeführt werden müssen:

1. Wird vor Erreichen von `chars` das Pipe-Ende signalisiert, so wird der Wert in `chars` mit dem tatsächlichen Längenwert im Signal-Handler belegt. Die wirkliche Länge kann bei Eintreffen von SIGPIPE mit dem `ioctl(.,PM_GETSTAT,.)`-Aufruf ermittelt werden. Die Variable `chars` muß natürlich global definiert werden.

2. Hat die Wertabschätzung für die Datenmenge (`chars`) nicht ausgereicht, muß die Einblendung angepasst werden. Die Pipe wird neuerlich mit entsprechend angepasstem Offset eingeblendet. Dabei wird die Schleifengrenze `chars` auch angepasst.

Punkt zwei sollte aber vermieden werden, da die resultierende Programmstruktur wieder der unter Verwendung der Standard-I/O-Methode gleicht. Wobei dabei aber bedeutend weniger Iterationen erfolgen würden. Wird der Wert für die Einblendung sehr groß gewählt, kann auf diese Schleife natürlich verzichtet werden.

In `wc` mit Pipe-Mapping wurde der folgender Handler an SIGPIPE gebunden:

```
void sigh(int sig) {
  struct pstat_struct pstat;

  pstat.addr = stat_base + i;

  if (ioctl(globfd,PM_GETSTAT,&pstat) == -1) {
    perror("sigh");
    exit (-5);
  }
  if (! pstat.sigflag) {
    printf("SIGPIPE: SIGPIPE not to this pipe ..\n");
    exit (-8);
  }
  chars = pstat.tot_rbuf + i;
  fifo_cont = 0;
  signal(SIGPIPE, sigh);
  return;
```

```
}
```

Wird die mmap()-Größe ausreichend gewählt, braucht das Programm nicht verändert werden. Hier soll als Beispiel für den Fall daß die Größe nicht abschätzbar ist, die notwendige Erweiterung gezeigt werden. Das zuvor gezeigte Programmstück wird um die in 2 beschriebene Schleife und um den Aufruf der Funktion next_mmap() erweitert. In dieser Funktion wird aus der Anzahl der bisher gelesenen Daten und dem Anfangsoffset, ein neuer Offset für den neuerlichen mmap()-Aufruf berechnet.

```
if ((buf = mmap (NULL, chars, map_prot, map_flag, fd, map_off))\
    != (char *) MAP_FAILED) { /* gemapptes File counten ... */
   if ((map_mode == FIFO_MAP) && (ioctl(fd,PM_DLEV,&dlevel) == -1)) {
      do_it_traditional = TRUE;
      perror("ioctl (PM_DLEV)");
      }
   do {
      stat_base = (unsigned long) buf;
      if (!print_words) /* Worte werden nicht gezaehlt ... */
         {
         for (i = 0; i < chars; i++)
            if (buf[i] == '\n') ++lines;
         }
      else       /* Count lines, words and bytes ... */
         {
         for (i = 0; i < chars; i++)
            switch (buf[i])
               {
               case '\n':
                  lines++;
                  /* Fall through. */
               case '\r':
               case '\f':
               case '\t':
               case '\v':
               case ' ':
                  if (in_word)
                     {
                     in_word = 0;
                     words++;
                     }
                  break;
               default:
                  in_word = 1;
                  break;
               }
```

```
        }

 /* Wurden char-Bytes abgearbeitet, muss eine Pipe nocheinmal
  * eingeblendet werden. War das Pipe-Ende schon signalisiert,
  * wurde im Signalhandler das fifo_cont Flag schon geloescht. */

      if ((map_mode == FIFO_MAP) && fifo_cont) {
         mchars += chars;
         buf = next_mmap(buf, chars, map_prot, map_flag,
                         fd, map_off + mchars);
         }
      }
   while ((map_mode == FIFO_MAP) && fifo_cont);
   munmap(buf, chars);
   if (map_mode == FIFO_MAP) chars += mchars;

   }
   else  do_it_traditional = TRUE; /* Wenn mapping nicht geht ...*/
}
```

Bei diesem Beispiel trat folgendes Problem auf: Beim Übersetzen des Programmes war das Optimierungs-Flag beim Compiler aktiviert. Dabei wurde die Variablenreferenzierung von `chars` in der `for`-Schleife durch eine Konstante ersetzt, da der Compiler nicht erkannte, daß die Variable im Signal-Handler verändert werden kann. Durch die zusätzliche Klassifizierung der globalen Variable mit *volatile* wird dieses Problem beseitigt.

5.2.1 Ergebnisse

Wie die Abbildungen 30 und 31 zeigen, ist die Verbesserung des Zeitverhaltens unter Verwendung des erweiterten `mmap()` nicht so deutlich zu sehen wie im Vergleich zwischen Standard-I/O-Methode und normalem `mmap()`. Dafür sind vor allem zwei Gründe verantwortlich:

1. Der Zugriff auf den Inhalt einer Pipe mit der Standard-I/O-Methode ist bedeutend schneller als das Einlesen von externen Inhalten. Daher kann die Ersparnis nicht mehr so groß ausfallen.

   Wie im Linux-Kernelsource zu sehen ist, erfolgt das `read()` aus einer Datei über das virtuelle Filesystem, das seinerseits auf die Funktionen der betreffenden Filesystem-Implementierung zurückgreift. Weiters sind die Funktionen des Buffer-Caches und die des Swap-Systems beteiligt.

   Die Puffer der Pipe wird durch eine einzige Kernel-Page realisiert, die nicht ausgelagert werden kann. Die Adresse dieser Page kann schnell über die typspezifische Komponente der Inode-Struktur ermittelt werden. Der Zugriff auf die Inhalte ist vergleichsweise unkompliziert.

2. Die Zielsetzung der vorliegenden Implementierung des Pipe-Mappings war vorallem einmal die Machbarkeit. Der Entwurf erfolgte lediglich unter den Gesichtspunkt der Übersichtlichkeit. Überlegungen zur Optimierung der Abläufe wurden bisher noch nicht angestellt (siehe Kap.: 7).

Anders als bei den vorangegangenen Messungen, wird hier die Laufzeit von zwei Programmen erfasst (Aufruf des, in `cmd_time` erzeugten Scripts, mit `time`; siehe 5). Mit Hilfe der minimalen Implementierung von `cat` wird ein Datenstrom über eine Pipe zum Partnerprogramm (`wc` oder `grep`) aufgebaut. In `cat` wird die angegeben Datei mit `mmap()` in den Speicher eingeblendet, um zu verhindern, daß die eigentlich zu messende Zeit für die Übertragung über die Pipe im I/O-Aufwand des Einlesens der Daten untergeht.

Der Transfer der Daten über die Pipe ist in folgenden Kombinationen möglich:

1. `write()` in `cat` und `read()` in `wc` bzw. `grep`
2. `write()` in `cat` und `mmap()` in `wc` bzw. `grep`
3. `mmap()` in `cat` und `read()` in `wc` bzw. `grep`
4. `mmap()` in `cat` und `mmap()` in `wc` bzw. `grep`

Fall 2 und 3 liefern die selben Ergebnisse. Daher wurde nur Fall 2 dargestellt.

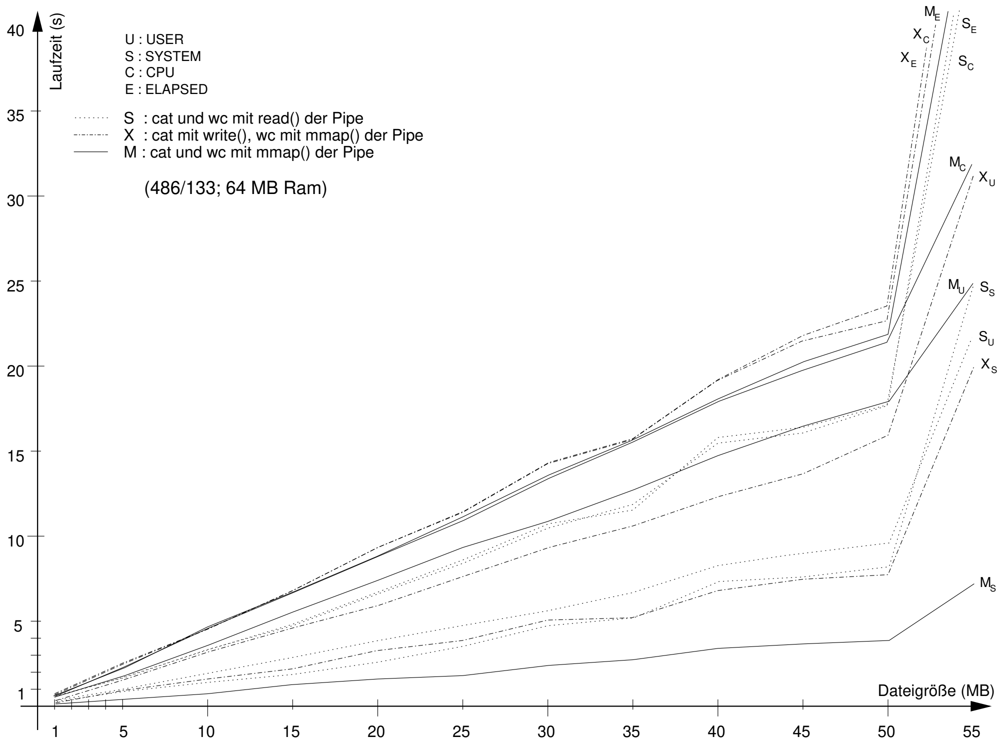

Abbildung 30: cat - wc Pipe; unter Verwendung von read() oder mmap()

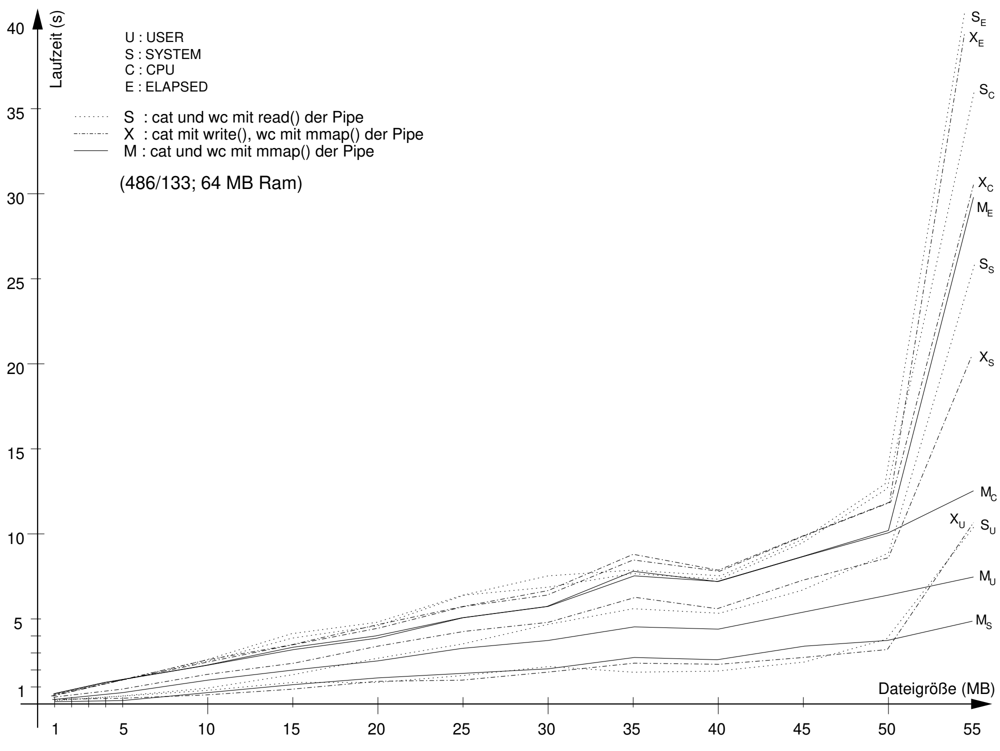

Abbildung 31: cat - grep Pipe; unter Verwendung von read() oder mmap()

6 Verwandte Arbeiten

6.1 Memory Mapping auf Kernelebene

Im Gegensatz zur API-Funktion auf Anwendungsebene, wird Memory Mapping in den Speichermodellen moderner Betriebssysteme intensiv verwendet. Es sind vorallem zwei Eigenschaften, die das Memory Mapping auf Betriebssystemebene so interessant machen. Durch das Einblenden wird es möglich auf einfache Weise auf das eingeblendete Objekt zuzugreifen. Eine einfache Schnittstelle zu den Objekten ist die Folge, die nur darin besteht, eine virtuelle Adresse anzusprechen.

Für die optimale Nutzung des vorhandenen Speichers ist es von Interesse, mehrfache Kopien von gleichen Daten in Speicher zu vermeiden. Das *Sharing* von Inhalten zwischen mehreren Prozessen oder Teilen des Betriebssystems ist mit Hilfe von Memory Mapping leicht zu realisieren. Jeder der beteiligten Partner blendet das betreffende Objekt in seinen virtuellen Adressraum ein.

Ob der `mmap()`-Aufruf auf Benutzerebene zur Verfügung steht, hängt nicht zuletzt auch davon ab, welche Rolle das Konzept des Memory Mappings im Speicherverwaltungs-Modell des Systems spielt.

BSD4.3:

Ein Speichermodell, welches wie bei BSD4.3 sehr eng an eine spezielle Hardware-Architektur (hier die VAX-MMU) gekoppelt ist, erreicht nur einen geringen Abstraktionsgrad beim Memory Management ([Vah96], [LMKQ88]). Bei BSD4.3 wird direkt auf der Ebene der Pages, Pageframes und Pagetabellen operiert. Das hat zwar wenige, kompakte Datenstrukturen und einfache Algorithmen zu deren Verwaltung zur Folge, ist aber insgesamt ein sehr wenig flexibles Design.

Damit stehen keine komplexeren Speicherobjekte als Pageframes zur Verfügung. Die Verwaltung des virtuellen Adressaumes erfolgt direkt über die Pagetabellen ohne die komplexere Beschreibung in Form von VMAs. Die BSD4.3 Definition beschreibt zwar einen `mmap()`-Aufruf, ohne ihn tatsächlich zu implementieren. Ein File Mapping wäre zwar grundsätzlich möglich, allerdings nur mit sehr viel Aufwand zu implementieren.

Das BSD4.3 Modell ist auf die optimale Verwaltung von Systemen mit wenig Hauptspeicher, großen externen Speicher und zentraler Abarbeitung vieler User abgestimmt. Die Anforderungen, die an ein Memory-Management in einem typischen Workstation-System mit relativ großem Hauptspeicher, hoher Rechenleistung und Zugriff auf verteilte Ressourcen in einem Netzwerk gestellt werden, sind von diesem Modell nicht befriedigend zu erfüllen.

Abhilfe brachten die Entwicklungen und neuen Konzepte der BSD4.4 Definition ([MBKQ96]).

System V Release 4:

Das Speichermodell, das in der zweiten großen Unix-Entwicklungslinie verwendet wird, wird diesen Anforderungen besser gerecht ([Vah96] 437ff). Das zentrale Konzept im SystemV/R4 Speichermodell ist das *Memory Mapping.* In

SVR4-Modell wird der Begriff auch auf beiden, zuvor schon angesprochenen Arten, gebraucht:

im Sinne des File-Mappings auf Anwendungsebene und
als zentrales Organisationsmodell im Kernel.

Das SVR4 VM-Modell (Virtual Memory) betrachtet den ganzen Adressraum einfach als die Menge der einzelnen Zuordnungen von Datenobjekten zu virtuellen Adressbereichen (VMAs), also als die Summe der Mappings.
Das *Speicherobjekt* (Memory Object), welches genau so ein Mapping beschreibt, ist das zentrale Element auf dem das VM-Modell aufbaut. Ein Datenobjekt ist die abstrakte Darstellung von tatsächlichen Speicherplätzen (*Backing Store*), wie Files, Swap-Bereiche, Pageframes, usw. Entsprechend dem objektorientierten Ansatz haben die Datenobjekte eine einheitliche Schnittstelle nach außen, mit einem genau festgelegtem Funktionsumfang wie z.B. eine Einheit auslesen oder zurückschreiben.

Die hardwareabhängigen Teile durch eine allgemeine, an der Funktionalität von Paging-Systemen orientierte Schnittstelle beschrieben. Damit ist das Modell einfach portierbar, indem diese unterste Schicht (*Hardware Address Translation*-Layer, HAT) auf der entsprechenden Plattform implementiert wird.

Die physischen Pages werden durch eine eigene Datenstruktur beschrieben und können dadurch flexibler verwaltet werden. Auf sie wird in den Strukturen auf höherer Abstraktionsebene verwiesen. Somit ist leicht festzustellen aus welchen Pages (Pageframes) ein bestimmtes Datenobjekt besteht, wodurch vor allem das Sharing beliebiger Datenobjekte wesentlich erleichtert wird.

So wird z.B. das Filesystem durch den *vnode*-Layer dargestellt. Ein *vnode* beschreibt eindeutig eine Datei im Filesystem und es kann leicht festgestellt werden, welche Teile der Datei zu einem Zeitpunkt im physischen Speicher präsent (quasi cached) sind. Da auch alle traditionellen I/O-Funktionen auf der VM-Funktionalität aufsetzen, greifen die normalen `read()/write()`-Aufrufe über den *vnode*-Layer auf das Dateisystem zu. Es besteht daher keine Notwendigkeit einen traditionellen Unix-Buffer-Cache zu implementieren, da diese Aufgabe implizit durch das VM-System erfüllt wird. Die gesamte File-I/O wird also intern über File-Mapping realisiert. Für die normale I/O wird eine *vnode* im Kernel eingeblendet. Soll die Datei durch einen User mittels `mmap()` eingeblendet werden, wird die *vnode* auch in den User-Adressraum abgebildet. Memory-Mapped- und `read()/write()`-Zugriff arbeiten direkt am selben Datenobjekt.

Für private Mappings kommt genauso wie für einen schnellen und effizienten `fork()`-Call das *Copy on Write*- (CoW) Konzept zum Einsatz. Eine private Kopie einer Page wird erst dann angelegt, wenn das erste Mal schreibend darauf zugegriffen wird. Ein spezielles Objekt, das *anonyme Objekt* wird dazu verwendet, alles was keine *Backing Store* repräsentiert, zu erfassen. PRIVATE-Mappings dieses Objektes werden beim CoW erzeugt, SHARED-Mappings auf das Objekt werden für die Realisierung von Shared Memory verwendet.

Die wichtige Funktion der *Shared Libraries* ist auf SVR4-VM basierten Systemen einfach durch Memory Mapping von Dateien zu realisieren.

Die Vorteile aus dem insgesamt komplexeren SVR4-VM Modell (erleichtertes Sharing, `mmap()`-Interface, Integration des Buffer-Caches, erhöhte Portabilität und flexiblere Anwendbarkeit durch objektorientiertes Design, ...) verursachen aber auch Kosten. Es müssen mehr und größere Datenstrukturen verwaltet werden, was den internen Speicherbedarf und die Verarbeitungszeit entsprechend steigen läßt. Dieser Bedarf wird aber durch die tatsächliche Steigerung der Leistungsfähigkeit moderner Rechner mehr als kompensiert.

MACH:

Intensivster Gebrauch von Memory Mapping in seinen verschiedenen Ausführungen wird auch im MACH-Kernel (Carnegie-Mellon University) gemacht. Das MACH-Konzept für die Speicherverwaltung weist auch große Ähnlichkeiten mit SVR4-VM auf, auch wenn sich die verwendete Terminologie stark unterscheidet, da es sich bei MACH um eine *Micro-Kernel*-Architektur handelt. Die Zielsetzung war, die Beschränkungen von BSD4.3 zu überwinden ([Vah96] 473ff). Besonders auf folgenden Punkten lag das Hauptaugenmerk:

- CoW- und Read/Write-Sharing von Speicher zwischen verwandten und beliebigen Prozessen; insbesondere auch zwischen Prozessen auf unterschiedlichen Rechnern.
- Memory Mapping von Dateien
- Abstimmung des Memory Management auf große, nur spärlich belegte Adressräume
- Die Möglichkeit, von User-Ebene aus, die Page-Replacement Strategien zu beeinflussen.

Folgende Begriffe bilden die elementaren Bestandteile des MACH-Designs (siehe dazu auch [Loe92]):

- *Task*: besteht aus einem Adressraum, einer Anzahl von *Ports*, und einem oder mehreren *Threads* die in ihm ablaufen.
- *Thread*: ein schedule- und ausführbarer Kontrollfluß der in einem *Task* abläuft.
- *Port*: eine geschützte *Message*-Queue. Beliebige *Tasks* können das *Senderecht* auf einem *Port* haben, jedoch hat nur genau ein *Task* das *Leserecht*.
- *Message*: eine Datenmenge eines bestimmten Types.
- *Memory Object*: Backing Store für die virtuellen Pages (entspricht den SVR4 *Data Objects* und nicht den dortigen *Memory Objects*).

Die zugreifbaren Adressen im Adressraum einer MACH-Task liegen in den *VM Objects* (das sind die Entsprechungen zu den SVR4 Memory Objects bzw. zu den Linux-VMAs), die in der *Address Map* zusammengefasst sind. Diese VM Objects repräsentieren den Zusammenhang zwischen einen *Memory Object* und einem geschlossenen Bereich virtueller Adressen. Jede Page in diesem Adressbereich verfügt über die gleichen Zugriffsrechte und Vererbungseigenschaften.

Ein *Memory Object* ist genauso wie ein SVR4 *Data Object* eine objektorientierte Abstraktion, hinter der sich jeglich Art von Speicher verbergen kann. Der Zugriff auf diese Objekte erfolgt durch Memory Mapping, wodurch sie direkt über die virtuellen Adressen angesprochen werden können. Ebenfalls wie im SVR4-Modell werden die Details der zugrundeliegenden Hardware in einer eigenen Schicht, dem *pmap-Layer*, verborgen ([RYT$^+$87]).

Die wirklichen Unterschiede zum SVR4-Modell resultieren aus der Micro-Kernel-Architektur. Der eigentliche Kernel erbringt nur ganz wenige, elementare Leistungen. Vorallem sind das die Leistungen des grundlegenden Memory Management und die Funktion des Informationstransportes. Die eigentlichen funktionalen Komponenten laufen auf Userebene außerhalb des Kernels. Das Interface zwischen diesen Teilen besteht nicht aus Funktionsaufrufen usw. Sie kommunizieren miteinander durch den Austausch von *Messages* über *Ports*. Diese *Ports* werden durch den Kernel zur Verfügung gestellt. Und hier liegt auch der bedeutendste Einsatzbereich von Memory Mapping im MACH-Konzept. Die Übertragung der *Messages* erfolgt nicht etwa indem die Daten vom Sender-Task in den Kernel und danach zum Empfänger-Task kopiert werden. Das wäre völlig unbrauchbar, da einerseits jede Menge Speicher verbraucht werden würde und andererseits die Kopiervorgänge sehr zeitaufwendig wären. Die Übermittlung der Daten erfolgt, indem sie in den Adressraum des Empfängers eingeblendet werden. Natürlich kommt auch, je nach Bedarf und Art der *Messages*, der Copy-on-Write Mechanismus zum Einsatz. Memory Mapping spielt bei diesen Abläufen in der MACH-Architektur eine ganz zentrale Rolle ([DA92]; weitere Referenzen zu MACH: [RYT$^+$], [CB93], [FBYR88]).

Multics:

Die Methode die File-I/O über direkten Speicherzugriff zu realisieren ist keine Entwicklung im Rahmen der modernen Speicherverwaltungen. Schon beim - Mitte der 60er Jahre entwickelten - Unix-Vorgängersystem *Multics* (*Multiplexed Information and Computing Service*), war dieser Weg möglich. Mehr noch, es war das ganze Modell des virtuellen Speichers auf eine direkte Abbildung eines Filesystems ausgerichtet.

Die Basis ist eine Hardware-Architektur, die Segmentierung und Paging anbietet. Es war dies das Mainframe-System *GE-635* aus dem später die *Honeywell 645* abgeleitet wurde.

Der virtuelle Adressraum eines Multics-Prozesses besteht aus (beliebig) vielen virtuellen Segmenten, die einer systemweit eindeutigen Namensgebung unterworfen sind. Die virtuellen Segmente werden direkt auf die hardwareunterstützten Sgmente der 645-MMU abgebildet.

Das Multics-Dateisystem wird direkt auf diese Organisationsstruktur des Adressraumes abgebildet. Jedes Verzeichnis des Dateisystems wird einem Segment zugeordnet, genauso wie jede Datei. Jeder Zugriff auf eine Datei erfolgt nur über einen direkten Speicherzugriff in einem Segment. Memory Mapping ist hier die einzige Methode über die alle I/O-Operationen ablaufen. Sharing von Dateien wird einfach durch gemeinsamen Zugriff auf das repräsentierende Segment erreicht.

Auch die Namensgebung in der Verwaltung der virtuellen Segmente ver-

wendet die, von den Dateisystemen her bekannten Begriffe. Die Verwaltung der Segmente erfolgt in hierarchisch angeordneten Katalogen - den *Directories*. Der Segmentname ist eine Liste von *Subnamen*, die die Position des Segments in der Katalog-Struktur wiedergibt. Die *Subnamen* werden als *Entrynames* bezeichnet, die ein Segment beschreibende Liste von *Entrynames* ist ein *Pathname*. *Entrynames* sind im Directory eindeutig, *Pathnames* sind es systemweit. Weiters gibt es zwei Arten von Directory-Einträgen: *Branch* und *Link*.
Ein *Branch* ist eine Eintrag der tatsächlich ein Segment beschreibt, also alle Attribute des Segments enthält. Ein *Link* enthält den vollständigen *Pathname* eines Segments.

Viele der in Multics entwickelten Konzepte sind bis heute in den Betriebssystemen zu finden. Genauere Beschreibungen des Multics Virtual Memory und des Multics Filesystems sind u.a. bei [Vle99], [BCD72], [Gre99] und [DN65] zu finden.

Beschreibungen des VMM in anderen Betriebssystemen: [LL82], [HG94], [MBKQ96], [Dei90], [Gra95], [Kel96]

6.2 Das *Mergemem*-Projekt

Möglichkeiten des Memory-Sharings zu finden, die durch die Speicherverwaltung nicht erfasst werden, ist das Ziel des *Mergemem*-Projekts. Viele Programme erzeugen beim Start Datenbereiche, die an sich privat sind, aber für längere Zeit unverändert bleiben. Solche Bereiche können unter Verwendung von *Copy on Write* gemeinsam verwendet werden. Mergemen ist vor allem dann sinnvoll einzusetzen, wenn man mehrere Instanzen von Interpretern oder Emulatoren (wie Java oder Prolog) verwendet. Diese Programme halten ihren Code in privaten Datenbereichen.

Allerding sollte das Sharing nur unter vertrauenswürdigen Prozessen erfolgen, da ansonsten gewisse statistische Informationen über Speicherinhalt und Speicherverwendung von unautorisierten Prozessen gesammelt und ausgewertet werden können.

Wie beim erweiterten `mmap()` wird durch Mergemem Hauptspeicher eingespart. Damit werden auch aufwendige Kopieroperationen vermieden. Das Sharing durch Mergemem ist aber für den Prozess nicht direkt sichtbar. Im Gegensatz dazu erfolgt der Einsatz des erweiterten `mmap()` explizit auf User-Ebene. Der Effizienzgewinn verteilt sich über die gesamte Benutzungszeit des eingeblendeten Stream-Objektes. Das Mergemem-Sharing bewirkt vor allem eine Beschleunigung beim Programmstart, da die Initialisierung von privaten Datenbereichen vermieden wird.

Mergemem ist ein Projekt, das am Institut für Computersprachen der TU-Wien von Philipp Richter und Philipp Reisner entwickelt wird ([RR99]).

6.3 I/O-Lite

Die Zusammenfassung aller Puffer- und Cache Aktivitäten ist das Ziel von *I/O-Lite* ([DPZ99]). Üblicherweise verwendet jedes I/O-Subsystem seinen eigenen Puffer bzw. Cache. Dazu kommen - gerade bei Server-Systemen - entsprechende

Implementierungen auf Anwendungsebene. Die Folge ist, daß Daten mehrfach in unterschiedlichen Puffern vorhanden sind. Weiters kommt es zu aufwendigen Kopieroperationen zwischen den einzelnen Systemen.

I/O-Lite ist eine zentrale Implementierung für alle I/O-Puffer. Dadurch können die Kopieroperationen vermieden werden und zusätzlich kann auch noch Hauptspeicher eingespart werden. Die Basis von I/O-Lite sind *unveränderliche (immutable)* Puffer. Der Zugriff erfolgt über die *Buffer Aggregate* Abstraktion. *Buffer Aggregates* (BA) sind Instanzen eines abstrakten Datentypes der I/O-Daten repräsentiert. Die Daten in einem BA sind nicht notwendigerweise kontinuierlich. Intern sind sie als Paar $< pointer, length >$ dargestellt. Die Operationen auf BAs, wie *truncate, append, prepend,* usw. erfolgen in Form von Pointermanipulation. Damit wird erreicht, daß die zugrundeliegenden, *unveränderlichen* I/O-Puffer auf Ebene der BA-Abstraktion *veränderbar* werden.

Indem jedes I/O-Subsystem den eigenen Puffer/Cache auf Basis der BAs realisiert, können Mehrfachkopien von I/O-Daten vermieden werden. Kopieroperationen von I/O-Daten werden auf das Kopieren von BAs reduziert. Die Übergabe der BAs zwischen den Subsystemen erfolgt per Wertübergabe, intern werden die I/O-Puffer aber per Referenz übergeben. Für weitere Optimierung steht der Zugriff auf die I/O-Buffer auch auf Anwendungsebene zur Verfügung. Wie in [DPZ99] angegeben, kann durch die Verwendung von I/O-Lite in einem Web-Server System eine Effizienzsteigerung von 40-80% erreicht werden.

Die Zielsetzung von I/O-Lite entspricht grundsätzlich der, die der Verwendung von Memory Mapping zugrunde liegt: Vermeidung kostspieliger Kopieroperationen von Hauptspeicherinhalten. Der Ansatz ist bei den Unified I/O-Buffers allerdings wesentlich allgemeiner gehalten, wodurch bedeutend größere Wirkung erzielt werden kann. Der Aspekt der einfacheren Programme, wie er beim Einsatz von `mmap()` ebenfalls zum Tragen kommt, ist bei I/O-Lite nicht gegeben.

7 Ausblick

Nach der Anwendung des erweiterten `mmap()` auf Pipes ist es vor allem die Anpassung an Sockets, die von besonderem Interesse ist. Zum Unterschied zu den Pipes, müssen bei den Sockets jedoch die unterschiedlichen Protokoll-Familien, Socket-Typen und Protokolle berücksichtigt werden. Alle diese Fälle haben besondere Eigenschaften, deren Auswirkungen auf einen eingeblendeten Socket zu definieren sind. Das erweiterte `mmap()` für Sockets ist ebenfalls beispielhaft unter Linux zu implementieren.

An der Implementierung des erweiterten `mmap()` ist vor allem noch die Swaping-Funktionalität zu ergänzen. Der grundsätzliche Lösungsweg wurde bereits im Kapitel 4.4.2 aufgezeigt. Das eigentliche Problem ist die fehlenden Möglichkeit, Kernelpages auszulagern. Dafür sind zwei Ansätze denkbar:

- Man bedient sich eines Tricks und legt dynamisch einen VMA im Adressbereich des Prozesses an, der im `read()/write()`-Aufruf indirekt für den

Page-Fault verantwortlich ist. Über diesen Dummy-VMA erfolgt dann die normale Page-Fault Behandlung des VMMs im User-Segment.

- Sauberer ist die Alternative dazu. Man erweitert das eingeschränkte virtuelle Speichermanagment für das Kernel-Segment. Damit wird die Möglichkeit geschaffen, Kernelpages gezielt für das Swaping freizugeben. Der Ausbau des Kernel-Segment VMMs erscheint auch unter dem Licht der Vergrößerung des Kernel-Segments auf 64bit Architekturen als sinnvoll und wichtig.

Wie in den Messergebnissen zu sehen war, hält sich der Effizienzgewinn beim Memory Mapping von Pipes in Grenzen. Das liegt vor allem daran, daß die vorliegende Implementierung noch keinen Optimierungsüberlegungen unterworfen war. Vor allem in der Art, wie der interne Puffer für den Pipe-Inhalt aufgebaut und verwaltet wird, ist ineffizient. Dabei wird das Verschieben des Gültigkeitsfensters über den Pipe-Inhalt nachgebildet. Das hat zur Folge, daß jedes tatsächliche Commit zu einem Puffer-Shift führt, bei dem alle Puffer-Page Repräsentationen (`struct pbpage`) kopiert werden. Der Puffer-Page-Pool sollte alls Ring-Puffer organisiert werden, womit bei einem Commit lediglich eine Pointer-Inkrementierung vorzunehmen wäre.

Die Ring-Struktur würde auch die Möglichkeit der dynamischen Größenänderung des Puffers erleichtern. Diese Funktionalität ist für den praktischen Einsatz sehr wichtig, da damit eine Anpassung an den jeweiligen Anwendungsfall möglich wird.

Für die Messungen wurden lediglich sehr einfache Anwendungsprogramme auf die Verwendung des erweiterten `mmap()` umgestellt. Interessant wäre die Umstellung einer großen Applikation wie zb. der Kommunikation zwischen Präcompiler `cpp` und C-Compiler `gcc` beim GNU-C-Compiler.

Das Prinzip das bei der Implementierung des erweiterten `mmap()` verwendet wurde, würde sich auch bestens zur Realisierung von SysV-Shared-Memory eignen. Die im Linux-Kernel vorhandene Implementierung stammt aus einer sehr frühen Kernel-Version, die noch über ein eingeschränktes virtuelles Speichermanagement verfügt hat. Die Folge ist, daß im Kernel-Code sehr viele Ausnahmebehandlungen für Shared-Memory vorhanden sind, da sich das Modell nicht in das objektorientierte Konzept im aktuellen VMM einfügt. Durch die Realisierung des Shared-Memory in Form der Repräsentation über Inodes könnten diese Inkonsistenzen beseitigt werden. Zusätzlich würden die Beschränkungen für die Gesamtanzahl von SysV-IPC-Objekten im System beseitigt werden.

Die Änderungen am Memory Management (minimierendes Verhalten und Byte-Granularität) könnten als zusätzliche Varianten zur herkömmlichen Verwaltungs-Strategie angeboten werden. Denkbar wäre, durch zusätzliche Attribute beim `mmap()`-Aufruf, die zu verwendende Strategie zu spezifizieren.

8 Zusammenfassung

Praktisch jede I/O-Operation erfolgt unter Unix nach dem Standard-I/O-Schema. Diese Normierung ist für den Anwendungsprogrammierer sehr angenehm, da man sich nicht um Datails der Eigenschaften des bearbeiteten Objektes kümmern muß. Egal ob man aus einer Datei, einer Netzwerkverbindung, oder einem Geräte-Treiber liest, die Art des Zugriffes ist immer gleich. Die verwendeten Programmstrukturen folgen demselben Schema. Ein Objekt wird geöffnet (`open()`, `pipe()`, `socket()`), auf den erzeugten File-Deskriptor erfolgen die geblockten I/O-Operationen (`read()/write()`) in einer Iteration und am Ende wird der Deskriptor wieder geschlossen (`close()`).

Nach diesem Schema erfolgt der Zugriff rein sequentiell. Das ist für alle Arten von Objekten möglich. Bestimmte Objekte - wie reguläre Dateien - erlauben auch einen wahlfreien Zugriff. Dabei wird die Funktion `lseek()` für die Positionierung im I/O-Objekt verwendet.

Die üblichen Programme zur Dateibearbeitung werden wahlweise direkt auf reguläre Dateien oder auf Datenströme angewendet. Diese Datenströme werden unterschiedlich realisiert. Sie können als Pipes, Netzwerkverbindungen, Terminals, usw. vorliegen. Gemeinsam ist ihnen die Eigenschaft characterorientiert zu sein. Daher ist kein wahlfreier Zugriff darauf möglich. Um in den Programmen keine aufwendigen Fallunterscheidungen implementieren zu müssen, bleibt die Möglichkeit des wahlfreien Zugriffs meist ungenutzt. Wird ein solcher benötigt, werden entsprechend große, lokale Puffer verwendet.

Aufgrund der internen Abläufe und der verwendeten Programmstrukturen kommt es bei der Standard-I/O-Methode zu Duplizierungen von Hauptspeicherinhalten. Diese Kopiervorgänge sind zeit- und speicheraufwendig.

Eine alternative Methode der File-I/O besteht im Memory Mapping. Dabei wird eine reguläre Datei mit Hilfe des System-Calls `mmap()` in den virtuellen Adressraum des Prozesses zum Teil oder zur Gänze eingeblendet. Der Zugriff erfolgt danach über eine direkte Adressreferenzierung.

Diese Methode ermöglicht einfachere Programmstrukturen und vermeidet die aufwendigen Kopieroperationen. Durch die beschränkte Anwendbarkeit würde die Verwendung im Normalfall lediglich zu einer Verkomplizierung der Programme führen. Daher wird die Methode nur selten verwendet.

Eine Ausweitung der Anwendbarkeit auf die wichtigsten characterorientierten Objekte (Pipe, FIFO, Socket, Terminal) unter Beibehaltung des APIs das für eingeblendete, reguläre Dateien gilt, könnte die Akzeptanz entscheidend erhöhen.

Wird eine Datei mit `mmap()` in den Speicher eingeblendet, werden vor allem zwei Werte benötigt. Die Länge des Abschnittes der eingeblendet werden soll und der Offset an dem dieser Abschnitt beginnt.

Dabei stellen vorallem zwei Eigenschaften der Stream-Objekte ein Problem dar. Stream-Objekte verfügen über keine definitive Größe. Sie beschreiben einen - theoretisch - unendlichen Datenstrom. Daher stellt sich die Frage, wie man zu dem, für das Einblenden benötigten Längenwert kommt und wie sich das Ende des Objekts programmtechnisch erfassen lässt.

Weiters stellt sich die Frage, wann eine Information, die in das Objekt geschrieben wurde, ausgelesen werden kann.

Aus den vorgegebene Verhalten von eingeblendeten, regulären Dateien und den spezifischen Eigenschaften von Stream-Objekten ergibt sich folgendes Bild. Durch die Möglichkeiten, die große, virtuelle Adressräume auf 64 bit Architekturen bieten, können beim Einblenden einfach sehr große Längenwerte angegeben werden. Damit kann das Objekt grundsätzlich erfasst werden.

Ein Stream-Objekt, auf das eine `mmap()`-Operation angewendet wird, ändert seinen Zustand. Damit beginnt intern eine Mitzählung des Datendurchflusses um die Größe und damit den Offset bestimmbar zu machen. Weiters beginnt eine interne Pufferung der Daten in ausgedehnterer Form um den wahlfreien Zugriff auf den Inhalt zu ermöglichen.

Das Objekt wird in unterschiedliche Zonen geteilt. Ein Schreibbereich, ein Lesebereich und ein Bereich der nicht mehr zugänglich ist. Die Größe der Bereiche hängt mit dem Ausmaß der internen Pufferung zusammen. Beim Einblenden sind, je nach Modus, gewisse Mindestoffsets einzuhalten. Am Ende der Lesezone kann eine Schreib-Einblendung beginnen, am Ende der ungültigen Zone kann eine Leseeinblendung beginnen. Offsets die in der abgearbeiteten Zone liegen werden nicht akzeptiert.

Die Daten im Objekt nehmen entsprechend der Zonen mehrere Zustände an: der Bereich, der zum Schreiben eingeblendet wird, stellt den *unfertigen* Bereich dar. Diese Daten sind Lesend noch nicht erreichbar. Durch ein *Commit* werden sie dem Schreibzugriff entzogen und gehen in den Zustand *gültig* über. Damit sind sie lesbar. Ein *Commit* auf gültige Daten bedeutet, daß sie nicht mehr weiter benötigt werden. Sie ändern den Zustand und sind *abgearbeitet*. Sie sind nur noch für die Größe des Objektes relevant, aber nicht mehr zugreifbar.

Das *Commit* kann explizit durch den Aufruf `msync()` oder implizit durch eine Auto-Commit-Funktionalität erfolgen.

Da der Standard die Bedeutung eines `msync()` auf Objekte ungleich reguläre Dateien undefiniert lässt, kann dieser Aufruf die Commit-Funktion übernehmen. Dadurch kommt es zu keinem Konflikt mit der Anwendung auf die Einblendung einer regulären Datei.

Tatsächlich kommt es durch die Erweiterung auf Stream-Objekte zu keinen wesentlichen Veränderungen im `mmap()`-API. Folgende Erweiterungen werden nötig:

1. Zwei weitere Werte für den Parameter *flag* werden benötigt, um Protektion-Konflikte zu lösen: MAP_RDSTREAM und MAP_WRITESTREAM.

2. Eine zusätzliche Schleife um eine Fehleinschätzung der Objektgröße abzufangen. Tritt dieser Fall auf, wird das Objekt mit geändertem Offset neu eingeblendet.

3. Aufruf von `msync()` um das Commit explizit zu veranlassen. Das kann durch Verwendung der Auto-Commit-Funktion vermieden werden.

4. Ein Signal-Handler an SIGPIPE um das Ende des Datenstroms programmtechnisch zu erfassen.

5. `ioctl()`-Aufrufe neuer Funktionen um aktuelle Informationen über das Objekt zu erhalten bzw. um Werte zu setzen.

Das Modell für das erweiterte `mmap()` hat allerdings wesentliche Konsequenzen auf das Memory Management. Es wird der Umstieg der Verwaltung der virtuellen Speicherbereiche von *Pagegranularität* auf *Bytegranularität* verlangt. Weiters wird die Änderung der *maximierenden* Strategie beim Anpassen von Anwendungsbereichen der Operationen auf VMAs benötigt. Diese Anpassungen müssen einer *minimierenden* Strategie folgen.

Darin ist das größte Problem für eine Implementierung des Modells zu sehen. Die Änderungen lassen sich allerdings in einer Form ausführen, die bestehende Applikationen in keiner Weise beeinflussen, wie an der praktischen Anwendung auf das Linux Memory Management zu sehen ist.

Im Rahmen der praktischen Anwendungen wurden einige einfache Anwendungen auf I/O unter Verwendung von `mmap()` umgestellt. Diese Anwendungen wurden auch für das erweiterte `mmap()` adaptiert. Die Unterschiede der beiden Varianten sind gering. Die Ermittlung des Längenwertes für den `mmap()`-Aufruf erfolgt im einen Fall mit Hilfe eines `ioctl()`-Aufrufes. Beim erweiterten `mmap()` wird er frei (möglichst groß) gewählt. Für die Endbehandlung ist beim erweiterten `mmap` auch noch ein Signal-Handler an SIGPIPE zu binden.

Die Messungen der Laufzeiten haben für den Vergleich zwischen Standard-I/O und normalem `mmap()` das erwartete Bild geliefert. Durch die Verwendung von `mmap()` wird auf allen betrachteten Plattformen die Systemzeit reduziert. Damit kommt es zu einer Reduktion der CPU-Zeit und damit auch zu einer verringerten Gesamt-Ausführungszeit.

Der Performance-Zuwachs beim erweiteten `mmap()` ist nicht so deutlich zu erkennen. Der Grund ist aber nicht prinzipieller Natur, sondern es liegt daran, daß bei der Implementierung noch keine Optimierungen vorgenommen wurden.

Anhang A (Messtabellen - mmap() vrs. read())

Laufzeitmessungen der Programme `wc` und `grep` unter Verwendung von `read()` und `mmap()` auf FreeBSD. Die einzelnen Messwert-Zeilen enthalten jeweils den angenäherten Median der durchgeführten Messungen (je 10 Programmläufe). Die vollstaendigen Tabellen stehen als Ascii-Text-Dateien zur Verfügung.

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.21	0.09	0.31	0.33
5	0.96	0.37	1.34	1.36
10	1.87	0.84	2.66	2.78
15	5.66	2.78	8.48	10.61
20	6.70	2.89	9.65	9.84
25	7.52	3.53	11.05	11.38
30	8.44	3.70	12.14	12.89
35	9.50	4.22	13.73	14.15
40	10.42	5.57	14.90	30.89
45	11.35	4.63	16.04	33.38
50	12.26	5.33	17.54	36.12
55	13.25	5.65	19.30	39.33

Tabelle 3: Messwerte wc (read()) auf FreeBSD

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.24	0.04	0.28	0.31
5	1.09	0.08	1.18	1.20
10	2.13	0.13	2.27	2.33
15	6.32	0.39	6.71	6.84
20	7.37	0.43	7.82	7.95
25	8.49	0.42	8.93	9.15
30	9.57	0.45	10.03	10.22
35	10.57	0.59	11.19	12.06
40	11.74	1.22	13.00	20.03
45	12.86	2.26	15.13	30.77
50	14.02	2.43	16.44	33.82
55	14.96	2.06	17.06	37.38

Tabelle 4: Messwerte mwc (mmap()) auf FreeBSD

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.05	0.13	0.19	0.20
5	0.20	0.43	0.65	0.68
10	0.38	0.85	1.13	1.22
15	0.66	1.39	2.07	2.16
20	0.89	1.82	2.57	2.74
25	0.91	2.07	2.98	3.04
30	1.14	2.44	3.61	3.72
35	1.35	3.03	4.45	4.58
40	1.63	3.60	5.22	5.37
45	1.86	3.94	5.91	6.10
50	1.83	4.32	6.21	6.54
55	1.78	4.59	6.48	21.87

Tabelle 5: Messwerte grep (iterativ read()) auf FreeBSD

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.07	0.15	0.22	0.24
5	0.23	0.61	0.84	0.87
10	0.44	1.28	1.72	1.84
15	0.65	2.03	2.67	2.77
20	0.85	2.67	3.52	3.63
25	0.96	2.05	3.03	3.24
30	1.08	2.60	3.71	3.96
35	1.25	2.87	4.10	4.19
40	1.69	3.57	5.26	5.37
45	1.78	3.98	5.62	5.84
50	1.84	4.30	6.12	6.47
55	2.06	4.42	6.35	22.41

Tabelle 6: Messwerte grep (read() slurp) auf FreeBSD

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.06	0.06	0.12	0.14
5	0.24	0.08	0.33	0.35
10	0.44	0.14	0.59	0.62
15	0.67	0.18	0.86	0.89
20	0.90	0.19	1.12	1.16
25	1.07	0.29	1.38	1.41
30	1.32	0.32	1.64	1.68
35	1.51	0.38	1.90	1.94
40	1.72	0.41	2.18	2.23
45	1.95	0.48	2.43	2.47
50	2.20	0.51	2.74	3.35
55	2.46	1.26	3.65	10.34

Tabelle 7: Messwerte grep (mmap()) auf FreeBSD

Laufzeitmessungen der Programme `wc` und `grep` unter Verwendung von `read()` und `mmap()` auf SCO. Die einzelnen Messwert-Zeilen enthalten jeweils den angenäherten Median der durchgeführten Messungen (je 10 Programmläufe). Die vollstaendigen Tabellen stehen als Ascii-Text-Dateien zur Verfügung.

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.28	0.14	0.41	0.43
5	1.70	0.76	2.43	3.92
10	3.36	1.47	4.86	7.49
15	4.86	2.21	7.06	12.28
20	5.56	1.45	7.01	8.69
25	6.90	1.66	8.62	9.64
30	8.23	1.90	10.14	11.15
35	9.57	2.10	11.67	12.86
40	10.82	2.34	13.08	15.01
45	12.59	3.25	15.83	20.44
50	14.39	4.04	18.45	25.23
55	16.34	4.88	21.16	30.10

Tabelle 8: Messwerte wc (read()) auf SCO

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.25	0.05	0.31	0.32
5	1.19	0.07	1.26	1.27
10	2.38	0.06	2.44	2.45
15	3.56	0.07	3.64	3.64
20	4.74	0.09	4.83	4.83
25	5.94	0.08	6.02	6.02
30	7.11	0.09	7.20	7.21
35	8.31	0.59	8.40	8.94
40	11.73	5.87	17.59	38.11
45	13.34	7.47	20.83	50.29
50	14.95	8.50	23.51	60.72
55	16.26	9.23	25.72	80.98

Tabelle 9: Messwerte mwc (mmap()) auf SCO

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.04	0.08	0.12	0.13
5	0.18	0.78	1.01	4.02
10	0.37	1.43	1.84	7.66
15	0.51	2.16	2.66	12.15
20	0.59	1.45	2.07	5.23
25	0.72	1.68	2.41	4.95
30	0.86	1.87	2.72	5.28
35	0.98	2.10	3.04	5.93
40	1.11	2.26	3.44	7.53
45	1.22	3.19	4.47	12.67
50	1.57	3.87	5.44	18.32
55	1.73	4.71	6.45	22.83

Tabelle 10: Messwerte grep (iterativ read()) auf SCO

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.04	0.12	0.15	0.16
5	0.13	0.98	1.12	3.90
10	0.25	1.94	2.20	7.77
15	0.38	2.89	3.25	12.45
20	0.50	3.67	4.17	15.64
25	0.61	4.80	5.42	19.27
30	0.74	5.69	6.49	23.18
35	0.90	6.69	7.68	34.31
40	1.45	10.51	11.92	106.26
45	1.43	11.22	12.72	119.28
50	1.75	13.99	15.74	156.46
55	1.82	15.20	17.21	166.39

Tabelle 11: Messwerte grep (read() slurp) auf SCO

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.04	0.05	0.09	0.10
5	0.15	0.05	0.19	0.20
10	0.25	0.07	0.32	0.32
15	0.37	0.07	0.44	0.45
20	0.50	0.05	0.57	0.57
25	0.61	0.08	0.69	0.70
30	0.74	0.09	0.83	0.83
35	0.91	0.11	0.99	1.03
40	1.47	6.75	8.28	43.89
45	1.66	8.26	9.87	54.98
50	1.84	8.91	10.65	74.00
55	1.94	9.67	11.63	95.09

Tabelle 12: Messwerte grep (mmap()) auf SCO

Laufzeitmessungen der Programme `wc` und `grep` unter Verwendung von `read()` und `mmap()` auf Sun Solaris. Die einzelnen Messwert-Zeilen enthalten jeweils den angenäherten Median der durchgeführten Messungen (je 10 Programmläufe). Die vollstaendigen Tabellen stehen als Ascii-Text-Dateien zur Verfügung.

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.09	0.06	0.14	0.99
5	0.30	0.11	0.42	1.53
10	0.53	0.21	0.73	2.24
15	0.77	0.29	1.05	2.93
20	0.98	0.33	1.31	3.34
25	1.26	0.44	1.72	4.71
30	1.54	0.56	2.09	5.11
35	1.74	0.59	2.33	5.55
40	2.03	0.69	2.71	6.56
45	2.26	0.78	3.03	6.71
50	2.55	0.96	3.44	8.19
55	2.81	0.94	3.75	9.63

Tabelle 13: Messwerte wc (read()) auf Sun Solaris

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.08	0.06	0.15	0.17
5	0.44	0.05	0.49	1.25
10	0.84	0.08	0.92	1.68
15	1.28	0.08	1.36	2.12
20	1.70	0.08	1.78	1.83
25	2.12	0.08	2.21	2.24
30	2.56	0.09	2.65	3.41
35	2.94	0.11	3.04	3.21
40	3.36	0.12	3.48	4.26
45	3.81	0.30	4.10	7.77
50	4.27	0.30	4.57	11.75
55	4.67	0.33	5.01	11.43

Tabelle 14: Messwerte mwc (mmap()) auf Sun Solaris

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.02	0.05	0.08	0.98
5	0.06	0.09	0.15	0.81
10	0.11	0.19	0.30	2.23
15	0.13	0.26	0.41	2.84
20	0.17	0.35	0.56	3.68
25	0.20	0.42	0.66	4.54
30	0.26	0.54	0.80	4.36
35	0.25	0.59	0.87	5.09
40	0.32	0.64	0.96	5.86
45	0.36	0.74	1.12	6.54
50	0.43	0.89	1.32	8.20
55	0.43	0.93	1.39	9.25

Tabelle 15: Messwerte grep (iterativ read()) auf Sun Solaris

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.03	0.05	0.09	0.86
5	0.07	0.17	0.25	1.27
10	0.12	0.31	0.42	0.70
15	0.17	0.45	0.61	1.66
20	0.22	0.52	0.74	2.12
25	0.28	0.78	1.04	8.19
30	0.35	1.18	1.50	18.38
35	0.36	1.51	1.87	23.13
40	0.42	1.77	2.16	26.66
45	0.49	2.47	3.03	57.49
50	0.60	2.77	3.40	71.28
55	0.64	3.30	3.96	84.41

Tabelle 16: Messwerte grep (read() slurp) auf Sun Solaris

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.03	0.04	0.07	0.29
5	0.07	0.06	0.13	0.88
10	0.13	0.06	0.18	0.94
15	0.17	0.07	0.24	1.01
20	0.22	0.08	0.31	1.05
25	0.28	0.10	0.37	1.14
30	0.32	0.12	0.44	1.19
35	0.35	0.15	0.49	1.27
40	0.40	0.15	0.56	1.34
45	0.52	0.32	0.88	8.12
50	0.57	0.32	0.90	9.91
55	0.58	0.38	0.99	12.62

Tabelle 17: Messwerte grep (mmap()) auf Sun Solaris

Laufzeitmessungen der Programme `wc` und `grep` unter Verwendung von `read()` und `mmap()` auf Linux. Die einzelnen Messwert-Zeilen enthalten jeweils den angenäherten Median der durchgeführten Messungen (je 10 Programmläufe). Die vollstaendigen Tabellen stehen als Ascii-Text-Dateien zur Verfügung.

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.29	0.10	0.38	0.39
5	1.14	0.35	1.50	1.51
10	2.30	0.62	2.93	2.95
15	3.47	0.84	4.31	4.32
20	4.59	1.18	5.77	5.24
25	5.76	1.59	7.36	7.41
30	6.89	1.73	8.51	8.63
35	7.98	1.98	9.89	10.25
40	9.23	2.56	11.70	11.91
45	10.49	3.16	13.59	13.93
50	11.69	3.60	15.32	15.52
55	26.98	10.22	37.20	51.49

Tabelle 18: Messwerte wc (read()) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.34	0.06	0.41	0.41
5	1.55	0.06	1.61	1.61
10	3.00	0.09	3.09	3.09
15	4.50	0.09	4.59	4.60
20	5.98	0.11	6.09	6.12
25	7.44	0.14	7.60	7.60
30	8.96	0.14	9.09	9.11
35	10.40	0.17	10.58	10.60
40	11.89	0.17	12.09	12.11
45	13.38	0.20	13.60	13.62
50	14.86	0.24	15.11	15.14
55	16.37	0.74	17.07	26.23

Tabelle 19: Messwerte mwc (mmap()) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.07	0.11	0.19	0.19
5	0.22	0.34	0.57	0.58
10	0.38	0.64	0.99	1.00
15	0.70	1.04	1.70	1.81
20	0.78	1.33	2.11	2.11
25	1.10	1.82	2.92	2.91
30	1.32	2.19	3.56	3.57
35	1.08	1.98	3.07	3.07
40	1.58	2.60	4.13	4.15
45	1.52	2.79	4.26	4.30
50	5.59	10.43	16.44	43.69
55	8.26	10.69	17.39	48.03

Tabelle 20: Messwerte grep (iterativ read()) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.09	0.15	0.25	0.24
5	0.24	0.54	0.80	0.81
10	0.47	1.05	1.52	1.53
15	0.68	1.55	2.24	2.26
20	0.90	2.05	2.95	2.96
25	1.12	5.38	6.50	13.59
30	1.37	10.26	11.60	29.28
35	1.65	11.55	13.18	34.79
40	1.87	13.44	15.30	39.82
45	2.12	15.13	17.39	45.35
50	2.25	16.84	18.99	50.29
55	2.22	31.81	34.03	129.22

Tabelle 21: Messwerte grep (read() slurp) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.10	0.05	0.15	0.15
5	0.27	0.07	0.35	0.37
10	0.50	0.09	0.59	0.65
15	0.72	0.12	0.83	0.89
20	0.92	0.13	1.07	1.14
25	1.15	0.12	1.27	1.35
30	1.37	0.15	1.52	1.60
35	1.64	0.16	1.78	1.89
40	1.79	0.20	1.98	2.11
45	2.03	0.21	2.24	2.37
50	2.25	0.23	2.49	2.64
55	2.57	1.04	3.77	25.39

Tabelle 22: Messwerte grep (mmap()) auf Linux

Anhang B (Messtabellen - Erweitertes mmap())

Laufzeitmessungen der Programme `wc` und `grep` in Kombination mit cat unter Verwendung von read()/write() und mmap() auf einer Pipe. Ablauf auf dem modifizierten Linux-Kernel. Die einzelnen Messwert-Zeilen enthalten jeweils den angenäherten Median der durchgeführten Messungen (je 10 Programmläufe). Die vollstaendigen Tabellen stehen als Ascii-Text-Dateien zur Verfügung.

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.23	0.20	0.43	0.44
5	0.97	0.71	1.65	1.67
10	1.88	1.26	3.13	3.14
15	2.77	1.90	4.72	4.80
20	3.89	2.62	6.51	6.57
25	4.68	3.54	8.35	8.39
30	5.57	4.86	10.39	10.51
35	6.68	5.04	11.74	11.57
40	8.14	7.15	15.29	15.43
45	8.88	7.41	15.98	16.18
50	9.61	8.08	17.66	17.60
55	21.66	24.60	45.10	57.35

Tabelle 23: Messwerte wc (cat und wc mit read()) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.34	0.20	0.56	0.56
5	1.54	0.80	2.34	2.36
10	3.06	1.51	4.58	4.59
15	4.52	2.16	6.74	6.74
20	5.98	3.04	9.01	9.05
25	7.54	3.84	11.34	11.37
30	9.09	5.00	14.12	14.15
35	10.57	5.08	15.59	15.62
40	12.11	6.86	18.98	19.00
45	13.58	7.40	21.25	21.41
50	14.91	7.63	22.53	23.17
55	31.00	19.94	50.72	57.98

Tabelle 24: Messwerte wc (wc mmap() der Pipe) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.40	0.14	0.55	0.56
5	1.86	0.42	2.28	2.28
10	3.64	0.78	4.45	4.46
15	5.40	1.14	6.53	6.54
20	7.32	1.51	8.83	8.84
25	9.11	1.75	10.93	11.06
30	10.81	2.39	13.21	13.30
35	12.64	2.61	15.31	15.34
40	14.54	3.33	17.90	17.92
45	16.29	3.53	19.85	20.06
50	17.92	3.76	21.79	21.88
55	24.85	7.04	31.80	50.49

Tabelle 25: Messwerte wc (cat und wc mmap() der Pipe) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.14	0.14	0.37	0.37
5	0.41	0.41	1.28	1.29
10	0.87	0.93	2.66	2.68
15	1.12	1.77	3.92	4.08
20	1.12	2.76	4.63	4.68
25	1.85	3.49	6.31	6.32
30	2.05	4.87	6.90	7.43
35	1.82	5.53	7.65	7.81
40	1.84	5.38	7.43	7.48
45	2.31	6.64	9.14	9.32
50	3.87	8.85	12.69	12.80
55	10.10	25.89	35.99	53.16

Tabelle 26: Messwerte grep (cat und grep mit read()) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.13	0.22	0.35	0.35
5	0.35	0.88	1.23	1.24
10	0.66	1.72	2.41	2.44
15	0.94	2.31	3.33	3.34
20	1.22	3.32	4.59	4.73
25	1.57	4.16	5.77	5.78
30	1.74	4.74	6.37	6.77
35	2.34	6.07	8.42	8.81
40	2.25	5.56	7.77	7.88
45	2.79	7.02	9.77	9.79
50	3.11	8.33	11.42	11.48
55	10.12	20.11	30.33	51.67

Tabelle 27: Messwerte grep (grep mmap() der Pipe) auf Linux

Größe (MB)	USER	SYSTEM	CPU	ELAPSED
1	0.19	0.14	0.32	0.32
5	0.62	0.46	1.11	1.11
10	1.31	0.76	2.10	2.11
15	2.00	1.05	3.06	3.12
20	2.42	1.45	3.87	3.91
25	3.12	1.79	4.96	4.97
30	3.61	2.07	5.69	5.70
35	4.64	2.85	7.45	7.80
40	4.49	2.61	7.17	7.17
45	5.35	3.44	8.79	8.80
50	6.13	3.83	10.00	10.13
55	7.34	4.85	12.33	29.93

Tabelle 28: Messwerte grep (cat und grep mmap() der Pipe) auf Linux

Literatur

[Bac87] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1987.

[BBD+97] M. Beck, H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux-Kernel Programmierung, Algorithmen u. Strukturen der Ver. 2.0, 4. akt. Auflage.* Addison Wesley Longman Verlag GmbH., 1997.

[BCD72] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[CB93] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In ACM, editor, *14th ACM Symposium on Operating Systems Principles, Ashville, NC, USA, December 5–8, 1993*, volume 27(5) of *Operating Systems Review*, pages 120–133, New York, NY 10036, USA, Dec 1993. ACM Press.

[CE85] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[DA92] R. W. Dean and F. Armand. Data movement in kernelized systems. In USENIX, editor, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 243–261, Berkeley, CA, USA, 1992. USENIX.

[Dei90] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, 2nd edition, 1990.

[Den97] Denning. Virtual Memory. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook, CRC Press, in cooperation with ACM, 1997.* 1997.

[DN65] R. C. Daley and P. G. Neuman. A General Purpose File System for Secondary Storage. In *Proc. FJCC, AFIPS*, volume 27, 1965.

[Dor76] Robert Doran. Virtual Memory. *Computer*, 9(10):27–37, October 1976.

[DPZ99] P. Druschel, V. S. Pai, and W. Zwaenepoel. I/O-Lite: A Unified I/O Buffering and Caching System. *OSDI 99*, 1999.

[FBYR88] Alessandro Forin, Joseph Barrera, Michael Young, and Richard Rashid. *Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach.* Number CMU-CS-88-185. Department of Computer Science, Carnegie Mellon University, 1988.

[Gra95] John R. Graham. *Solaris 2.x: Internals and Architecture.* McGraw-Hill, New York, NY, USA, 1995.

[Gre99] P. Green. Multics Virtual Memory - Tutorial and Reflections. *ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html*, 1999.

[HG94] D. Hecker and H-J. Goetz. *OS/2 Integrationsplattform.* Franzis-Verlag GmbH, 1994.

[Joh97] Michael K. Johnson. Linux Kernel Hackers' Guide, 1997. http://metalab.unc.edu/mdw/LDP/khg/HyperNews/get/khg.html.

[Kel96] David A. Kelly. *AIX/6000 Internals and Architecture.* McGraw-Hill, New York, NY, USA, 1996.

[Ker00] Kernel.org. The Linux Kernel Archives, 2000. http://www.kernel.org/.

[LL82] H. M. Levy and P. H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *Computer*, 15(3):35–41, March 1982.

[LMKQ88] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1988.

[Loe92] K. Loepere. Mach 3 kernel principles. Technical Report MK67, Open Software Foundation and Carnegie Mellon University, 1992.

[MBKQ96] McKusick, Bostic, Karels, and Quaterman. *The Design and Implementation of the 4.4 BSD Operation System.* Addison Wesley, Reading, 1996.

[MJLF84] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[Ope98] The Open Group. The Single UNIX Specification; Version 2 - 6 Vol Set for UNIX 98, 1998. http://www.unix-systems.org/online.html.

[Pom99] O. Pomerantz. Linux Kernel Module Programming Guide, 1999. http://metalab.unc.edu/mdw/LDP/lkmpg/mpg.html.

[POS90] *System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[RR99] P. Richter and P. Reisner. The Mergemem Project, 1999. http://das.ist.org/mergemem/.

[Rus99] David A. Rusling. The Linux Kernel, 1999. http://metalab.unc.edu/mdw/LDP/tlk/tlk.html.

[RYT$^+$] R. Rashid, M. Young, A. Tevanian, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. A Unix Interface for Shared Memory and Memory Mapped Files under Mach. Department of Computer Science, Carnegie Mellon University. http://almond.srv.cs.cmu.edu/afs/cs/project/mach/public/www/doc/abstracts/mapfiles87.html.

[RYT$^+$87] R. Rashid, M. Young, A. Tevanian, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Indepedent Virtual Memory Management for Paged Uniprocessor and Multiprozessor Architectures. In *2. Symposium on Architectual Support for Programming Languages and Operating Systems*, 1987.

[Sta95] W. Stallings. *Operating Systems.* Prentice Hall, 1995.

[Syr00] P. Syrowatka. Verallgemeinerung von mmap() am Beispiel des Memory Mappings von Pipes in Linux - Messwerttabellen, 2000. http://www.complang.tuwien.ac.at/syro/Tabellen.tgz.

[Tan87] A. S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1987.

[Tor94] L. Torvalds. Linux kernel implementation. In Anonymous, editor, *Proceedings of the AUUG94 Conference: Open systems. Looking into the future: 6–9 September 1994, World Congress Centre, Melbourne, Australia*, pages 9–14, Kensington, NSW, Australia, 1994. AUUG Inc.

[V.89] Sasse V. *Intel 80386 Systemprogrammierung.* Markt & Technik, 1989.

[Vah96] Uresh Vahalia. *UNIX Internals.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1996.

[Vle99] T. Van Vleck. Tom Van Vlecks Multics Homepage, 1999. http://www.best.com/ thvv/multics.html.