Parallelisierung des Unix Dateisystems für speichergekoppelte Multiprozessoren

Diplomarbeit im Fach Informatik

vorgelegt
von

Frank Kardel, geb. am 2.1.1963 in Kiel

Angefertigt am
Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:
Prof. Dr. Fridolin Hofmann
Dr. Ing. Claus Uwe Linster
Dipl. Inf. Michael Fäustle

Beginn der Arbeit: 1.5.1988
Abgabe der Arbeit: 14.10.1988

1. Motivation

In den letzten Jahren hat sich die Entwicklung auf dem Computersektor immer mehr gewandelt. Waren früher hauptsächlich Großrechner mit großen Betriebssystemen häufig anzutreffen, so geht der Trend heute zu leistungsfähigen vernetzten Rechnern. Die Halbleiterindustrie hat in den letzten Jahren immer leistungsfähigere Rechner ermöglicht. Doch auch der Bedarf an Rechenleistung ist gestiegen. Um noch mehr Rechenleistung nutzen zu können, hat sich auch die Rechnerarchitektur geändert. Man versucht durch den Einsatz von mehreren Zentraleinheiten noch größere Rechenleistungen anzubieten. Die Architektur hat auch einen Einfluß auf die Struktur des Betriebssystems, das auf dem Multiprozessor laufen soll. In dieser Arbeit werden die Möglichkeiten zur Parallelisierung des Unix^{TM} Dateiverwaltungssystems auf einem symmetrischen, speichergekoppelten Multiprozessorrechner untersucht werden. Unix ist ein Betriebssystem, das von Thompson 1969 bei den Bell Laboratories für eine (PDP7) entwickelt wurde. Durch die leichte Portierbarkeit dieses Betriebsystems und der sehr flexiblen Benutzeroberfläche hat es sich mittlerweile auf den Workstation und Kleincomputermarkt durchgesetzt.

2. Übersicht über das Dateisystem

Bei dem Unix[1] Dateisystem handelt es sich um ein hierarchisches Dateisystem. Es können sich also in einen Inhaltsverzeichnis weitere Inhaltsverzeichnisse befinden, was die Organisation von umfangreichen Datenmengen vereinfacht und damit die Übersicht auch erleichtert.

Das Dateisystem bildet einen Baum, an dessen Spitze das Wurzelinhaltsverzeichnis steht. Dateien und Inhaltsverzeichnisse in dem Unix Dateisystem werden über Pfadnamen angesprochen. Ein Pfadname wird durch das '/' Zeichen in mehrere Komponenten zerlegt, wobei alle Komponenten, bis auf die letzte, Inhaltsverzeichnisse bezeichnen müssen. Die einzelnen Komponenten eines Pfadnamen sind die Namen der Inhaltsverzeichnisse, die nacheinander durchsucht werden müssen, um schließlich zur letzten Pfadnamenkomponente zu gelangen, die die gesuchte Datei darstellt. Beginnt ein Pfadname mit einem '/', so handelt es sich um einen absoluten Pfadnamen und die Auflösung des angegebenen Pfadnamens beginnt mit dem Wurzelinhaltsverzeichnis. Bei Pfadnamen, die nicht mit einem '/' beginnen, handelt es sich um einen relativen Pfadnamen. Hier beginnt die Pfadauflösung mit dem Arbeitsinhaltsverzeichnis. Ein Prozeß kann jederzeit sein Arbeitsinhaltsverzeichnis wechseln. Das Arbeitsinhaltsverzeichnis und relative Pfadnamen ermöglichen einen einfachen Mechanismus um Dateien in einem Teilbaum des Dateibaums bequem anzusprechen, ohne den gesamten Dateibaum betrachten zu müssen.

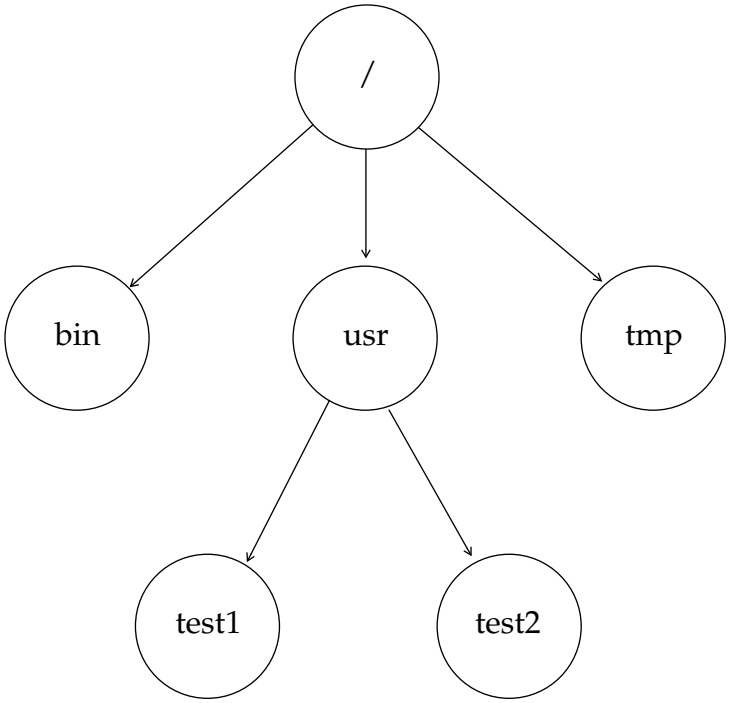


Abb. 1 Unix Dateibaum

[1] UNIX ist ein eingetragenes Warenzeichen der Firma Bell Laboratories, Inc.

Jedes Inhaltsverzeichnis enthält außer den Einträgen für Dateien und weitere Inhaltsverzeichnisse noch zwei besondere Einträge. Es handelt sich hierbei um die Einträge "." und "..". Der "." Eintrag steht für das Inhaltsverzeichnis, in den der "." Eintrag eingetragen ist. Der ".." Eintrag steht für dasjenige Inhaltsverzeichnis, in dem das Inhaltsverzeichnis, das den ".." Eintrag enthält, eingetragen ist. Der ".." Eintrag erlaubt es also, das im Dateibaum eine Stufe höher liegende Inhaltsverzeichnis anzusprechen. Mit Hilfe des ".." Eintrags kann man die relativen Pfadnamen noch sinnvoller nutzen, da man auch auf in dem Dateibaum höhergelegene Inhaltsverzeichnisse zugreifen kann.

Auf die in der Abb. 1 gezeigte Datei "test2" kann durch verschiedene Pfadnamen zugegriffen werden.

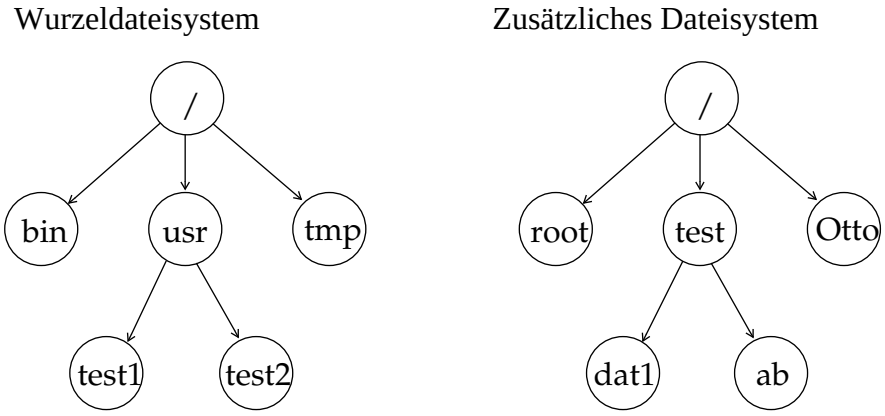
Der absolute Pfad für die Datei "test2" ist "/usr/test2". Setzt man als Arbeitsinhaltsverzeichnis das Inhaltsverzeichnis "/tmp" voraus, so kann für die Datei "/usr/test2" auch der relative Pfad "../usr/test2" angegeben werden.

Dateien im Unix Dateisystem sind nur ein Strom von Bytes. Das Dateisystem prägt den Dateien keine weitere Interpretation auf. Es werden demnach also keine Datensätze oder indexsequentielle Dateien, wie sie von anderen Betriebssystemen her bekannt sind, vom Unix Dateisystem aus angeboten. Die Interpretation der Daten innerhalb einer Unix Datei wird dem jeweiligen Anwenderprogramm überlassen. Aufgrund der sehr einfachen Struktur der Unix Dateien gestaltet sich auch die Schnittstelle zum Betriebssystem entsprechend einfach.

Neben den Dateien und Inhaltsverzeichnissen gibt es noch Spezialdateien. Zu diesen Spezialdateien gehören Gerätedateien, die es erlauben über die Dateisystemschnittstelle Geräte, wie zum Beispiel Platten und Bildschirmsichtgeräte, anzusprechen. Eine weitere Art der Spezialdatei ist der FIFO-Puffer. Diese Spezialdatei wird zur Prozeßkommunikation verwendet.

Um mit mehreren externen Datenträgern umgehen zu können, gibt es im Unix Dateisystem einen Mechanismus um Datenträger in den schon existierenden Dateibaum einbinden zu können. Auf jedem externen Datenträger existieren die Datenstrukturen für ein vollständiges Unix Dateisystem. Dieses Dateisystem kann in den schon bestehenden Dateibaum eingebunden werden. Die Einbindung in den schon bestehenden Dateibaum geschieht dermaßen, daß der Inhalt eines schon bestehenden Inhaltsverzeichnisses durch den Inhalt des Wurzelinhaltsverzeichnisses auf dem externen Datenträger überlagert wird. Das Dateisystem auf dem externen Datenträger stellt einen neuen Teilbaum für den gesamten Dateisystembaum zur Verfügung. Nachdem das zusätzliche Dateisystem eingebunden wurde, kann man transparent

auf das zusätzliche Dateisystem zugreifen. Die Nahtstelle zwischen den verschieden physikalischen Datenträgern ist nicht mehr sichtbar. Das Wurzeldateisystem und die angebundenen Dateisysteme bilden einen gemeinsamen Dateibaum..



Dateibaum nach Anbinden des zusätzlichen Dateisystems an dem "/usr" Inhaltsverzeichnis

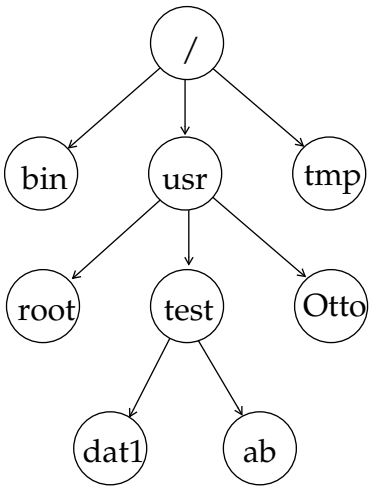


Abb. 2 Unix Dateibaum mit Dateisystemanbindung

Das System V.3 Dateisystem gliedert sich in vier logische Schichten. In der obersten Schicht werden die Unix Systemaufrufe, die sich auf das Dateisystem beziehen, realisiert. Unterhalb dieser Schicht liegt das eigentliche Dateisystem, was sich in zwei logische Schichten gliedert. Diese Zweiteilung ermöglicht es auch fremde Dateisysteme, wie zum Beispiel Dateisysteme von anderen Betriebssystemen auch in das UNIX Dateisystem einbinden zu können. Um dies zu erreichen, wurden Dateisystemtypen eingeführt. Mit Hilfe dieser Dateisystemtypen ist es möglich, zwischen den verschie-

denen Dateisystemen, wie sie auf dem externen Datenträger hinterlegt sind, zu unterscheiden. Das eigentliche Dateisystem besteht aus einem dateisystemtypabhängigen Teil und einem dateisystemtypunabhängigen Teil. Während es nur einen dateisystemtypunabhängigen Teil gibt, existieren für jeden möglichen Dateisystemtyp die abhängigen Teile, die sich mit dem exaktem Aufbau der Daten auf dem externen Datenträger befassen.

Die vierte Schicht im Unix Dateisystem ist die Blockpufferverwaltung. Sie stellt die Schnittstelle zwischen Dateisystem und externen Geräten dar. Die Aufgabe der Blockpufferverwaltung ist es, die Anzahl der Datentransporte zwischen Hauptspeicher und externen Geräten zu minimieren, um somit den Datendurchsatz zu erhöhen.

2. 1. Datenstrukturen des Unix Dateisystems

Das System V.3 Dateisystem verwendet drei wesentliche Datenstrukturen. Die erste Datenstruktur ist der "Superblock". Der "Superblock" enthält grundsätzliche Daten über das Dateisystem. Hierzu gehören:

```
struct  filsys
{
        ushort  s_isize;        /* size in blocks of i-list */
        daddr_t s_fsize;        /* size in blocks of entire volume */
        short   s_nfree;        /* number of addresses in s_free */
        daddr_t s_free[NICFREE];/* free block list */
        short   s_ninode;       /* number of i-nodes in s_inode */
        ino_t   s_inode[NICINOD];/* free i-node list */
        char    s_flock;        /* lock during free list manipulation
                                 */
        char    s_ilock;        /* lock during i-list manipulation */
        char    s_fmod;         /* super block modified flag */
        char    s_ronly;        /* mounted read-only flag */
        time_t  s_time;         /* last super block update */
        short   s_dinfo[4];     /* device information */
        daddr_t s_tfree;        /* total free blocks*/
        ino_t   s_tinode;       /* total free inodes */
        char    s_fname[6];     /* file system name */
        char    s_fpack[6];     /* file system pack name */
       long    s_fill[13];    /* ADJUST to make sizeof filsys be 512 */
        long    s_magic;        /* magic number to indicate new file
system */
        long    s_type;         /* type of new file system */
};
```

Damit das Unix Dateisystem mit der externen Repräsentation der Daten umgehen kann, wird, solange ein Dateisystem eingebunden ist, eine Kopie des "Superblocks" im Hauptspeicher gehalten.

Die nächste wichtige Datenstruktur im Zusammenhang mit dem Unix Dateisystem ist die "Inode". Die Inode enthält alle Daten, die eine Datei im Unix Dateisystem zu beschreiben. Eine Inode enthält folgende Informationen:

```
typedef struct  inode
```

```
    {
            struct  inode   *i_forw;     /* inode hash chain */
            struct  inode   *i_back;     /* '' */
            struct  inode   *av_forw;    /* freelist chain */
            struct  inode   *av_back;    /* '' */
            int     *i_fsptr;     /* "typeless" pointer to fs dependent */
            long    i_number;     /* i number, 1-to-1 with dev address */
            char    i_epoch;      /* epoch of inode */
            ushort  i_ftype;      /* file type = IFDIR, IFREG, etc. */
            short   i_fstyp;      /* File system type */
            off_t   i_size;       /* size of file */
            ushort  i_uid;        /* owner */
            ushort  i_gid;        /* group of owner */
            ushort  i_flag;
            ushort  i_fill;
            cnt_t   i_count;      /* reference count */
            short   i_nlink;      /* directory entries */
            dev_t   i_rdev;       /* Raw device number */
            dev_t   i_dev;        /* device where inode resides */

            struct  mount   *i_mntdev; /* ptr to mount dev inode resides
                                       on */
            union i_u {
                    struct  mount   *i_mton;/* pntr to mount table entry*/
                                            /* that this inode is
                                               "mounted on" */
                    struct stdata   *i_sp;  /* Associated stream.*/
            } i_un;
    #ifdef FSPTR
            struct fstypsw *i_fstypp;       /* pointer to file system */
                                            /* switch structure */
    #endif
            long    *i_filocks;             /* pointer to filock list */
            struct  rcvd    *i_rcvd;        /* receive descriptor */
            unsigned long   i_vcode;        /* inode version code */
            ushort  i_wcnt;                 /* open for write count (RFS
caching) */} inode_t;
```

Diese Daten beschreiben ein Datei im Unix Dateisystem mit den Rechten, dem Dateityp, und den zu dieser Datei gehörenden Blöcken.

Eine Datei wird durch ihre Inode beschrieben. Die Unterscheidung zwischen den Dateitypen geschieht durch den Dateityp, der in der Inode abgelegt ist.

Normale Dateien bestehen aus einer Inode und Datenblöcken. Inhaltsverzeichnisse werden wie normale Dateien gespeichert, können aber nicht von der Benutzerschnittstelle aus beschrieben werden.

Spezialdateien bestehen nur aus einer Inode, in der der jeweilige Gerätetreiber und ein Parameter für den Gerätetreiber vermerkt sind.

Als letzte Datenstruktur sind die Datenblöcke zu nennen. Hier werden die Daten, die in die Datei geschrieben werden, abgelegt.

Die Struktur der System V.3 Inhaltsverzeichnisse ist ganz einfach gehalten. Ein Inhaltsverzeichnis besteht aus einer Folge von 16 Byte großen Einträgen. Die ersten beiden Bytes stellen die Inodenummer dar, während die letzten 14 Bytes für einen Dateinamen mit bis zu 14 Zeichen Länge verwendet werden. Die Inhaltsverzeichnisse werden nur durch die *namei()* Routine des Dateisystems manipuliert. Weil in den Inhaltsverzeichnissen keine Information über die eigentliche Datei vermerkt ist, sondern nur die Nummer der Inode, die diese Datei beschreibt, ist es möglich, Verweise auf Dateien zur Verfügung zu stellen. Ein Verweis wird dadurch erzeugt, daß man in einen Inhaltsverzeichnis ein weitere Inodenummer-Dateinamen Paar einträgt, was auf eine schon existierende Inode verweist. So können Dateien im Unix Dateisystem mehrere verschiedene Namen haben.

Abb. 3 Physikalische Organisation des System V.3 Dateisystems

2. 2. Die Benutzerschnittstelle

Im Folgenden soll auf die Benutzerschnittstelle des Dateisystems und die Blockpufferverwaltung eingegangen werden. Es werden nicht alle Aspekte des Dateisystems exakt beschrieben werden, sondern nur die Teile, die im Zusammenhang mit der Parallelisierung des Dateisystems interessant sind. Das Betriebssystem Unix stellt dem Benutzer an der Systemaufrufschnittstelle 17 Operationen zur Verfügung, die direkten Einfluß auf das Dateisystem haben. Weiterhin gibt es noch einige Systemaufrufe, die Einfluß auf die Benutzersicht des Dateisystems haben.

Um die Anforderungen an die Parallelisierung des Dateisystems zu verstehen, soll hier eine kurze Übersicht über die Systemaufrufe, die das Dateisystem direkt beeinflussen, gegeben werden. Die Systemaufrufe lassen sich in drei Gruppen aufteilen.

Die erste Gruppe umfaßt die Systemaufrufe, die den Zugriff auf Dateien realisieren. Hierzu gehören:

- *open(Pfad, Anzeigen, Zugriffsrechte)*
  Der *open* Systemaufruf erhält als Parameter einen Dateipfad, der die gewünschte Datei angibt. Weiterhin geben die Anzeigen darüber Auskunft, in welcher Form die Datei geöffnet werden soll. Es kann angegeben werden, ob die Datei gelesen oder geschrieben werden soll, oder ob sie angelegt werden soll, falls sie noch nicht existiert. Es gibt noch weitere Anzeigen, deren Bedeutung im Handbuch nachzulesen ist.
  Das Resultat des erfolgreichen *open* Aufrufs ist ein Dateideskriptor, der für die *read, write, lseek* und *close* Systemaufrufe benötigt wird.

- *creat(Pfad, Zugriffsrechte)*
  Der *creat* Aufruf ist im wesentlichen ein *open* Aufruf, bei dem die Datei mit der CREATE Anzeige geöffnet wird. Diese Schnittstelle besteht noch aus Gründen der Kompatibilität zu älteren Unix Systemen.

- *close(Dateideskriptor)*
  Mit dem *close* Aufruf wird eine Datei wieder geschlossen und damit der angegebene Dateideskriptor wieder ungültig gemacht.

- *read(Dateideskriptor, Adresse, Länge)*
  Durch den *read* Aufruf können von einer Datei Daten in den Adreßraum des Prozesses gelesen werden. Die Daten werden von der angegebenen *Adresse* an in dem Adreßraum des Prozesses abgelegt. Es werden maximal soviele Bytes transferiert, wie der *Längen*-Parameter angibt. Der *read*-Aufruf gibt die Anzahl der tatsächlich transferierten Bytes als Resultat zurück. Für Dateien, die sich auf einem Externspeicher befinden, ist dieser Aufruf atomar, d.h. daß die Datei in einem Stück gelesen wird, ohne daß während des Lesens ein anderer Prozeß gleichzeitig in die Datei schreiben kann.

- *write(Dateideskriptor, Adresse, Länge)*
  Der Systemaufruf *write* erlaubt es Daten aus dem Adreßraum des Prozesses in eine Datei zu schreiben. Der *write*-Aufruf ist ebenso wie der *read*-Aufruf atomar. Ebenso werden die Parameter gleich verwendet, wobei allerdings der Transfer vom Adreßraum des Prozesses auf die Datei geschieht. Das Resultat des *write*-Systemaufrufes ist die Anzahl der tatsächlich transferierten Bytes.
  Das Unix Dateisystem kennt kein Transaktionskonzept, wie es von Datenbanksystemen her bekannt ist. Sollten mehrere Prozesse gleichzeitig eine Datei schreiben und lesen wollen, so müssen sich diese Prozesse selbst synchronisieren. Es wird nur die Unteilbarkeit einer einzelnen *read* oder *write* Operation garantiert.

- *lseek(Dateideskriptor, Startpunkt, Versatz)*
  Das Positionieren innerhalb einer Datei wird durch den *lseek*-Systemaufruf ermöglicht. Der *lseek* Systemaufruf verändert die mit dem Dateideskriptor assoziierte Schreib- / Lesemarke. Der nächste *read*- oder *write*-Aufruf wird dann ab dieser Position lesen beziehungsweise schreiben. Der Parameter *Position* gibt an, ob relativ zum Anfang, zur aktuellen Position oder zum Ende der Datei positioniert werden soll. Der *Versatz* gibt den Wert an, um den die Schreib-/Lesemarke verschoben werden soll.

Die nächste Gruppe von Systemaufrufen hat primär die Modifikation des Dateibaums zum Ziel. Mit diesen Systemaufrufen kann die Struktur des Dateibaums geändert werden.

- *mkdir(Pfad, Zugriffsrechte)*
  Ein neues Inhaltsverzeichnis wird mit dem *mkdir* Systemaufruf angelegt.

- *rmdir(Pfad)*
  Der *rmdir*-Aufruf entfernt ein Inhaltsverzeichnis, sofern es keine anderen Einträge als "." und ".." enthält.

- *link(alter Pfad, neuer Pfad)*
  Ein neuer Verweis auf eine Datei wird mit dem *link*-Aufruf hergestellt. Durch das Eintragen eines neuen Verweises wird der Referenzzähler der betroffenen Datei um eins erhöht.

- *unlink(Pfad)*
  Der Systemaufruf *unlink* entfernt einen Verweis auf eine Datei aus einem Inhaltsverzeichnis, dabei wird der Verweiszähler dieser Datei um eins erniedrigt. Eine Datei wird erst dann endgültig gelöscht, wenn der Verweiszähler auf 0 sinkt und kein Prozeß mehr einen gültigen Dateideskriptor für diese Datei besitzt.

- *mknod(Pfad, Typ und Zugriffsrechte, Gerätekennung)*
  Durch den *mknod*-Systemaufruf werden Spezial- und Gerätedateien angelegt. Mit diesem Mechanismus werden Gerätetreiber, wie zum Beispiel Terminaltreiber, in das Unix Dateisystem eingebunden. Dies erlaubt ein einheitliches Ansprechen von Dateien und Geräten.

- *mount(Spezial Datei, Pfad eines Inhaltsverzeichnisses, Schreibkennung)*
  Durch den *mount*-Systemaufruf werden weitere Dateisysteme in den schon bestehenden Dateibaum eingebunden. Hierbei wird das Inhaltsverzeichnis, das beim *mount*-Aufruf angegeben wurde, durch das Dateisystem, das sich hinter der Spezialdatei verbirgt, überlagert. Über den angegebenen Pfad erreicht man jetzt die Dateien und Inhaltsverzeichnisse des neu eingebundenen Dateisystems, während alle früher zugänglichen Dateien nicht mehr zugänglich sind. Dieser Überlagerungsmechanismus erlaubt es weitere Dateisysteme in den Dateibaum transparent einzubinden.

- *umount(Spezialdatei)*
  Mit dem *umount*-Systemaufruf wird ein eingebundenes Dateisystem wieder aus dem Dateibaum entfernt. Hierdurch sind die Dateien des eingebundenen Dateisystems nicht mehr zugreifbar, aber der ursprünglich überlagerte Dateisystemteilbaum wird wieder verfügbar.

In der dritten Gruppe findet man die Systemaufrufe, die die Zugriffsrechte von Dateien und Inhaltsverzeichnissen überprüfen oder modifizieren.

- *access(Pfad, Zugriffsart)*
  Der *access*-Systemaufruf erlaubt es zu prüfen, ob eine Datei oder ein Inhaltsverzeichnis geschrieben, gelesen, ausgeführt oder durchsucht werden darf.

- *chmod(Pfad, neue Zugriffsrechte)*
  Der *chmod*-Systemaufruf setzt neue Zugriffsrechte für die mit Pfad angegebene Datei.

- *chown(Pfad, Eigner, Gruppe)*
  Durch den *chown*-Systemaufruf wird der Eigner und die Gruppenzugehörigkeit der Datei verändert.

Bei allen Systemaufrufen wird die Zulässigkeit der Parameter überprüft und im Fehlerfalle eine Fehlerkennung zurückgeliefert.

2. 3. Die *namei()* Routine

Die *namei()* Routine stellt einen zentralen Teil des Unix Dateisystems dar. Die vom Benutzer angegeben Dateipfade werden durch die *namei()* Routine in Inodestrukturreferenzen übersetzt. Die Inodestruktur beschreibt die eigentliche Datei und wird für alle weiteren Dateisystemfunktionen benötigt. Die *namei()* Routine stellt nicht nur die Pfadübersetzung zur Verfügung, sondern auch 7 weitere Operationen auf dem Dateisystem. Diese Operationen sind:

- LOOKUP
  Übersetzung eines Pfades in eine *Inode*-Struktureferenz.
- LINK
  Einen Dateiverweis in einen Inhaltsverzeichnis erzeugen
- MKNOD
  Eine Spezialdatei anlegen
- XCREAT
  Eine Standartdatei, die noch nicht existiert, erzeugen
- CREAT
  Eine Standartdatei öffnen, falls sie existiert, und erzeugen, wenn sie nicht existiert
- MKDIR
  Ein Inhaltsverzeichnis erzeugen
- RMDIR
  Ein Inhaltsverzeichnis entfernen
- DEL
  Einen Dateiverweis aus einem Inhaltsverzeichnis löschen

Diese Liste zeigt, daß die *namei()* Routine nur für die Operationen LOOKUP, XCREAT und CREAT eine Inodestrukturreferenz zurückliefert. Alle anderen Operation liefern nur eine Fehlerkennung zurück. Weil bei allen Operationen der *namei()* Routine der Pfad übersetzt werden muß, sind alle Operationen in der *namei()* Routine zusammengefaßt worden.

Wie schon oben erwähnt wurde, ist das eigentlich Dateisystem in zwei Schichten zergliedert. Diese Aufteilung gibt es auch bei der *namei()* Routine.

2. 3. 1. Dateisystemtypunabhängiger Teil der *namei()* Routine

Die Übersetzung des gesamten Pfades wird durch den dateisystemtypunabhängigen Teil der *namei()* Routine gesteuert. Beim Aufruf der *namei()* Routine wird die gewünschte Operation zusätzlich zum Pfad angegeben.

Zu Beginn der *namei()* Routine wird der Pfad zuerst daraufhin untersucht, ob er mit einem '/'-Zeichen beginnt. Ist dies gegeben, so beginnt die Pfadübersetzung mit dem Wurzelinhaltsverzeichnis, ansonsten wird das Arbeitsinhaltsverzeichnis verwendet.

Um den Pfad zu übersetzen, wird vom Pfad eine Pfadkomponente abgetrennt. Mit dieser Pfadkomponente wird dann der dateisystemtypabhängige Teil der *namei()* Routine aufgerufen, der dann eine von acht möglichen Operationen ausführt, sofern es sich um die letzte Pfadkomponente handelt, ansonsten wird die Operation *LOOKUP (*Aufsuchen) ausgeführt. Die dateisystemtypabhängige *namei()* Routine liefert zwei Resultate. Zum einen eine Inodestrukturreferenz, die aus der Übersetzung der Pfadnamenkomponente herrührt, und zum anderen einen Rückgabewert, der angibt, wie die Operation verlaufen ist.

Solange noch Pfadkomponenten vorhanden sind und die Operation LOOKUP erfolgreich war, wird die zurückgegebene Inodestrukturreferenz daraufhin untersucht, ob es sich um ein Inhaltsverzeichnis handelt. Ist dies gegeben, so wird die nächste Pfadkomponente in dem diesem Inhaltsverzeichnis übersetzt. Alle anderen Operationen werden nur ausgeführt, wenn es sich um die letzte Pfadkomponente gehandelt hat.

2. 3. 2. Dateisystemtypabhängiger Teil der *namei()* Routine

Der dateisystemtypabhängige Teil der *namei()* Routine führt die geforderte Operation auf einer einzelnen Pfadkomponente in dem gerade durchsuchten Inhaltsverzeichnis aus.

Für alle Operationen wird das Inhaltsverzeichnis nach der angegebenen Pfadkomponente durchsucht und, falls die Pfadkomponente gefunden wird, deren Position vermerkt. Während des Durchsuchens wird auch eine freie Position im Inhaltsverzeichnis notiert, um neue Einträge anlegen zu können.

Nachdem das gesamte Inhaltsverzeichnis durchsucht wurde, wird die geforderte Operation ausgeführt.

- LOOKUP
  Bei der LOOKUP Operation wird, falls die Pfadkomponente gefunden wurde, die entsprechende Inodenummer zurückgegeben.
- LINK
  Die LINK Operation erzeugt einen neuen Eintrag mit dem in der Pfadkomponente angegebenen Namen der Inodenummer aus der beim Aufruf angegebenen Inodestrukturreferenz. Es wird überprüft, ob die Datei, die einen weiteren Verweis bekommen soll, sich auch auf dem gleichen externen Datenträger befindet. Dies ist notwendig, weil in den Inhaltsverzeichnissen nur die Nummer der Inode verzeichnet wird und die Nummer nur innerhalb eines externen Dateisytems eindeutig ist. Außerdem wird der Verweiszähler der Datei um eins erhöht.
- MKNOD
  Mit der MKNOD Operation wird eine neue Inode und ein Eintrag im Inhaltsverzeichnis erzeugt. In der Inode werden die Daten für eine Spezialdatei eingetragen.
- XCREAT
  Es wird eine leere Datei (Inode und Eintrag im Inhaltsverzeichnis) erzeugt, wobei sichergestellt wird, daß diese Datei noch nicht existiert.
- CREAT
  Bei der CREAT Operation wird, falls es die Datei schon gibt, die Datei auf die Länge 0 verkürzt und eine Inodestrukturreferenz auf die schon bestehende Datei zurückgegeben. Existiert die Datei noch nicht, so wird eine neue Inode und ein Eintrag im Inhaltsverzeichnis erzeugt und die Strukturreferenz auf diese Inode zurückgegeben.
- MKDIR
  Die MKDIR Operation erzeugt eine neue Inode vom Typ Inhaltsverzeichnis und legt in der Inhaltsverzeichnisdatei die "." und ".." Einträge an. Hierbei muß auf der Verweiszähler für das übergeordnete Inhaltsverzeichnis um eins erhöht werden, da der ".." Eintrag einen Verweis auf das übergeordnete Inhaltsverzeichnis darstellt. Das neue Inhaltsverzeichnis hat einen Verweiszählerwert von zwei, da es zum einen in dem übergeordneten Inhaltsverzeichnis eingetragen ist und zum anderen der "." Eintrag auch ein Verweis auf dieses Inhaltsverzeichnis ist.

- RMDIR
  Um die RMDIR Operation durchzuführen, wird zuerst überprüft, ob in dem Inhaltsverzeichnis noch Einträge außer "." und ".." vorhanden sind. Ist dies der Fall, so kann das Inhaltsverzeichnis nicht gelöscht werden. Anderenfalls wird der Eintrag im übergeordneten Inhaltsverzeichnis gelöscht. Ist der "." Eintrag in dem zu löschenden Inhaltsverzeichnis vorhanden, so wird der Verweiszähler für das Inhaltsverzeichnis um zwei erniedrigt, ansonsten nur um eins. Dies hat gewöhnlich ein Löschen der Inhaltsverzeichnisdatei zur Folge. Wenn der ".." Eintrag vorhanden ist, der Verweiszähler für das übergeordnete Inhaltsverzeichnis um 1 erniedrigt
- DEL
  Bei der DEL Operation wird der Eintrag im Inhaltsverzeichnis gelöscht und der Verweiszähler der Datei um eins erniedrigt. Erreicht dieser Zähler den Stand null, so werden alle Blöcke dieser Datei freigegeben und die Inode als unbenutzt markiert.

Die Operationen werden deswegen in der dateisystemtypabhängigen Routine realisiert, weil hier für die Einträge in die Inhaltsverzeichnisse und die Verwaltung der freien Datenblöcke eine genaue Kenntnis der Datenstrukturen auf dem externen Datenträger nötig ist.

2. 4. Die Blockpufferverwaltung

Mit der Blockpufferverwaltung versucht man den Datentransport zwischen Hauptspeicher und externem Datenträger zu minimieren. Aus diesem Grund wurde bei der Blockpufferverwaltung die *Least-Recently-Used* Strategie verwendet, in der Hoffnung, daß ein gerade benötigter Block sehr wahrscheinlich in der Zukunft wieder benötigt wird. Man hat zeigen können, daß bei ausreichend großen Puffergrößen (4 - 8Mb) eine sehr gute Trefferrate (80-90%) [Ousterhout et. al85] für diesen Puffer erzielt wird.

Bei der Blockpufferverwaltung handelt es sich um eine stark verzeigerte Datenstruktur (Abb. 4).

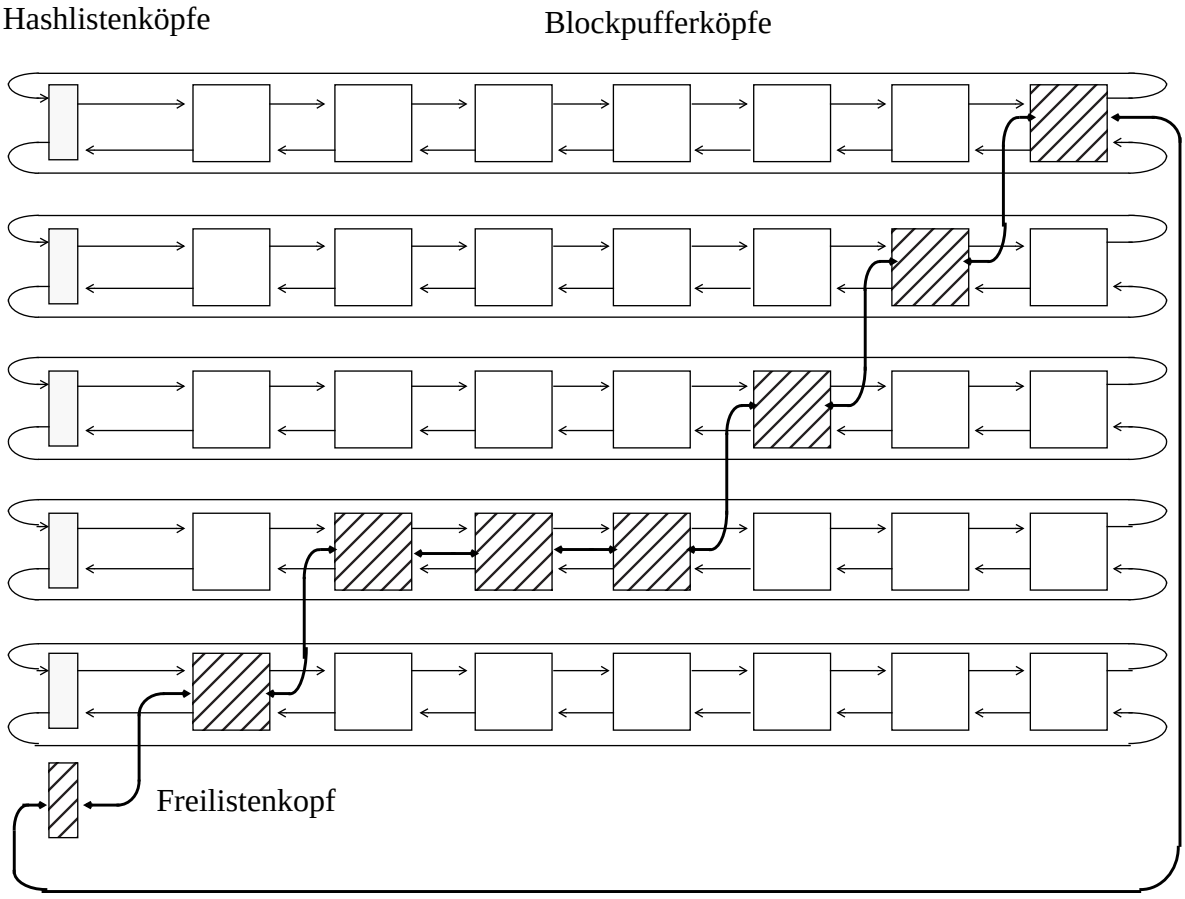


Abb. 4 Blockpufferdatenstrukturstruktur

Der Blockpuffer verwendet vier Datenstrukturen:

- *Blockpufferkopf*

  Der Blockpufferkopf wird zur Verwaltung der einzelnen Blöcke, deren Datenbereiche über einen Zeiger im Blockpufferkopf referenziert werden, verwendet. Inbesondere wird hier die pro-Block-Synchronisation abgewickelt. Weiterhin enthält der Blockpuffer die Gerätekennung und die Blocknummer auf dem Gerät, um die Blöcke zu identifizieren. Außerdem gibt es ein Feld, das die Fehlerkennungen für Ein- und Ausgabeoperationen aufnimmt. Um gewährleisten zu können, daß die Daten innerhalb einer gewissen Zeitspanne auf den externen Datenträger geschrieben wVerden, wird vermerkt, wann der letzte Ausgabeauftrag gegeben wurde. Die Blockpufferköpfe haben noch Verkettungseinträge für zwei doppelt verkettete Listen, die das Auffinden der Blockpufferköpfe ermöglichen.

- *die Blöcke selbst*

  Die Blöcke enthalten die eigentlichen Daten der Blockpufferverwaltung und werden über einen Zeiger im Blockpufferkopf referenziert.

– *die Freiliste*

  Die Freiliste ist eine doppelt verkettete Liste, die alle zur Zeit nicht belegten Blöcke verwendet. In diesen Blöcken können noch gültige Daten stehen und sogar Blöcke, die erst noch auf den Datenträger geschrieben werden müssen. Blöcke werden der Freiliste überall entnommen, aber nur an am Anfang oder am Ende hinzugefügt. Die Freiliste realisiert die *Least-Recently-Used* Strategie der Pufferverwaltung. Blöcke, die freigegeben werden, aber noch gültige Daten enthalten, werden am Ende der Freiliste eingehängt. Blöcke, die keine gültigen Daten enthalten werden am Anfang der Liste eingehängt. Wird ein neuer leerer Datenblock benötigt, so wird er von Anfang der Freiliste entfernt. Hierbei kann es vorkommen, das der Datenblock am Anfang der Freiliste noch Daten enthält, die noch auf den externen Datenträger geschrieben werden müssen. In diesem Fall wird ein Schreibauftrag an den Gerätetreiber abgegeben und der nächste Datenblock von der Freiliste geholt. Da die Datenblöcke mit den noch gültigen Daten immer am Ende der Freiliste eingefügt werden, stehen sie noch so lange zur Verfügung, bis sie zum Anfang der Freiliste gewandert sind, um dann wiederverwendet zu werden. Wird ein Datenblock benötigt, der sich gerade in der Freiliste befindet, so wird er einfach aus der Freiliste ausgekettet und nach Verwendung wieder ans Ende der Freiliste gehängt. Am Ende der Freiliste steht immer der zuletzt freigegebene Block, der noch gültige Daten enthält.

– *die Hashlisten*

  Um möglichst schnell einen durch Gerätenummer und Blocknummer beschriebenen Block im Blockpuffer zu finden, gibt es noch die Hashlisten. Aus der Blocknummer und der Gerätenummer wird ein Index in ein Feld von doppelt verketteten Listen berechnet. Wenn der geforderte Block sich in der Blockpufferverwaltung befindet und gültige Daten enthält, so ist er in der entsprechenden Hashliste eingekettet. Wenn der Block gerade nicht benötigt wird und gültige Daten enthält, so ist er einerseits in der entsprechenden Hashliste vermerkt und ebenso in der Freiliste. Durch die Hashlisten können also die Datenblöcke gefunden werden, die dann aus der Freiliste entnommen werden, um nach Gebrauch wieder ans Ende der Freiliste gehängt zu werden.

2. 5. Die Inodepufferverwaltung

Die Struktur der Inodepufferverwaltung entspricht der Blockpufferverwaltung. Auch hier wird die *Least-Recently-Used* Strategie für freigegebene Inodestrukturen verwendet, um oft verwendete Inodes schnell zur Verfügung zu haben. Der einzige wesentliche Unterschied besteht in der verwendeten Hashfunktion, die bei der Inodepufferverwaltung nur von der Inodenummer abhängt.

3. Multiprozessorsysteme

Es sind inzwischen viele verschiedene Multiprozessorarchitekturen bekannt. Es gibt eng und lose gekoppelte Multiprozessoren. Unter lose gekoppelten Multiprozessoren versteht man Zentraleinheiten, die unter Verwendung von Protokollen für lose gekoppelte Netze (z. B. IP - Internet Protocol) kommunizieren. Bei eng gekoppelten Multiprozessoren handelt es sich um speichergekoppelte Systeme, bei denen alle Zentraleinheiten auf einen gemeinsamen Hauptspeicher zugreifen können. Hierdurch wird eine sehr leistungsfähige Kommunikation ermöglicht. Rechner dieser Art lassen sich sehr gut für Realzeitanwendungen und Mehrbenutzersysteme anwenden. Während es schon viele speichergekoppelte Multiprozessoren gibt, muß noch viel Arbeit auf dem Gebiet der Betriebssysteme für Multiprozessoren geleistet werden. Unter den schon bestehenden speichergekoppelten Multiprozessoranordnungen gibt es zwei Hauptanordnungen. Einmal die symmetrischen Multiprozessoren und zum anderen die asymmetrischen. Unter symmetrischen Multiprozessoren versteht man Zentraleinheiten, die alle gleichberechtigt sind. Bei asymmetrischen Multiprozessoren werden die einzelnen Prozessoren nur für bestimmte Teilaufgaben verwendet. Diese Spezialisierung kann zum einen durch unterschiedliche Hardwareeigenschaften der Prozessoren hervorgerufen werden (z. B. Ein-/Ausgabegeräte sind nur an einem bestimmten Prozessor angeschlossen), oder zum anderen durch Strukturierungsüberlegungen im Betriebssystem.

3. 1. Prozessorvergabestrategien

Wichtig für das Verständnis der Vorgänge in einem Rechner ist der Begriff des Prozesses. Es gibt viele Definitionen für den Prozeß. Ein sehr gebräuchliche ist die der Ausführung eines Programms. Andere Definitionen bezeichnen einen Prozeß als asynchrone Aktivität oder als Einheit, der Prozessoren zugewiesen werden können.

In einem Mehrprozeßsystem gibt es verschiedene Vorgehensweisen, wie der Prozessor den einzelnen Prozessen zugeordnet wird. Ein wichtiges Kriterium ist, ob ein Prozeß den Prozessor solange behält, bis er den Prozessor freiwillig aufgibt, weil er sich (z. B.) an einem Betriebsmittel blockiert. Kann der Prozeß den Prozessor solange behalten, bis er sich blockiert, so wird eine nicht präemptive Prozessorvergabestrategie verwendet. Kann der Prozessor dem Prozeß jederzeit entzogen werden, so spricht man von einer präemptiven Prozessorvergabestrategie.

Die verwendete Prozessorvergabestrategie hat einen entscheidenden Einfluß auf die Implementation von Synchronisationsmechanismen. Bei einer nicht präemptiven Vergabestrategie kann es nicht passieren, daß Datenstrukturen unerwartet aufgrund von Prozeßumschaltungen verändert werden, weil alle Situationen, bei denen eine

Prozeßumschaltung stattfinden kann, bekannt sind. Man muß Zugriffe auf gemeinsame Betriebsmittel nur dann synchronisieren, wenn während des kritischen Abschnitts eine Prozeßumschaltung stattfinden kann.

Bei präemptiven Prozessorvergabestrategien besteht jederzeit die Gefahr, daß eine Prozeßumschaltung stattfindet. Hier existieren keine atomar ausführbaren Programmabschnitte uns somit muß jeder kritische Abschnitt, abgesichert werden.

Präemptive Prozessorvergabestrategien werden bevorzugt in Realzeit-Betriebssystemen verwendet, weil hier Antwortszeiten garantiert werden müssen.

Um dies zu erreichen, werden den einzelnen Prozessen Prioritäten zugeteilt, die der Wichtigkeit der Prozesse entsprechen. Der Prozessor wird immer demjenigen laufbereiten Prozeß zugeteilt, der die höchste Priorität hat. Sobald ein höherpriorer Prozeß laufbereit wird, wird der Prozessor dem aktuell laufenden Prozeß entzogen und dem höherprioren Prozeß zugeteilt.

Eine präemptive Prozessorvergabe wird auch bei Timesharing-Prozessorvergabestrategien verwendet. Hier wird der Prozessor dem aktuell laufenden Prozeß dann entzogen, wenn seine Zeitscheibe abgelaufen ist.

Die Problematik einer präemptiven Prozessorvergabestrategie liegt darin, daß ein Prozeß jederzeit den Prozessor zugunsten anderer Prozesse entzogen werden kann. Greifen mehrere Prozesse auf gemeinsame Datenstrukturen zu, so kann jeder Prozeß gerade während der Manipulation der gemeinsamen Datenstrukturen unterbrochen werden. Die anderen Prozesse können dann die Datenstrukturen in einem inkonsistenten Zustand vorfinden. Um dies zu vermeiden müssen die beteiligten Prozesse so synchronisiert werden, daß jeder Prozeß die gemeinsamen Datenstrukturen immer in einem konsistenten Zustand vorfindet.

3. 2. Synchronisationsmechanismen

Ein großes Problem bei Mehrprozeßsystemen ist die Synchronisation der verschiedenen Prozesse beim Zugriff auf gemeinsame Datenstrukturen. Um eine korrekte Abwicklung der Algorithmen zu gewährleisten, muß jeder Prozeß ein konsistentes Bild der gerade verwendeten Datenstruktur sehen. In der Literatur sind viele Synchronisationsmechanismen bekannt. Hierzu zählen:

- Interruptsperren
- aktives Warten
- Semaphore Operationen (P() und V()) [Dijkstra68]
- Bedingte kritische Abschnitte [Brinch72,73, Hoare72]
- Monitore [Dijkstra72,Brinch72,Hoare72]
- Pfadausdrücke [Campbell74,Habermann74]
- Verträglichkeitsrelationen [Keramidis82,Mackert83]

Obwohl es hardwareunabhängige Mechanismen zur Implementation von Synchronisationsmechanismen gibt [Lamport86a,b] werden jedoch hardwareunterstützte Primitiven aufgrund ihrer höheren Effizienz bevorzugt. Die höheren programmiersprachlichen Synchronisationsmechanismen werden dann auch mit Hilfe dieser Hardwareprimitiven realisiert.

Der Vorteil der höheren Synchronisationsmechanismen, wie z. B. Monitore und Verträglichkeitsrelationen, ist für neue Programmentwicklungen von unbestreitbaren Vorteil. Durch diese Hilfsmittel wird das Programmsystem erheblich leichter handhabbar.

3. 2. 1. Softwaresynchronisationsmechanismen

Der holländische Mathematiker Dekker hat als erster eine reine Softwarelösung für die Prozeßsynchronisation gefunden. Einzige Voraussetzung ist, daß simultane Speicherzugriffe von der Hardware serialisiert werden und eine einzelne Machineninstruktion nicht unterbrochen werden kann. Hier soll der Dekkeralgorithmus skizziert werden, dessen Entwicklung Dijkstra [Dijkstra65] beschrieben hat.

Der Dekkeralgorithmus erreicht die Synchronisation zweier Prozesse durch Verwendung dreier Variablen. Die erste boolsche Variable (p1wantstoenter) gibt an, ob der erste Prozeß seinen kritischen Abschnitt betreten möchte, die zweite boolsche Variable (p2wantstoenter) erfüllt diese Aufgabe für den zweiten Prozeß. Die dritte Variable (favoredprocess) gibt an, welcher der beiden Prozesse favorisiert werden soll, um ein Aushungern eines Prozesses zu vermeiden.

Die Grundidee des Dekkeralgorithmus liegt darin, daß der Prozeß, wenn er seinen kritischen Abschnitt betreten möchte, dies dadurch bekanntgibt, daß er seine boolsche Variable (p1/2wantstoenter) auf *True* setzt. Danach wird überprüft, ob der andere Prozeß auch seinen kritischen Abschnitt betreten möchte. Ist dies nicht der

Fall, so ist der kritische Abschnitt erfolgreich betreten worden. Andernfalls muß der Konflikt aufgelöst werden. Dies erfolgt durch die Variable, die angibt, welcher Prozeß gerade favorisiert werden soll.

Soll der gerade betrachtete Prozeß favorisiert werden, so wird in einer Schleife darauf gewartet, daß der nichtfavorisierte Prozeß seine Kennung, daß er den kritischen Abschnitt betreten möchte, zurücksetzt, dann kann der kritische Anschnitt betreten werden.

Ist der Prozeß nicht favorisiert, so setzt er seine Kennung zurück, und wartet darauf, daß der favorisierte Prozeß den kritischen Abschnitt verläßt, was durch das Zurücksetzen der boolschen Variablen geschieht. Nachdem der andere Prozeß seinen kritischen Abschnitt wieder verlassen hat, wird wieder versucht den kritischen Abschnitt zu betreten. Dies gelingt auch, da jetzt dieser Prozeß favorisiert wird. Das Verlassen des kritischen Abschnitts erfordert zwei Operationen. Zu einen wird die Variable, die den favorisierten Prozeß kennzeichnet auf den jeweils anderen Prozeß gesetzt, was ein Aushungern im Konfliktfall verhindert, und zum anderen wird die eigene Variable, die angibt, das der kritische Abschnitt betreten werden soll, auf *False* gesetzt, um anzuzeigen, daß der kritische Abschnitt freigegeben wurde.

Es sind nach Dekker noch weitere Softwarelösungen für die Synchronisation von mehreren Prozessen gefunden worden. Die erste Lösung für mehr als zwei Prozesse stammt von Dijkstra [Dijkstra65a]. Weitere Referenzen findet man in [Deitel84].

3. 2. 2. Hardwareunterstützte Synchronisationsmechanismen

Hauptsächlich werden zwei Arten von Synchronisationprimitiven von der Hardware aus angeboten.

- "Test and Set" Instruktion
- "Compare and Swap" Instruktion

Diese beiden Synchronisationsprimitiven werden mit Hilfe von Hardwareunterstützung angeboten. Die Realisation dieser Befehle beruht auf der Möglichkeit den Zugriff des Prozessors auf eine Speicherstelle unteilbar zu gestalten (*read-modify-write* Speicherzugriffe).

3. 2. 2. 1. "Test and Set" Operation

Die "Test und Set" Instruktion wird von vielen Hardwareherstellern in dem Instruktionssatz ihrer Prozessoren zur Verfügung gestellt. Je nach Hersteller setzt diese Instruktion ein oder mehrere Bits im Hauptspeicher auf den Wert 1. Vor dem Setzen der Bits wird überprüft, ob alle Bits vorher den Wert 0 hatten. Das Ergebnis dieses

Vergleich setzt entsprechende Anzeigen im Prozessorstatusregister, wodurch es möglich ist, den Erfolg der "Test and Set" Instruktion programmtechnisch abzufragen. Wichtig bei der "Test and Set" Operation ist, daß der Testvorgang und das Setzen der Bits unteilbar geschieht. Die Operation ist einmal nicht durch Unterbrechungsanforderungen von der Hardware aus unterbrechbar und zum anderen darf es für andere Prozessoren nicht möglich sein, von der mit der "Test and Set" Operation angesprochenen Speicherstelle zu lesen, während eine "Test and Set" Operation ausgeführt wird. Dieser Effekt wird durch Unterbindung der Buszuteilung an andere Prozessoren erreicht.

3. 2. 2. 2. "Compare and Swap" Operation

Die "Compare and Swap" Operation ist eine verallgemeinerte Form der "Test and Set" Operation. Hier wird gewöhnlich ein ganzes Datenwort manipuliert. Der Vorteil der "Compare and Swap" Operation liegt darin, das nicht nur ein Binärwert manipuliert wird, sondern ein ganzes Datenwort manipuliert werden kann.

Die "Compare and Swap" Operation läuft folgendermaßen ab.

Zuerst wird das zu manipulierende Datenwort gelesen und daraufhin verglichen, ob es den erwarteten Wert enthält. Ist dies der Fall, so wird in das Datenwort der neue Inhalt geschrieben und die Instruktion wird erfolgreich beendet. Stimmen das Datenwort und der erwartete Wert nicht überein, so wird das Datenwort nicht verändert und die Instruktion wird mit dem Status nicht erfolgreich beendet.

Der Hauptvorteil der "Compare and Swap" Operation liegt darin, daß man auch komplexere Operationen wie zum Beispiel Ein- und Ausketten in Listen ohne Sperren von Unterbrechungen oder Verwendung von binären Semaphoren mit aktivem Warten durchführen kann.

3. 2. 2. 3. Implementation einer einfachen binären Semaphore

Mit Hilfe der Hardwaresynchronisationsprimitiven läßt sich eine binäre Semaphore schnell realisieren. Der einfachste Algorithmus sieht folgendermaßen aus:

```
Pspin(semaphore)
        DO
        WHILE NOT TestAndSet(semaphore)
END Pspin

Vspin(semaphore)
        semphore = 0
END Vspin
```

Die "Test and Set" Operation wird also solange wiederholt, bis sie erfolgreich ausgeführt wird. Diese einfache Implementierung weist einen entscheidenden Vorteil auf: Sie ist extrem schnell, wenn die Semaphore nicht belegt ist, und stellt damit nur einen sehr geringen Synchronisationsaufwand dar. Anders sieht die Sache aus, wenn die Semaphore belegt ist. In diesem Fall wird die "Test and Set" Operation immer wieder in einer engen Schleife ausgeführt, weshalb diese Art der Implementation der Semaphore auch Spinlock genannt wird.

Ist die Semaphore für lange Zeit belegt, so wird durch das aktive Warten wertvolle Prozessorzeit verschwendet. Bei einem Monoprozessorsystem kann dies je nach Prozessorvergabestrategie mehr oder weniger schwerwiegende Folgen haben. Im günstigsten Fall wird bei einer präemptiven Vergabestrategie der Prozeß nach einer Weile verdrängt und der Prozeß, der die Semaphore belegt, gibt schließlich die Semaphore frei.

Spinlocks weisen eine Anomalie auf, wenn sie im Zusammenhang mit präemptiven Prozessorzuteilungsstrategien verwendet werden. Diese Anomalie tritt beim Zusammenspiel von zwei Prozessen auf:

Ein hochpriorer Prozeß versucht über ein Spinlock in den kritischen Abschnitt, der durch einen niedrigprioren Prozeß belegt ist, zu gelangen. Da der niedrigpriore Prozeß von dem hochprioren Prozeß am Laufen gehindert wird, kann er den kritischen Abschnitt nicht verlassen. Der hochpriore Prozeß verhindert also durch seine hohe Priorität am Spinlock sein eigenes Vorankommen. Eine Lösung dieses Problems kann man dadurch versuchen, daß der hochpriore Prozeß sich von Zeit zu Zeit schlafen legt (d. h. den Prozessor für eine gewisse Zeit aufgibt), damit niedrigpriore Prozesse laufen können. Leider läßt sich auch hier ein Szenario erstellen, wo der hochpriore Prozeß ewig warten muß. Hierzu wird nur noch ein dritter mittelpriorer Prozeß benötigt, der ständig laufbereit ist. Durch diesen Prozeß wird der niedrigpriore Prozeß immer noch am Laufen gehindert, obwohl der hochpriore Prozeß die Zentral-

einheit aufgibt. Die nächste Lösung wäre, die Priorität des im kritischen Abschnitt befindlichen Prozesses zu erhöhen. Diese Erhöhung der Priorität muß dann wieder beim Verlassen des kritischen Abschnitts rückgängig gemacht werden.

Man kann diesen Anomalien begegnen, indem man verhindert, daß einem Prozeß, der erfolgreich eine binäre Semaphore des oben beschriebenen Typs belegt hat, nicht mehr der Prozessor aufgrund von Prozessorvergabestrategien entzogen werden kann. Hierdurch wird gewährleistet, das der kritische Abschnitt möglichst schnell durchlaufen wird. Es versteht sich von selbst, daß die durch diese Art Semaphoren gesicherten Abschnitte möglichst kurz sein sollten.

Hauptvorteil der Implementation der binären Semaphore ist ihre extrem effektive Implementierung (bei freier Semaphore müssen nur zwei Maschineninstruktionen ausgeführt werden). Die Anomalie im Zusammenhang mit prioritätsgesteuerten präemptiven Prozessorvergabestrategien hat gezeigt, das Spinlocks nicht immer unproblematisch sind. Auch eignen sich Spinlocks nicht für die Absicherung kritischer Abschnitte, bei denen unter Umständen lange auf die Freigabe gewartet werden muß.

3. 2. 2. 4. Implementation einer blockierenden Semaphore

Muß man allerdings gegenseitigen Ausschluß für Abschnitte garantieren, bei denen nicht bekannt ist wie lange die Ausführung des Programmstücks in diesen Abschnitten dauert, so ist es oft günstiger den Prozeß für die Zeit zu blockieren, in der er auf die Freigabe des kritischen Abschnitts wartet. Eine Implementation für eine blokkierende Semaphore [Dijkstra69] ist aufwendiger als die Implementation eines Spinlocks, da im Falle der Blockierung des Prozesses zwei Prozeßumschaltungen in Kauf genommen werden müssen.

Mit Hilfe der im Abschnitt 3.2.2.3 beschriebenen binären Semaphoren kann man blockierende und wertebereichbegrenzte blockierende Semaphoren implementieren. Hierbei wird die Datenstruktur, die für die blockierenden Semaphoren benötigt wird, mit Hilfe der Spinlocks abgesichert.

Eine blockierende Semaphore besteht in der Implementation aus drei Teilen:

- binäre Semaphore (Spinlock) zur Absicherung der Datenstruktur der blockierenden Semaphore
- Wert der blockierenden Semaphore
- Warteschlange für die an der blockierenden Semaphore blockierten Prozesse

Der Algorithmus für die blockierende Semaphore sieht dann so für die P() Operation aus:

```
P(semaphore)
    P_Spin(semaphore.semlock)          - Absicherung der Datenstruktur
    IF semaphore.Value = 1 THEN
        Wert = 0                       - Semaphore belegen
        V_Spin(semaphore.semlock)      - Semaphore freigeben
    ELSE
        enqueue(markBlocked(thisProcess),semaphore.queue)
                                       - in die Warteschlange nach einer Strategie (FIFO,LIFO) einhängen
        V_Spin(semaphore.semlock)      - Semaphore freigeben
        Schedule()                     - Prozessor aufgeben
END P
```

Die V() Operation wird dann folgendermaßen implementiert:

```
V(semaphore)
    P_Spin(semaphore.semlock)          - Warteschlange absichern
    semaphore.Value = 1
    IF semphore.queue ≠ EMPTY THEN
        Ready(dequeue(semaphore.queue)) - den ersten Prozeß aus der Warteschlange als
                                          - laufbereit markieren

    V_Spin(semlock)                    - Semaphore wieder freigeben
```

Nach diesem Muster können auch allgemeine wertebereichsbeschränkte Semaphoren realisiert werden.

3. 2. 2. 5. Mehrfach Leser / Exklusiver Schreiber Semaphore

Der durch die eben vorgestellten Synchronisationsmechanismen exklusive Zugriff auf Datenstrukturen ist in vielen Fällen zu restriktiv, da hier alle Zugriffe serialisiert werden. Beim Leser-Schreiber Problem zeigt es sich, daß auch andere Synchronisationsmechnismen neben dem exklusiven Zugriff wünschenswert sind. Solange Daten nur gelesen werden, dürfen mehrere Prozesse gleichzeitig auf die Daten zugreifen. Die lesenden Zugriffe sind also miteinander verträglich. Nur in dem Fall, wo die Daten geändert werden sollen, muß man exklusiven Zugriff garantieren. Also sind schreibende Zugriffe miteinander unverträglich wie auch schreibende und lesende Zugriffe unverträglich sind. Indem man mehreren Prozessen, die die Daten lesen wollen, den Zugriff gleichzeitig erlaubt erhält man einen höheren Grad an Parallelität als bei rein exklusivem Zugriff.

Für die Mehrfach-Leser / Exklusiver-Schreiber Synchronisationsmechanismen gibt es mehrere Möglichkeiten der Implementation. Die jeweiligen Implementationen unterscheiden sich in der Strategie, mit der die Leser bzw. die Schreiber in den kritischen Abschnitt vorgelassen werden.

Courtois, Heymans und Parnas [Courtois71] beschreiben zwei Verfahren, wie man mit Hilfe von binären P und V Operationen den Mehrfach-Leser / Exklusiver-Schreiber Mechanismus realisieren kann. Das erste Verfahren ist sehr leicht überschaubar und erlaubt den lesenden Prozessen immer Zugang zum kritischen Abschnitt, es sei denn, es wurde ein Zugang für einen schreibenden Prozeß gewährt.

Die Implementation sieht folgendermaßen aus:

```
Pread(semaphore)
        P(semaphore.mutex)              - exklusiver Zugriff für den Vergleich
        semaphore.readcount = semaphore.readcount + 1
        IF semaphore.readcount = 1  THEN P(semaphore.write)
                                        - Sicherstellen, daß kein Schreibprozeß vorhanden ist
        V(semaphore.mutex)              - freigeben
END Pread

Vread(semaphore)
        P(semaphore.mutex)              - exklusiver Zugriff für den Vergleich
        semaphore.readcount = semaphore.readcount - 1
        IF semaphore.readcount = 0 THEN V(semaphore.write)
                                        - Zugriff für schreibende Prozesse erlauben
        V(semaphore.mutex)
END Vread

Pwrite(semaphore)
        P(semaphore.write)              - zum Schreiben belegen
END Pwrite

Vwrite(semaphore)
        V(semaphore.write)              - zum Schreiben freigeben
END Vwrite
```

Bei dieser Implementation kann es dazu kommen, daß die Prozesse, die schreibenden Zugang zum kritischen Abschnitt haben wollen, von den lesenden Prozessen ausgehungert werden, da diese den kritischen Abschnitt immer dann betreten können, wenn dieser frei ist, oder sich schon mindestens ein lesender Prozeß im kritischen Abschnitt befindet. Die schreibenden Prozesse müssen warten, bis der kritische Abschnitt frei ist.

Die zweite Lösung von Courtois et al. räumt den schreibenden Prozessen absoluten Vorrang ein. Diese Lösung ist komplizierter, da hier gewährleistet werden muß, daß, sobald ein schreibender Prozeß auf den kritischen Abschnitt wartet, auch alle neu ankommenden lesenden Prozesse warten müssen. Ein schreibender Prozeß kann den kritischen Abschnitt nur betreten, wenn er frei ist. Ein lesender Prozeß kann den kritischen Abschnitt nur betreten, wenn der Abschnitt frei ist oder von einem lesenden Prozeß belegt ist und kein schreibender Prozeß wartet. Die mit P und V Operationen realisierte Lösung für die zweite Form der Leser/Schreiber Synchronisation ist folgendermaßen implementiert:

```
Pread(semaphore)
        P(semaphore.mutex3)                     - verhindert daß Leser und Schreiber simultan auf ein
                                                - V(semaphore.read) warten
        P(semaphore.read)                       - Leseblockade wen Schreiber aktiv sind
        P(semaphore.mutex1)
        semaphore.readcount = semaphore.readcount + 1
        IF semaphore.readcount = 1 THEN P(semaphore.write)- Schreiberblockage wenn Leser aktiv sind
        V(semaphore.mutex1)
        V(semaphore.read)
        V(semaphore.mutex3)
END Pread

Vread(semaphore)
        P(semaphore.mutex1)
        semaphore.readcount = semaphore.readcount - 1
        IF semaphore.readcount = 0 THEN V(semaphore.write)- Letzter Leser startet wartende Schreiber
        V(semaphore.mutex1)
END Vread

Pwrite(semaphore)
        P(semaphore.mutex2)
        semaphore.writecount = semaphore.writecount + 1
        IF semaphore.writecount = 1 THEN P(semaphore.read)- Leser blockieren
        V(semaphore.mutex2)
        P(semaphore.write)
END Pwrite

Vwrite(semaphore)
        V(semaphore.write)
        P(semaphore.mutex2)
        semaphore.writecount  = semaphore.writecount - 1
        IF semaphore.writecount = 0 THEN V(semaphore.read)- Leser freigeben
        V(semaphore.mutex2)
END Vwrite
```

3. 2. 2. 6. Monitore

Auf der Suche nach höheren Synchronisationsmechanismen haben Dijkstra [Dijkstra71], Brinch Hansen[Brinch72,73a] und Hoare[Hoare74] den Monitor vorgestellt. Monitore sind ein Synchronisationsmechanismus, bei dem Daten und Algorithmen zusammengefaßt werden. Auf die Daten eines Monitors kann nur von den im Monitor realisierten Routinen zugegriffen werden (Information hiding). Ein Monitor kann nur über bestimmte Eintrittspunkte betreten werden. An diesen Eintrittspunkten wird gegenseitiger Ausschluß garantiert. Es befindet sich zu einem Zeitpunkt nie mehr als ein Prozeß im Monitor. Wenn eine Monitorroutine feststellt, daß bestimmte Betriebsmittel nicht vorhanden sind, so kann sie die *wait(condition)* Routine aufrufen. Der Aufruf der *wait()* Routine bewirkt, daß der aktuelle Prozeß in die Warteschlange, die mit der Bedingungsvariable *condition* verknüpft ist, eingereiht wird, den Monitor freigibt und sich dann blockiert. Wenn ein Betriebsmittel wieder verfügbar ist, so

kann die Monitorroutine *signal(condition)* aufrufen, wodurch der erste Prozeß in der Warteschlange für *condition* wieder laufbereit wird und sich damit wieder um den Monitor bemühen kann. Die vier Operationen für ENTER, WAIT, SIGNAL und LEAVE können folgendermaßen implementiert werden:

```
Enter(Monitor)
        P(Monitor)              - Monitor betreten / exklusiver Zugriff
END Enter

Wait(Condition)
        disable preemption      - Prozeß darf nicht mehr unterbrochen werden
        enqueue(markBlocked(currentProcess), Condition.queue)
        V(Monitor)              - Monitor wieder freigeben
        Schedule()              - nächsten lauffähigen Prozeß aufgreifen
        P(Monitor)              - wieder um Monitor bewerben
END Wait
Signal(Condition)
        Ready(dequeue(Condition.queue))     - auf Condition wartende Prozesse wieder lauffähig machen
END Signal

Leave(Monitor)
        V(Monitor)              - Monitor wieder freigeben
END Leave
```

3. 3. Grob- und feingranulare Synchronisation

Bei allen Synchronisationsmechanismen hängt der mögliche Grad der Parallelität auf einem Multiprozessor von der Granularität der synchronisierten Objekte ab. Je größer die Bereiche des gegenseitigen Ausschlusses sind, desto geringer wird die mögliche Parallelität, da über größere Bereiche serialisiert wird. Um also größtmögliche Parallelität zu erreichen sollte man die Synchronisationsabschnitte so klein wie möglich wählen. Dies wird auch bei dem Konzept der Verträglichkeitsrelation mit dem abstraktem synchronisiertem Datentyp verfolgt. Da aber die Synchronisation auch Rechenzeit erfordert, ist es nicht immer vorteilhaft die kleinstmöglichen Synchronisationsabschnitte zu wählen, da der Gewinn an Parallelität durch die Verluste bei der Synchronisation zerstört werden kann. Man benötigt also, um kleine Synchronisationsabschnitte wirtschaftlich verwenden zu können, schnelle, leistungsfähige Synchronisationsmechanismen. Die für die Synchronisation benötigte Rechenzeit sollte einen Bruchteil der Rechnenzeit des zu synchronisierenden Abschnitts überschreiten, anderenfalls wird die nutzbare Rechenkapazität durch die Abarbeitung der Synchronisationsalgorithmen aufgezehrt und der Vorteil der Parallelisierung wäre verloren.

3. 4. Verklemmungsproblematik

3. 4. 1. Verklemmung

Verklemmungen treten unter folgenden Voraussetzungen auf[Coffman71]:

- Ein Betriebsmittel ist entweder exklusiv an einen Prozeß vergeben oder noch verfügbar
- Prozesse, die schon Betriebsmittel zugeteilt bekommen haben können weitere Betriebsmittel anfordern
- Betriebsmittel können einem Prozeß nicht entzogen werden. Der Prozeß muß die Betriebsmittel selbst freigeben.
- Es gibt einen Ring von zwei oder mehr Prozessen, die jeweils auf ein Betriebsmittel warten, das das nächste Mitglied des Rings besitzt.

Sind alle diese Punkte erfüllt, dann liegt eine Verklemmung vor. Alle beteiligten Prozesse warten dann auf Betriebsmittel, die sie nie mehr erhalten können.

3. 4. 2. Verklemmungsbehandlung

Es gibt mehrere Verfahren um das Verklemmungsproblem zu behandeln.

– Verhinderung

  Um Verklemmungen zu verhindern, muß man dafür sorgen, daß mindestens eine der oben genannten Bedingungen nicht zutrifft [Havender68]. Die Forderung nach exklusiven Zugriff kann nicht aufgegeben werden, da dies ja das Ziel der Synchronisationsmechanismen ist. Wenn man das Nachfordern von Betriebsmitteln umgehen möchte, so lautet die Lösung, daß alle Betriebsmittel gleichzeitig zu Beginn einer Operation angefordert werden müssen. Diese Lösung hat den entscheidenden Nachteil, daß über lange Zeit unnötig viele Betriebsmittel belegt werden, was dazu führt, das eine sehr starke Serialisierung eintritt. Das Auftreten von zirkularem Warten läßt sich relativ leicht vermeiden, indem man alle Betriebsmittel linear anordnet und die Betriebsmittel nur in aufsteigender Reihenfolge anfordert. Hierdurch kann es nicht zu einem zirkularem Warten kommen. Dieses Verfahren läßt sich dann besonders gut anwenden, wenn die Reihenfolge der Anforderungen bekannt ist, was zum Beispiel bei einem bekannten Programm gegeben ist.

– Erkennung

  Hier wird nur versucht, die aufgetretenden Verklemmungen zu erkennen um dann die Verklemmung aufzulösen. Das Auflösen einer Verklemmung ist ein schwieriges Problem. In vielen Betriebssystemen werden die von der Verklemmung betroffenen Prozesse beendet. Hierbei geht ein Teil der errechneten Ergebnisse verloren. Diese Vorgehensweise ist nur sinnvoll, wenn für nicht wiederholbare Operationen, wie z. B. das Schreiben einer Datei, ein Transaktionskonzept zur Verfügung steht.

– Vermeidung

  Bei der Vermeidung von Verklemmungen werden die Betriebsmittel so an die anfordernden Prozesse ausgegeben, daß mindestens ein Prozeß sicher terminieren kann, wodurch dann auch wieder Betriebsmittel für die anderen Prozesse frei werden (Dijkstras Bankers Algorithm [Dijkstra65]). Ein Vorteil der Verfahren zur Vermeidung von Verklemmungen liegt in einer besseren Ausnutzung der Betriebsmittel, die allerdings durch einen erhöhten dynamischen Aufwand bei der Vergabe von Betriebsmitteln erkauft wird.

Für eine Realisierung einer Verklemmungsbehandlung im einen Programm wie dem Betriebssystem bietet sich also nur das Verfahren der Verhinderung von Verklemmungen an, und dort auch nur das Verfahren der linearen Anordnung von Betriebsmitteln und deren geordnete Anforderung. Ein anderes mögliches Verfahren wäre zwar auch die Vermeidung von Verklemmungen, doch bei jeder Betriebsmittelanforderung muß überprüft werden, ob eine sofortige Zuteilung des angeforderten Betriebsmittels in Zukunft zu einer Verklemmung führen könnte. Die Vermeidung von Verklemmungen ist also mit einem erheblichen dynamischen Aufwand verbunden, bietet aber größere Freiheiten in der Betriebsmittelzuteilung.

4. Parallelisierung des Dateisystems

4. 1. Sicherung der kritischen Abschnitte

Das Unix Betriebssystem wurde ursprünglich für einen Monoprozessor entwikkelt. Hierbei wurde ein sehr einfaches Synchronisationskonzept verwendet. Der Betriebssystemkern von Unix stellt einen Monitor dar, wodurch sich die Synchronisation der einzelnen Algorithmen im Kern äußerst einfach gestaltet. Durch die Monitoreigenschaft des Kerns kann ein Prozeß, der sich gerade im Kern befindet, nicht unterbrochen werden. Diese Eigenschaft wird für die Algorithmen im Kern ausgenutzt. Die Synchronisation im Kern geschieht dann im Prinzip folgendermaßen. Es wird überprüft, ob ein Betriebsmittel verfügbar ist, ist dies nicht der Fall, so wird die Routine *sleep()* aufgerufen, wobei *sleep()* mit einem Parameter versorgt wird, der angibt, auf welches Betriebsmittel gewartet werden soll. Der Aufruf von *sleep()* führt dazu, daß ein anderer lauffähiger Prozeß seine Arbeit fortsetzen kann. Wird dann irgendwann das Betriebsmittel freigegeben, so wird von dem Prozeß, der das Betriebsmittel freigibt, die Routine *wakeup()* aufgerufen. Hier wird derselbe Parameter wie beim *sleep()* für das Betriebsmittel verwendet. Die *wakeup()* Routine versetzt nun alle Prozesse in den Zustand laufbereit, die auf ein Betriebsmittel warten, das durch den Parameter des *wakeup()* Aufrufs gekennzeichnet war. Da *wakeup()* alle Prozesse aufweckt, die auf das Betriebsmittel warten, kann es passieren, daß, wenn ein Prozeß wieder aus der *sleep()* Routine zurückkehrt, das Betriebsmittel schon an einen anderen Prozeß wieder vergeben wurde. Es ist also nötig nach der Rückkehr aus dem *sleep()* wieder die Verfügbarkeit des Betriebsmittels zu überprüfen und unter Umständen wieder *sleep()* aufzurufen, falls das Betriebsmittel nicht verfügbar war. Dieser einfache wie elegante Mechanismus erlaubt es einfache boolsche Variable zu verwenden, um die Verfügbarkeit eines Betriebsmittels anzuzeigen.

Um also im Monoprozessor Unix zum Beispiel von einer Datei zu lesen, war es nur nötig die *Inode* als nicht verfügbar zu markieren und schon konnte man sicher die Daten von der Datei in den Benutzerprozeßadreßraum kopieren. Ein mögliches *sleep()* aufgrund von Ein- und Ausgabevorgängen war nicht schädlich, da alle Prozesse, die auch von derselben Datei lesen wollten eine nicht verfügbare *Inode* vorgefunden haben und sich dementsprechend Mithilfe von *sleep()* blockiert haben. Zu dem Zeitpunkt, wo die Leseoperation abgeschlossen war, wurde die *Inode* wieder freigegeben und alle Prozesse, die auf diese *Inode* gewartet haben, aufgeweckt.

In einem Multiprozessorsystem mit einer präemptiven Prozessorvergabestrategie ist es nicht mehr sinnvoll, die Monitoreigenschaft des Betriebssystemkerns erhalten zu wollen, da dies zu einer sehr starken Serialisierung führen würde. Sinnvoller wäre es, allen Prozessen zu erlauben, den Kern zu betreten. Um die Forderung

erfüllen zu können, müssen die Zugriffe auf die Datenstrukturen des Kerns synchronisiert werden. Es gibt im Unix Dateisystem 10 Datenstrukturen, bei denen die Prozesse synchronisiert werden müssen. Als Synchronisationsmechanismus wurde zum einen das aktive Warten (Spinlock) für kurze kritische Abschnitte, und zum anderen die blockierende Semaphore für längere kritische Abschnitte gewählt.

Im Folgenden werden die Datenstrukturen aufgelistet, bei denen die Zugriffe durch Semaphoren und Spinlocks synchronisiert werden müssen:

- file (Semaphore)
  Zugriff auf Einträge der System-Open-File-Table
- inode (Semaphore)
  Exklusiver Zugriff auf die Inode und damit auf die Datei
- Superblock-Inode-Freilist (Semaphore)
  Zugriff auf die Inode-Freiliste
- Superblock-Block-Freiliste (Semaphore)
  Zugriff auf die Block-Freiliste
- buffer (Semaphore)
  Zugriff auf den Inhalt eines Pufferblocks
- ifreelist (Spinlock)
  Zugriff auf die Inode-Freiliste
- mount (Spinlock)
  Zugriff auf die Mountstruktur
- bfreelist (Spinlock)
  Zugriff auf die Freiliste der Blockpufferverwaltung
- s5ifreelist (Spinlock)
  Zugriff auf die Freiliste der SystemV spezifischen Inodes
- fsinfo (Spinlock)
  Zugriff auf dateisystemspezifische Informationen

Dies sind alle Datenstrukturen innerhalb des Dateisystems für die man die Zugriffe synchronisieren muß. Es darf natürlich nicht vergessen werden, daß sich die Synchronisation auf alle im Kern vorhandenen Datenstrukturen erstrecken muß.

Folgende Hierarchie wurde für die Semaphoren und Spinlocks verwendet:

- file
- inode
- Superblock-Inode-Freiliste
- Superblock-Block-Freiliste
- buffer
- ifreelist
- mount
- bfreelist
- s5ifreelist
- fsinfo

Die Reihenfolge von oben nach unten ist auch die Reihenfolge in der die Semaphoren angefordert werden dürfen. Sie ergibt sich aus dem Ablauf welche Betriebsmittel (hier Datenstrukturen) durch Algorithmen in welcher Reihenfolge benötigt werden. Diese hier dargestellte totale Ordnung der Semaphoren und Spinlocks ist nicht zwingend, da nicht jeder Algorithmus alle Semaphoren und Spinlocks benötigt. Eine partielle Ordnung der Semaphoren und Spinlocks wäre völlig ausreichend. Weil sich aber jede partielle Ordnung in eine totale Ordnung einbetten läßt und eine totale Ordnung auch für eine einfache Laufzeitüberprüfung herangezogen werden kann, bietet sich hier die Verwendung einer totalen Ordnung an.

4. 1. 1. Die *namei()* Routine

Die *namei()* Routine benötigt zur Pfadnamenübersetzung die *Inode*-Datenstruktur. Bevor also auf ein Inhaltsverzeichnis über eine *Inode* zugegriffen wird, wird die blockierende Semaphore der *Inode* belegt. Um das Inhaltsverzeichnis dann zu Durchsuchen, wird die dateisystemabhängige *namei()* Routine aufgerufen. Diese Routine führt dann die geforderte *namei()* Operationen aus. Um jedoch die korrekten Modifikationen im Inhaltsverzeichnis vornehmen zu können, werden zur Ausführung der aller *namei()* Operationen mit Ausnahme der LOOKUP Operation weitere *Inodes* benötigt, deren Semaphoren auch belegt sein müssen, um die verschiedenen Prozesse zu koordinieren. Dies widerspricht der linearen Ordnung der Betriebsmittel, weil hier zwei Betriebsmittel vom selben Typ angefordert werden. In den meisten Fällen führt das Belegen der zweiten Semaphore vom Typ *Inode* nicht zu einer Verklemmungsgefahr, weil eine freie *Inode* belegt wird. Es gibt drei Ausnahmen, wo eine Verklemmungsgefahr besteht.

Die eine Ausnahme ist die DEL Operation. Hier kann es durch Verweise auf Inhaltsverzeichnisse dazu kommen, daß zwei DEL Operationen sich verklemmen, da beide Operationen auf das jeweils andere Inhaltsverzeichnis warten

.

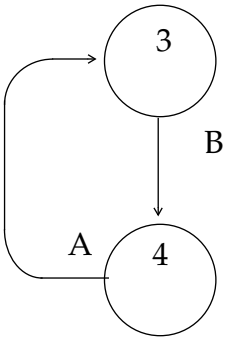


Abb. 5 Verklemmung bei *namei()* Operationen

Ein Beispiel soll diese Situation verdeutlichen:

Prozeß A versucht den Verweis A auf das Inhaltsverzeichnis mit der Inodenummer 3 zu löschen. Hierzu muß die Semaphore der *Inode* mit der Nummer 4 belegt sein, da der Eintrag aus diesem Inhaltsverzeichnis gelöscht werden soll.

Prozeß B versucht den Verweis B auf das Inhaltsverzeichnis mit der Inodenummer 4 zu löschen und belegt seinerseits das Inhaltsverzeichnis mit der Inodenummer 3.

Beide Prozesse müssen die Semaphore der *Inode* des jeweils anderen Inhaltsverzeichnisses belegen. Es kommt zum zirkurlaren Warten - die Verklemmung ist da.

Eine ähnliche Situation tritt bei der LINK und bei der RMDIR Operation auf.

Alle diese Verklemmungsgefahren lassen sich darauf zurückführen, daß das Unix Dateisystem es erlaubt, auch Verweise auf Inhaltsverzeichnisse zu erzeugen. Verbietet man das Erzeugen von Verweisen auf Inhaltsverzeichnisse und das Löschen von Verweisen auf Inhaltsverzeichnisse, so kann es nicht mehr zu der oben beschriebenen Art von Verklemmungen kommen.

Um für die aktuelle Implementation die Verweise auf Inhaltsverzeichnisse erlauben zu können, wird bei den DEL, LINK und RMDIR Operationen die zweite Semaphore vorsichtig, d.h. nur wenn sie frei ist, belegt. Kann die zweite Semaphore nicht belegt werden, so wird die belegt Semaphore wieder freigegeben und die Operation wiederholt, in der Hoffnung, daß die entsprechende Inode wieder freigegeben wurde.

Ein Nachteil dieser Art der Synchronisation der *namei()* Operationen ist, daß die Semaphore der *Inode* des zu durchsuchenden Inhaltsverzeichnisses während des gesamten Suchvorgangs belegt ist.

4. 1. 2. Blockpufferverwaltung

Die Zugriffe auf die Strukturen der Blockpufferverwaltung werden einerseits mit einem Spinlock für die Freiliste der Pufferblöcke und zum anderen mit einer blokkierenden Semaphore für den jeweiligen Pufferblock selbst abgesichert. Manipulationen an den Verkettungen für die Hashlisten werden auch durch das Spinlock für die Freiliste der Pufferblöcke abgesichert, da die Zeiten für die Manipulation der Hashlisten kurz sind. Diese Vorgehensweise unterscheidet sich nicht wesentlich von der ursprünglichen Synchronisation in Unix, dort wurden auch die einzelnen Pufferblöcke exklusive reserviert.

4. 1. 3. Inodepufferverwaltung

Die Synchronisation der Inodepufferverwaltung geschieht analog zur Synchronisation der Blockpufferverwaltung. Die Inodepufferfreiliste wird über ein Spinlock abgesichert und die Synchronisation beim Zugriff auf die Daten einer *Inode* geschieht über eine blockierende Semaphore.

4. 2. Erhöhung der Parallelität der *namei()* Routine

Um die in Abschnitt 4.1.1 beschrieben Nachteile, die durch exklusives Belegen einer *Inode* entstehen bei Operationen auf Inhaltsverzeichnissen zu vermeiden wird ein anderer Synchronisationsmechanismus benötigt.
Die *namei()* Routine bietet acht Operationen an:

- LOOKUP
  Übersetzung eines Pfades in eine *Inode*-Struktureferenz.
- LINK
  Einen Dateiverweis in einen Inhaltsverzeichnis erzeugen
- MKNOD
  Eine Spezialdatei anlegen
- XCREAT
  Eine Standartdatei, die noch nicht existiert, erzeugen
- CREAT
  Eine Standartdatei öffnen, falls sie existiert, und erzeugen, wenn sie nicht existiert
- MKDIR
  Ein Inhaltsverzeichnis erzeugen
- RMDIR
  Ein Inhaltsverzeichnis entfernen
- DEL
  Einen Verweis aus einem Inhaltsverzeichnis löschen

Diese acht Operationen lassen sich in drei Klassen zerlegen. Die erste Klasse wird von den Operationen, die einen Eintrag erzeugen, gebildet. Hierzu gehören LINK, MKNOD, XCREAT, CREAT, MKDIR. Die zweite Klasse besteht aus den Operationen RMDIR und DEL, die einen Eintrag löschen. Zur dritten Klasse gehört nur die LOOKUP Operation.
Aufgrund bestimmter Eigenschaften von Inhaltsverzeichnissen kann man Verträglichkeitsrelationen zwischen den drei Klassen angeben.
Inhaltsverzeichnisse in Unix werden wie normale Dateien gespeichert, allerdings mit zwei Besonderheiten:

- Kein Benutzer, nicht einmal der "Superuser", der alle Rechte besitzt, kann in die Inhaltsverzeichnisse schreiben. Alle Schreiboperationen auf Inhaltsverzeichnisse werden innerhalb des Kerns durch die *namei()* Routine durchgeführt. Diese Eigenschaft erlaubt es ein eigenes Synchronisationsprotokoll für Inhaltsverzeichnisse zu verwenden, ohne Gefahr zu laufen, mit einem Benutzerprozeß, der auch auf Inhaltsverzeichnisse schreibenderweise zugreift, eine Verklemmung hervorzurufen. Der Algorithmus, der die Inhaltsverzeichnisse manipuliert, befindet sich vollständig im Kern und ist somit unter Kontrolle.
- Die zweite wesentliche Eigenschaft ist, daß Inhaltsverzeichnisse niemals kleiner werden, es sei denn, sie werden gelöscht. Das Wachsen der Inhaltsverzeichnisdatei bedeutet, das einmal für die Datei zur Speicherung verwendete Blöcke immer weiter verwendet werden. Es kann also nicht passieren, daß während ein Prozeß die Datei liest, die Information, wo sich die nächsten Datenblöcke auf den Gerät befinden, ändert. Es kann allenfalls ein neuer Datenblock hinzukommen. Damit kann ein Inhaltsverzeichnis immer gelesen werden, ohne daß plötzlich auf ungültige Daten zugegriffen wird.

Setzt man bei der Blockpufferverwaltung voraus, daß sie mehrere lesende Prozesse auf einem Datenblock erlaubt, aber immer nur einen schreibenden Prozeß zur Zeit, so erhält man eine Parallelisierung, weil die *Inode* zu Lesen nicht mehr exklusiv belegt werden muß. Der Fall, daß ein Inhaltsverzeichnis gelesen wird, war schon immer unkritisch. Die Probleme tauchen erst dann auf, wenn man auch in das Inhaltsverzeichnis schreiben möchte.
Die naheliegendste Lösung ist, sobald eine Schreiboperation auf ein Inhaltsverzeichnis wahrscheinlich ist, sich nach dem Muster der Mehrfachlesens und Exklusivschreibens exklusiven Zugriff auf das Inhaltsverzeichnis zu besorgen. Die genauere Analyse zeigt jedoch, daß die drei Klassen von Operationen erheblich verträglicher sind, als es zunächst den Anschein hat.

Geschieht das Schreiben eines einzelnen Eintrags eines Inhaltsverzeichnisses unteilbar, so sind die erzeugenden und die löschenden Operationen miteinander verträglich, da jede Einzeloperation immer einen konsistenten Zustand des Inhaltsverzeichnisses wahrnimmt. Ebenso ist, unter der Voraussetzung des unteilbaren Schreibens eines Inhaltsverzeichnisses, die LOOKUP Operation mit den erzeugenden und löschenden Operationen verträglich.

Es gibt nur zwei Unverträglichkeiten. Die erzeugenden Operationen sind unverträglich mit sich selbst. Diese Unverträglichkeit ist in einer Regel für Inhaltsverzeichnisse begründet. Innerhalb eines Unix Inhaltsverzeichnisses darf jeder Name nur genau ein einziges Mal auftreten. Diese Regel führt zur Unverträglichkeit der erzeugenden Operationen untereinander, weil hier die Algorithmen sicherstellen müssen, daß der jeweilige Pfadname eindeutig innerhalb des Inhaltsverzeichnisses ist. Wären die erzeugenden Operationen verträglich miteinander, so könnte es vorkommen, daß mehrere Einträge unter gleichem Namen innerhalb eines Inhaltsverzeichnisses erzeugt werden.

Die löschenden Operationen sind aus einem anderem Grund unverträglich mit sich selbst. Der Grund hier liegt in der Verwaltung des Verweiszählers innerhalb der Inode. Der Zähler gibt an, in wieviele Verweise auf diese durch die Inode beschriebene Datei sich in den Inhaltsverzeichnissen befinden. Die Verträglichkeit der löschenden Operation würde erlauben, das mehr als ein Prozesse die entsprechenden Eintrag in einem Inhaltsverzeichnis finden könnte und, da dieser Eintrag ja gelöscht werden soll, den Verweiszähler der entsprechenden *Inode* herunterzählt. Da hier dann mehrere Prozesse für ein und denselben Eintrag innerhalb eines Inhaltsverzeichnisses den Verweiszähler dekrementieren, stimmt der Verweiszähler nicht mehr mit der aktuellen Anzahl Verweise auf dieses Datei überein. Der Wert des Zählers wird geringer sein, als die aktuelle Anzahl der Verweise. Dies ist besonders unangenehm, da eine Datei genau in dem Augenblick gelöscht wird, wo ihr Verweiszähler den Stand Null erreicht und keine offenen Referenzen mehr existieren. Die Datei würde zu frühzeitig gelöscht werden.

Nimmt man alle Verträglichkeiten zusammen, dann können simultan (bis auf Synchronisation im Moment des Schreibens der neuen Inhaltsverzeichnisinformationen) mehrere LOOKUP Operationen mit einer löschenden Operation und einer erzeugenden Operation ablaufen können.

Für die von *namei()* zu Verfügung gestellten Operation gibt es also folgende Verträglichkeitsrelationen:

- erzeugende Operationen sind nicht verträglich mit sich selbst
- löschende Operationen sind nicht verträglich mit sich selbst
- LOOKUP Operationen sind verträglich mit sich selbst
- erzeugende, löschende und LOOKUP Operationen sind verträglich

miteinander

4. 2. 1. Parallelisierung der *namei()* Operationen

Um die im vorangegangenen Abschnitt angegebenen Verträglichkeiten auch im *namei*() zu realisieren, benötigt man außer der Semaphore für die *Inode* auch noch zwei weitere Semaphoren. Diese Semaphoren werden für die beiden Unverträglichkeiten beim Erzeugen eines Eintrags und beim Löschen eines Eintrags verwendet.

Die Inode besitzt demnach drei verschiedene Semaphoren:

- DELETE
  Diese Semaphore übernimmt die Absicherung der Löschunverträglichkeit
- CREATE
  Diese Semaphore übernimmt die Absicherung der Erzeugungs-unverträglichkeit
- INODE
  Diese Semaphore übernimmt die Absicherung der wichtigen Inode Felder, wie z. B. Referenzzähler und gewährt exklusiven Zugriff beim Schreiben von Inhaltsverzeichnisdaten

Die Semaphore INODE wird in der dateisystemtypunabhängigen Routine kurzzeitig belegt, um den Referenzzähler der *Inode* zu erhöhen und einige Rechteüberprüfungen vorzunehmen. Danach wird die Semaphore wieder freigegeben. Bevor die dateisystemtypabhängige *namei()* Routine aufgerufen wird, wird die CREATE Semaphore belegt, wenn eine der Operationen LINK, MKNOD, XCREAT, MKDIR ausgeführt werden soll. Für die Operationen RMDIR und DEL wird die DELETE Semaphore belegt. Für die LOOKUP Operation wird keine Semaphore belegt.

Die dateisystemabhängige *namei()* Routine wird mit einer *Inode* aufgerufen, wobei für Operationen, bei denen Einträge im Inhaltsverzeichnis gemacht werden müssen, die CREATE Semaphore belegt ist und für Operationen, bei denen Einträge aus dem Inhaltsverzeichnis entfernt werden müssen, die DELETE Semaphore belegt ist. Bei LOOKUP Operationen ist keine der in der *Inode* vorhandenen Semaphoren belegt. Lediglich der Referenzzähler der *Inode* ist erhöht, um anzuzeigen, daß die *Inode* verwendet wird.

Der eben beschriebene Synchronisationsmechanismus erlaubt es Inhaltsverzeichnisse fast jederzeit zu durchsuchen, da die INODE Semaphore nur für ganz kurze Zeitabschnitte belegt wird und durch die Eigenschaften von Inhaltsverzeichnissen garantiert wird, daß keine falschen Daten aus den Inhaltsverzeichnissen gelesen werden, auch wenn die Inhaltsverzeichnisse simultan zum Suchvorgang erweitert werden. Dies bedeutet eine Parallelisierung der LOOKUP Operation der *namei()* Routine.

In der dateisystemtypabhängigen *namei()* Routine wird das Inhaltsverzeichnis nach der entsprechenden Pfadkomponente durchsucht und die entsprechende Operation ausgeführt. Die Implementation der einzelnen *namei()* Operationen weicht nicht wesentlich von der nicht parallelisierten Version ab, da die unverträglichen Operationen schon in der dateisystemtypunabhängigen *namei()* Routine durch das Belegen der CREATE und DELETE Semaphoren synchronisiert werden. Die wesentlichsten Änderungen laufen darauf hinaus, daß die INODE Semaphore des Inhaltsverzeichnisses belegt werden muß, um auf das Inhaltsverzeichnis schreiben zu können (exklusiver Schreibzugriff und Inode-Zugriff, da sich die Blockabbildungsinformation ändern kann).

4. 2. 2. Die *rmdir* Problematik

Die Operationen LINK, MKNOD, XCREAT, CREAT, MKDIR und DEL lassen sich auf diese Weise einfach parallelisieren. Problematisch ist nur die Operation RMDIR. Ohne Vorkehrungen kann die "rmdir" Semantik beim parallelen Erzeugen von Dateien und Löschen von Inhaltsverzeichnissen verletzt werden. Folgendes Szenario soll das Problem deutlich machen. Ausgegangen wird hierbei von zwei Inhaltsverzeichnissen, von denen das eine, was in dem anderen enthalten ist, gelöscht werden soll.

Weiterhin soll es zwei Prozesse geben: Der erste Prozeß versucht das Inhaltsverzeichnis B zu löschen, während der zweite Prozeß versucht innerhalb des Inhaltsverzeichnisses "B" eine Datei "C" anzulegen versucht. Es kann zu drei möglichen Abläufen kommen, wobei sich zwei als korrekt erweisen und bei einem Ablauf unreferenzierte Dateien entstehen.

Erzeugung eines Eintrags:
P(Inhaltsverzeichnis.CREATE)
Durchsuchen
Inode = iget(ILOCK)
V(Inode.INODE)
P(Inhaltsverzeichnis.INODE)
schreiben
iput(Inhaltsverzeichnis
P(Inode.INODE)
iput(Inode)

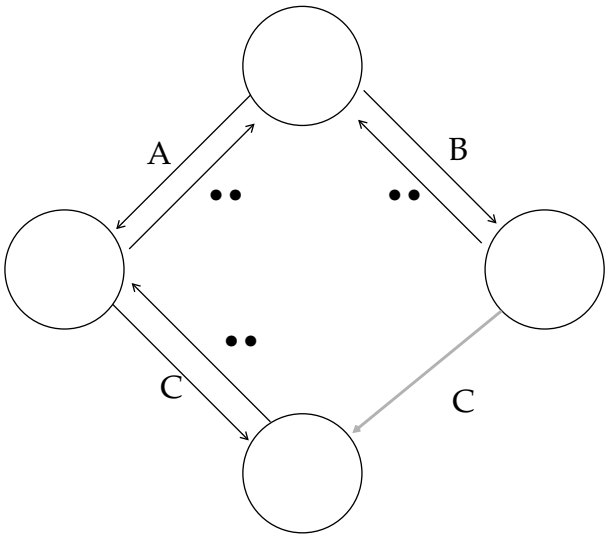
Inhaltsverzeichnis löschen:
P(Inhaltsverzeichnis.DELETE)
Durchsuchen
Inode = iget(ILOCK)
leer verifizieren(Inode)
V(Inode.INODE)
--- genau hier kann in das
--- Inhaltsverzeichnis wieder
--- geschrieben werden
P(Inhaltsverzeichnis.INODE)
schreiben
Verweiszähler korrigieren
iput(Inhaltsverzeichnis)
P(Inode.INODE)
Verweiszähler korrigieren
iput(Inode)

Der fehlerhafte Ablauf entsteht dadurch, daß nachdem überprüft wurde, ob das zu löschende Inhaltsverzeichnis leer ist, die Semaphore für die *Inode* wieder freigegeben werden muß, um in dem übergeordneten Inhaltsverzeichnis den Eintrag zu löschen. Dazu muß natürlich die *Inode* des übergeordneten Inhaltverzeichnisses wieder belegt werden. Durch das Freigeben der *Inode* Semaphore für das zu löschende Inhaltsverzeichnis ist es möglich, daß ein weiterer Prozeß in der Zwischenzeit das zu löschende Inhaltsverzeichnis findet und dort Dateien anlegt. Diese würden, nachdem das Inhaltsverzeichnis gelöscht ist, verloren sein, bis das Programm *fsck* diese Inkonsistenz ("unreferenced directory") feststellt. Um nun diesen Ablauf vorzubeugen wird, noch bevor das zu löschende Inhaltsverzeichnis darauf überprüft wird, ob es leer ist, die CREATE Semaphore auf dem zu löschenden Inhaltsverzeichnis belegt. Diese Semaphore verhindert, daß weitere Einträge in diesem Inhaltsverzeichnis stattfinden können. Somit wird auch die Überprüfung, ob das Inhaltsverzeichnis leer ist, sicher. Die CREATE Semaphore wird also in zwei Fällen angewendet, einmal, um die normalen erzeugenden Operationen wie LINK, MKNOD, XCREAT, CREAT, MKDIR

zu serialisieren, und zum anderen um zu verhindern, daß neue Einträge in einem Inhaltsverzeichnis gemacht werden, das gerade gelöscht werden soll. Die korrekte Implementation für die RMDIR Operation ist also:

Inhaltsverzeichnis löschen:
P(Inhaltsverzeichnis.DELETE)
Durchsuchen
Inode = iget(LOCK,CREATE)
leer verifizieren(Inode)
V(Inode.INODE)
--- genau hier kann in das
--- Inhaltsverzeichnis nicht
--- wieder geschrieben werden
--- da die CREATE Semaphore
--- belegt bleibt
P(Inhaltsverzeichnis.INODE)
Inhaltsverzeichnis schreiben
Verweiszähler Inhaltsverzeichnis korrigieren
iput(Inhaltsverzeichnis)
P(Inode.INODE)
Verweiszähler Inode korrigieren
iput(Inode)

Bei der Durchsicht der Implementation der RMDIR Operation in der System VR3 sind einige Fehler im Zusammenhang mit Verweisen auf Inhaltsverzeichnissen aufgedeckt worden. Laut Semantik ist es dem "Superuser" erlaubt Verweise auf Inhaltsverzeichnisse zu erstellen. Diese Verweise sind grundsätzlich gefährlich, da hierdurch die Baumstruktur des Dateisystems zerstört wird, deswegen die Einschränkung auf den "Superuser". Weiterhin ist auch die Verwaltung der Verweiszähler im Falle von Verweisen auf Inhaltsverzeichnisse inkorrekt. Angenommen man hat folgende Struktur von Inhaltsverzeichnissen, wobei die Beschriftungen der Kanten den jeweiligen Namen des Inhaltsverzeichnisses angeben.



Wenn nun ein weiter Verweis auf das Inhaltsverzeichnis C in dem Inhaltsverzeichnis B gemacht wird, so ist alles vorerst noch korrekt. Probleme tauchen in dem Moment auf, wo eine RMDIR Operation auf einem der beiden Einträge für das Inhaltsverzeichnis C gemacht wird. Bei der RMDIR Operation wird nur geprüft, ob das zu löschende Inhaltsverzeichnis leer ist und welche der beiden "." und ".." Einträge da sind. Ist der ".." Eintrag vorhanden, so wird in dem übergeordneten Inhaltsverzeichnis der Verweiszähler dekrementiert. Diese Vorgehensweise ist falsch, falls die RMDIR Operation über einen zusätzlichen Verweis auf das Inhaltsverzeichnis zugreift (In dem Beispiel wäre das der Pfad B/C). Weiterhin wird der Verweiszähler für das zu löschende Inhaltsverzeichnis immer um 2 dekrementiert, weil davon ausgegangen wird, daß das Inhaltsverzeichnis danach nicht mehr existiert. Allerdings ist diese Annahme nur richtig, wenn keine weiteren Verweise auf das Inhaltsverzeichnis bestehen. Wenn noch Verweise auf das Inhaltsverzeichnis bestehen, so wird das Inhaltsverzeichnis nicht gelöscht und der ".." Eintrag im Inhaltsverzeichnis verweist unter Umständen gar nicht mehr auf ein Inhaltsverzeichnis. Außerdem stimmt der Verweiszähler für das Inhaltsverzeichnis nicht mehr, weil ja der ".." und der "." Eintrag nicht gelöscht wurden. Da in der Praxis sehr selten Verweise auf Inhaltsverzeichnisse verwendet werden und in neueren Systemen das Hilfsmittel des symbolischen Verweises und die RENAME Operation existiert, ist die Verwendung von Verweisen im Zusammenhang mit Inhaltsverzeichnissen überflüssig. Verweise auf Inhaltsverzeichnisse können auch zu Verklemmungen führen, da die *namei()* Routine sich zur Verklemmungsverhinderung darauf verläßt, daß Inhaltsverzeichnisse von der Wurzel her aufgesucht werden. Diese Eigenschaft, das Anfordern ein Betriebsmittels in einer vorgegebenen Reihenfolge, wird durch Verweise auf Inhaltsverzeichnisse verletzt. Wegen des geringen Nutzen von Verweisen auf Inhaltsverzeichnisse und die durch Verweise auf Inhaltsverzeichnisse hervorgerufenen Probleme, sollte man auf Verweise auf Inhaltsverzeichnisse vollständig verzichten. Die im BSD System vorhandenen Hilfsmittel, wie symbolische Verweise und die RENAME Operation erlauben den Verzicht auf Verweise für Inhaltsverzeichnisse.

4. 3. Erhöhung der Parallelität der Blockpufferverwaltung

4. 3. 1. Mehrfach Leser /Exklusiver Schreiber Mechanismus

Um einen noch höheren Durchsatz mit der Blockpufferverwaltung zu erreichen, bietet sich an den Blockpuffer mit einem mehrfach Leser/exklusiver Schreiber Mechanismus auszustatten. Der Vorteil dieses Synchronisationsprotokolls liegt auf der Hand. Es gibt innerhalb des Unix Dateisystems viele Blöcke, die häufig gelesen werden. Hierzu gehören Blöcke von Inhaltsverzeichnissen, Blöcke die Blockabbildungsinformationen enthalten und auch die Datenblöcke von häufig gelesenen Dateien (/etc/passwd, /etc/group, ...).

Um eine mehrfach Leser/exklusive Schreiber Synchronisation auf den Pufferblöcken realisieren zu können, benötigt man einen Spinlock für die Anzeigen im Pufferblockkopf und eine mehrfach Leser/exklusiver Schreiber Semaphore für die Pufferdaten. Das zusätzliche Spinlock wird deswegen benötigt, weil häufig die Gültigkeit der Daten im Pufferblock überprüft werden muß (z. B. bei der Suche nach einem bestimmten Block), ohne auf die Daten zugreifen zu müssen.

4. 4. Erhöhung der Parallelität bei Dateizugriffen

4. 4. 1. Struktur der Inode

Die bisher beschriebenen Änderungen haben an offensichtlichen Stellen im Unix Dateisystem stattgefunden. Innerhalb der *namei()* Routine war es möglich parallelen Zugriff auf die Inhaltsverzeichnisse erhalten. Dies ist deswegen zu realisieren, weil Inhaltsverzeichnisse nur in der Länge zunehmen.

Wenn man die Datenstruktur *Inode* genauer untersucht, so stellt man fest, daß die *Inode* aus drei wesentlichen Teilen besteht:

- Inodeverwaltungsinformation
- Dateiattribute
- Blockabbildungsinformation

In der Inodeverwaltungsinformation sind der Referenzzähler, der angibt, wie oft die Inode zur Zeit im Kern bekannt ist, und diverse Zusatzinformationen, die angeben, welche Daten der Inode aktualisiert werden müssen.

Die Dateiattribute geben den Eigner der Datei, die Gruppe und die Zugriffsrechte an. Weiterhin ist noch vermerkt, um welchen Dateityp es sich handelt, wieviele Verweise auf diese Datei existieren, wie groß die Datei ist und auf welchem Gerät sich die Inode befindet.

Die Blockabbildungsinformation gibt an welche Blöcke in welcher Reihenfolge zu diese Datei gehören.

Man erhält einen höheren Grad an Parallelität, wenn diese drei Teile der Inodestruktur einzeln synchronisiert werden.

Der Referenzzähler in den Inodeverwaltungsinformationen wird sehr häufig verwendet. Er wird jedesmal, wenn die Inode verwendet wird, inkrementiert und, wenn die Inode nicht mehr benötigt wird, dekrementiert. Diese Operation sind sehr kurz und es besteht keine Gefahr des Blockierens. Es bietet sich also an, für die Inodeverwaltungsinformation ein Spinlock zu verwenden, da der Aufwand für eine blokkierende Semaphore ungerechtfertigt hoch für diese Datenstruktur ist.

Auch die Dateiattribute können mit einem Spinlock gesichert werden, denn auch hier werden nur kurzzeitige Modifikationen vorgenommen.

Bei der Blockabbildungsinformation reicht es nicht aus, ein Spinlock zu verwenden, da es hier zu längerfristigen Blockaden aufgrund von Ein- und Ausgabeoperationen kommen kann. Hier würde sich eine Semaphore anbieten. Wenn man allerdings simultanes Lesen einer Datei und exklusives Schreiben unterstützen möchte, so wäre hier die Verwendung einer Leser-Schreiber-Semaphore angebracht, denn die Blockabbildungsinformation kann sich nur ändern, wenn auf die Datei geschrieben wird. Doch würde man hier eine Leser-Schreiber-Semaphore zur Absicherung der Blockabbildungsinformation verwenden, so erzwingt man für die gesamte Datei den mehrfach-Leser/exklusiver Schreiber Mechanismus, obwohl man sich vorstellen könnte, daß solange parallel auf einer Datei gearbeitet werden kann, wie die einzelnen referenzierten Abschnitte nicht miteinander in Konflikt geraten. Hier muß allerdings auch die Frage nach den Nutzen zu Aufwand Verhältnis gestellt werden, denn die Sicherung der Atomarität der Systemaufrufe "read" und "write" würde ein komplettes mehrfach Leser/exklusiver Schreiber Bereichssperrensystem erfordern. Der algorithmische Aufwand für ein solches System ist erheblich, und es erscheint äußert zweifelhaft, ob eine Realisierung dieses Systems noch hinreichend große Parallelitätsvorteile bringt, zumal die Datenstrukturen dieses Synchronisationsmechanismus wieder exklusiven Zugriff erfordern würden, was eine weitere Serialisierung mit sich bringt. Insofern erscheint es sinnvoll, die Blockabbildungsinformationen mit einer mehrfach Leser/exklusiver Schreiber Semaphore abzusichern, da diese Lösung effizient zu implementieren ist. Weiterhin muß man berücksichtigen, daß in einem realem System die Situation, daß mehrere Prozesse gleichzeitig auf eine Datei schreiben und von derselben Datei lesen wollen äußerst selten auftaucht, mit Ausnahme von Datenbanksystemen, die aber eigene Synchronisationsmechanismen verwenden, um die Rekonstruktion der Datenbestände und die konsistente Abspeicherung gewährleisten zu können. Die Einschränkung auf nur exklusives Schreiben erscheint also vor diesem Hintergrund als nicht so gravierend. Der Fall, daß mehrere Prozesse von derselben Datei lesen ist häufiger und hier wird auch die neu eingeführte Parallelität Vorteile bringen.

4. 4. 2. Das Inode-Aktualisierungsproblem

Die Aufspaltung der Inode in drei einzeln synchronisierte Datenstrukturen erlaubt einen höheren Grad an Parallelität, allerdings ergibt sich ein Problem, wenn die Inode wieder auf den Externspeicher geschrieben werden muß. Hierbei müssen die Daten der gesamten Inode konsistent sein. Man kann allerdings nicht exklusiven Zugriff auf die Daten dadurch gewähren, indem man einfach die beiden Spinlocks und die Semaphore belegt. Die Spinlocks können nämlich nicht für die ganze Zeit, in der die Inode auf den externen Datenträger geschrieben wird, belegt werden, da hier Ein- und Ausgabevorgänge auftreten. Es wird also ein anderer Synchronisationsmechanis-

mus benötigt, der es einerseits erlaubt, daß mehrere Prozesse gleichzeitig mit der Inode arbeiten können, andererseits exklusiven Zugriff für einen Prozeß auf die Datenstruktur erlaubt. Ein Synchronisationsmechanismus, der die eben geforderten Eigenschaften erfüllt, ist die Leser/Schreiber Semaphore. Mit Hilfe dieser Semaphore ist es möglich den Zugriff auf die gesamte Inode zu steuern. Alle Operationen, die nur einen Teil der Inode verwenden, also die Inodeverwaltung, die Dateiattribute oder die Blockabbildungsinformation, belegen auch die Aktualisierungssemaphore im Lesemodus. Sollen zu irgendeinem Zeitpunkt die Daten der Inode auf den externen Datenträger herausgeschrieben werden, so wird die Aktualisierungssemaphore der Inode im Schreibmodus belegt, was bewirkt, daß, sobald alle zu Zeit aktiven Prozesse die Inode freigegeben haben, exklusiver Zugiff auf die Inode gewährt wird und die Daten dann konsistent herausgeschrieben werden können.

4. 4. 3. Parallelität und Erhaltung der Unix Systemaufrufsemantik

Mit den in Abschnitt 4.4.1 beschriebenen Änderungen der Synchronisation an der Inode und einer Blockpufferverwaltung, die einen mehrfach Leser / exklusiver Schreiber - Zugriff realisiert, wäre es möglich, mehrere Unix Prozesse simultan auf Dateien zugreifen zu lassen. In der Unix Semantik für Dateien wird festgelegt, daß alle Lese- und Schreibaufrufe atomar sind. Diese Eigenschaft muß natürlich auch für ein Multiprozessorsystem gewährleistet werden. Für eine maximale Parallelität würde man sich wünschen, daß alle Lese- und Schreiboperationen parallel ablaufen, sofern die Atomarität gewahrt bleibt. Für Leseoperationen bleibt die Atomarität grundsätzlich gewahrt. Bei Schreiboperationen muß man dafür sorgen, daß während einer Schreiboperation keine weiteren Lese- und Schreiboperationen auf den betroffenem Bereich der Datei stattfinden. Um also eine maximale Parallelität zu erreichen müßte man ein Bereichssperrensystem verwenden. Die Implementierung eines solchen Systems ist aber relativ aufwendig, und bewirkt, wegen der nötigen Synchronisation, auch eine Serialisierung. Das Problem der Wahrung der Unix Semantik für Lese- und Schreiboperationen ist also sehr mit dem Problem der Absicherung der Blockabbildungsinformation verwandt. Hier gelten auch die Aufwandsabschätzungen, wie für die Absicherung der Blockabbildungsinformation gelten. Weil auch sehr selten parallel auf Dateien geschrieben wird, kann man mit der mehrfach Leser / exklusiver Schreiber Semaphore die Lese- und Schreiboperationen gegeneinander sichern. Da dieses Verfahren schon für die Absicherung der Blockabbildungsinformation verwendet wird, muß man außer der Sicherung der Blockabbildungsinformation keine weiteren Maßnahmen treffen.

4. 5. Verbesserung der Hashlistenorganisation für die Blockpuffer- und

Inodeverwaltung

Für die Effizienz der Blockpufferverwaltung ist die Hashfunktion und die Synchronisation von entscheidender Bedeutung. In der jetzigen Unix System V.3 Version wird folgende Hashfunktion verwendet:

(Blocknummer +Gerätenummer) modulo 2^n

Der Nachteil dieser Hashfunktion liegt auf der Hand. Durch die Addition der Block und Gerätenummer und der Struktur des Dateisystems auf dem externen Datenträger entstehen Kollisionen, die sich in langen Hashlisten niederschlagen. Da von jedem angebundenen Dateisystem der "Superblock", der alle wichtigen Informationen des Dateisystems enthält, im Speicher des Rechners vorhanden sein muß und der "Superblock" immer auf dem selben Blockindex zu finden ist, wird die Häufung der "Superblöcke" nur durch die Gerätenummer verhindert. Leider folgt gleich hinter dem "Superblock" die Inodeliste des Dateisystems. Da Inodes immer von Anfang an belegt werden entstehen hier natürlich Kollisionen mit den Inodeblöcken und Superblockblöcken der anderen Dateisysteme. Auf den ersten Inodeblöcken ist das Wurzelinhaltsverzeichnis des entsprechenden Dateisystems zu finden. Die Blöcke des Wurzelinhaltsverzeichnis sind auch am Anfang des Dateisystems zu finden und kollidieren mit Blöcken von Wurzelinhaltsverzeichnissen auf anderen Dateisystemen. Alle diese Kollisionen werden durch den additiven Teil der Hashfunktion hervorgerufen, die die belegten Blöcke nur unwesentlich verschiebt. Da durch die Kollisionen die Länge der Hashlisten wächst, und eine lineare Suche auf den Hashlisten durchgeführt wird, kann es hier zu Zeitverlusten kommen, zumal der Zugriff auf jeden einzelnen Block auf der Hashliste bis zu drei Vergleichen durchgeführt werden müssen. Je länger also diese Hashlisten werden, umso aufwendiger ist die Suche in diesen linearen Listen. Die Häufung der Kollisionen in bestimmten Bereichen der Hashlisten läßt sich sehr einfach dadurch beseitigen, daß man anstatt der Addition der Gerätenummer und der Blocknummer eine Multiplikation verwendet, wobei man darauf achten sollte, daß nicht mit 0 multipliziert wird. Die Kosten einer Multiplikation im Vergleich zur Addition mögen zwar höher erscheinen, doch durch die Verminderung der Kollisionen wird sehr viel Zeit beim Durchsuchen der Hashlisten gespart. Weiterhin kann man die Hashlisten entsprechend der *Least-Recently-Used* Strategie verwalten, indem man ein in der Hashliste gefundenen Block an den Anfang der Hashliste hängt. Dies ist allerdings nur bei langen Hashlisten interessant und man sollte versuchen, die Hashlisten so kurz wie möglich zu gestalten, was durch eine großzügige Dimensionierung des Feldes der Hashlistenköpfe erreicht werden kann. Heutzutage sind Blockpufferverwaltungen mit einem Puffervolumen von mehreren Megabyte nicht

unüblich. Bei solchen Puffergrößen würde eine Feldgröße von 128 Einträgen, wie es noch in einigen Implementationen üblich ist, und eine Puffergröße von 2 Mb zu Hashlistenlängen im Mittel von 16 Einträgen führen. Diese Listen werden noch länger, wenn Kollisionen auftreten. Idealerweise sollte eine Hashliste aber nur ca. 2-4 Einträge haben, was bei der angegebenen Puffergröße eine Feldgröße von 512 Einträgen bedeuten würde. Der für ein solches Feld benötigte Speicherplatz fällt kaum ins Gewicht, da es sich pro Eintrag um zwei Zeiger handelt.

Für die Inode-Pufferverwaltung gelten ähnliche Überlegungen, wie für die Blockpufferverwaltung. Hier leidet die derzeitige Implementation noch sehr stark unter einer noch einfacheren Hashfunktion, als in der Blockpufferverwaltung. Bei der Inode-Pufferverwaltung wird nur die Inodenummer zur Berechnung der Hashfunktion herangezogen, was zu starken Kollisionen bei häufigen Inodes wie die Inodes von Wurzelinhaltsverzeichnissen (Inodenummer = 2) führt.

Um eine hohe Parallelität beim Durchsuchen der Hashlisten für beide Datenstrukturen zu erreichen sollte jede einzelne Liste eine eigenes Spinlock bekommen.

5. Alternative Konzepte

5. 1. Grobgranulare Synchronisation durch einen Monitor

Die hier vorgestellte feingranulare Synchronisation im Dateisystem beruht auf der Annahme, daß es sehr wenig Konflikte bei der Synchronisation geben wird, da nur über sehr kurze Zeitabschnitte Datenstrukturen mit Spinlocks belegt werden und bei längeren Operationen Semaphoren belegt werden. Wo es möglich war, wurden anstatt von Semaphoren, die exklusiven Zugriff erlauben, mehrfach Leser / exklusive Schreiber Semaphoren verwendet, um auch hier zumindest beim Lesen von Datenstrukturen Parallelität zu erreichen. Es sind auch alternative Konzepte zur Synchronisation in einem speichergekoppelten Multiprozessorsystem entwickelt worden. Bei der von W. Hofmann [Hofmann88] vorgeschlagenen Lösung wird ein grobgranulares Synchronisationsschema verwendet. Der Unixkern wird dabei zur Synchronisation in mehrere Monitore unterteilt. Folgende Teile des Unixsystems werden dabei als Monitor ausgebildet:

- das Dateisystem
- Operationen im Zusammenhang mit gemeinsamen Speicher (shared memory)
- Mitteilungsoperationen
- Semaphoreoperationen
- Gerätetreiber

Da dieses System schon funktionsfähig ist, konnten Messungen gemacht werden, die einen etwas besseren Aufschluß über das Synchronisationsverhalten geben.

Als größtes Problem hat sich der Prozeßwechsel erwiesen. Dieser Vorgang ist extrem zeitaufwendig und hat bei den ersten Implementationen zu sehr großen Synchronisationsaufwand geführt, weil alle Monitore mit der FIFO-Strategie implementiert waren. Die Beobachtung der langen Prozeßwechselzeit führte dann dazu, die FIFO Strategie für Monitore aufzugeben und den Monitor so früh wie möglich freizugeben, sowie den ersten lauffähigen Prozeß aus der Monitorwarteschlange in den Zustand laufbereit zu versetzen. Durch diese Änderungen hat sich das Gesamtverhalten des Systems erheblich verbessert. Die frühzeitige Freigabe des Monitors erlaubte anderen Prozessoren den Zugriff auf den Monitor, noch bevor der Prozessor, der den Monitor gerade freigegeben hat, seinen Prozeßwechsel beendet hat. Weitere Untersuchungen haben gezeigt, daß die Zeit, die der Prozessor in einem Monitor verbringt ungefähr ein Fünftel der Zeit beträgt, die ein Prozeßwechsel benötigt. Diese Beobachtung zeigt, daß man in jedem Fall Prozeßwechsel vermeiden sollte, und so viele Datenstrukturen, wie möglich mit Hilfe von Spinlocks absichern sollte, obwohl der Mechanismus des aktiven Wartens auf den ersten Blick keinen guten Eindruck macht. Bei

einem Zeitverhältnis von 1:5 zwischen Monitorbelegungszeit und Prozeßwechselzeit wäre es schon überlegenswert, ob man mit Hilfe eines Spinlocks den Monitor absichert, zumal bei einer guten Hardwarestruktur (Cache um den Speicherbus zu entlasten) das aktive Warten mit einem Spinlock den Speicherbus so gut wie gar nicht belastet.

6. Zusammenfassung

Um ein Betriebssystem, oder Teile davon, in einem Multiprozessorsystem zu parallelisieren benötigt man leistungsfähige Synchronisationsmechanismen. Mit den verwendeten Synchronisationmechnismen kann man unterschiedlich feine synchronisierte Abschnitte schaffen. Die Wahl der Granularität hat einen unmittelbaren Einfluß auf die Synchronisationseigenschaften des Gesamtsystems. Bei sehr grobgranularen Synchronisationsverfahren werden relativ große Einheiten miteinander synchronisiert. Hier kann es erheblich häufiger zu Serialisierungen kommen, als bei feingranularer Synchronisation. Der Aufwand, der für die verschiedenen Synchronisationsprimitiven getrieben werden muß, bestimmt im Zusammenhang mit dem gewählten Synchronisationskonzept die mögliche Parallelität der Implementation. Es hat sich bei schon laufenden Systemen gezeigt, daß ein Prozeßwechsel sehr teuer ist und man versuchen sollte einen Prozeßwechsel zu vermeiden. Während bei grobgranular synchronisieren Systemen die Wahrscheinlichkeit für einen Konflikt relativ hoch ist, ist für feingranular synchronisierte Systeme diese hohe Konfliktwahrscheinlichkeit nicht zu erwarten. Allerdings gibt es bei feingranular synchronisierten Systemen eine häufigere Verwendung der Synchronisationsprimitiven, wodurch man fordern muß, daß diese Primitiven sehr effizient implementiert werden müssen, damit, selbst wenn keine Konflikte auftauchen, der Effekt der Parallelisierung nicht durch den Aufwand für Synchronisationsprimitiven wieder aufgezehrt wird.

Weiterhin sollte man versuchen, sobald sich Prozesse innerhalb des Kerns synchronisieren müssen, einen Prozeßwechsel möglichst effizient zu gestalten. Hierzu müßte man die Struktur der Betriebssysteme dahingehend ändern, daß man zu Beginn eines Systemaufrufes alle Parameter, die zur Durchführung des Systemaufrufes benötigt werden, in den Kerndatenbereich kopiert, um dann die Algorithmen des Kerns so gestalten zu können, daß es nicht mehr nötig ist während der ganzen Zeit des Systemaufrufs auch den Kontext des Benutzerprozesses mitzuführen. Im allgemeinen reicht es im Kern nämlich aus, nur die Registersätze des Prozessors umzuschalten. Prozeßwechsel dieser Art sind erheblich effizienter, als wenn man für jeden Prozeßwechsel auch den Benutzerkontext mitführen muß. Am Ende der Systemaufrufe kann dann der Benutzerkontext wieder hergestellt werden, um die Ergebnisse des Systemaufrufs vom Kerndatenbereich in den Benutzerdatenbereich zurückzukopieren. Mit diesen kontextarmen Prozessen innerhalb des Kerns kann man sich eher einen Prozeßwechsel erlauben, als mit den mit einem großen Kontext behafteten Prozessen.