Anbindung des Network-File-Systems an den Virtual-File-System-Layer der SB-PRAM

Fortgeschrittenenpraktikum von

Alexander Leidinger
Alexander@Leidinger.net

24. November 2000

Inhaltsverzeichnis

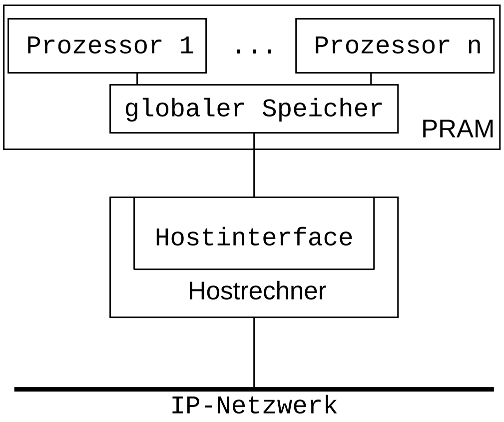

Abbildung 1: Die SB-PRAM.

1 Einleitung

Die SB-PRAM ist ein experimenteller shared-memory Parallelrechner [1], der das PRAM-Modell (parallel random access machine) der theoretischen Informatik [2, 3] realisiert. Die SB-PRAM unterscheidet sich von anderen shared-memory Rechnern dadurch, daß mehrere Programme *parallel* in konstanter Zeit auf den Inhalt derselben Speicherzelle zugreifen können. Die Hardware der SB-PRAM (Abbildung 1) besteht aus dem eigentlichen PRAM Parallelrechner und einem Hostrechner. Die PRAM ist hierbei über das Hostinterface an den Hostrechner angeschlossen, der im laufenden Betrieb der SB-PRAM als IO-Schnittstelle (ähnlich einem Monitor, einer Tastatur und einer Netzwerkkarte bei einem normalen Rechner) dient. Der Hostrechner ist ein PC mit einem handelsüblichen UNIX-Betriebssystem, der an ein IP-Netzwerk angeschlossen ist.

Das Betriebssystem der SB-PRAM ist eine Eigenentwicklung die aus zwei Teilen besteht, dem Hostprogramm [4] und dem PRAMOS [5]. Die Funktionalität des Hostprogramms wird durch Module erweitert, die wie das Hostprogramm als eigenständige Programme auf dem Hostrechner ablaufen. Das Hostprogramm ist für das Initialisieren der PRAM zuständig, erledigt das Laden und Starten von Programmen auf der SB-PRAM und startet bei Bedarf von der PRAM gesteuert Module auf dem Hostrechner.

Das PRAMOS läuft direkt auf der PRAM und stellt den Anwendungsprogrammierern eine UNIX-ähnliche Programmierumgebung zur Verfügung. Sobald ein Programm vom Hostprogramm gestartet wird, übernimmt das PRAMOS das Kommando über die SB-PRAM und das Hostprogramm hat nur noch IO-Operationen durchzuführen.

Als logische Datenübertragungschicht zwischen beiden Teilen des SB-PRAM Betriebssystemes (PRAMOS, Hostprogramm) dient ein Kommunikationsprotokoll; das Memory Transfer Request Protokoll (MTRP) [4, Kapitel 3]. Gesteuert und initiiert wird das MTRP vom PRAMOS.

Da ein Anwendungsgebiet für die SB-PRAM die Nutzung als Datenbankserver ist und eine Datenbank ihre Daten überwiegend auf Festspeichern lagert, wurde auch der Zugriff auf Dateisysteme im PRAMOS vorgesehen. Ein Dateisystem stellt für den Zugriff üblicherweise eine Programmierschnittstelle zur Verfügung (z.B. die UNIX Dateisystem-Zugriffsfunktionen), die das Öffnen, Schließen, Lesen, Schreiben und Löschen von Dateien und Verzeichnissen ermöglicht.

Die ursprüngliche Implementierung des Dateisystemzugriffes ermöglichte den Zugriff auf das SB-PRAM eigene Dateisystem (SB-PRAM FS) und auf das Dateisystem des Hostrechners (Host-FS). Ein Nachteil dieser Implementierung waren die unterschiedlichen Programmierschnittstellen für die jeweiligen Dateisysteme. Ein Applikations-Programmierer hatte deshalb für jede Art von Dateisystem spezielle Zugriffsfunktionen zu verwenden. Ein weiterer Nachteil war, daß z.B. `/foo/bar` und `/example/../foo/bar` physikalisch dieselbe Datei auf dem Speichermedium bezeichnen, beides jedoch für das PRAMOS logisch zwei verschiedene Dateien darstellten. Es konnte dadurch geschehen, daß das Betriebssystem `/example/../foo/bar` nicht als gelöscht ansah, wenn

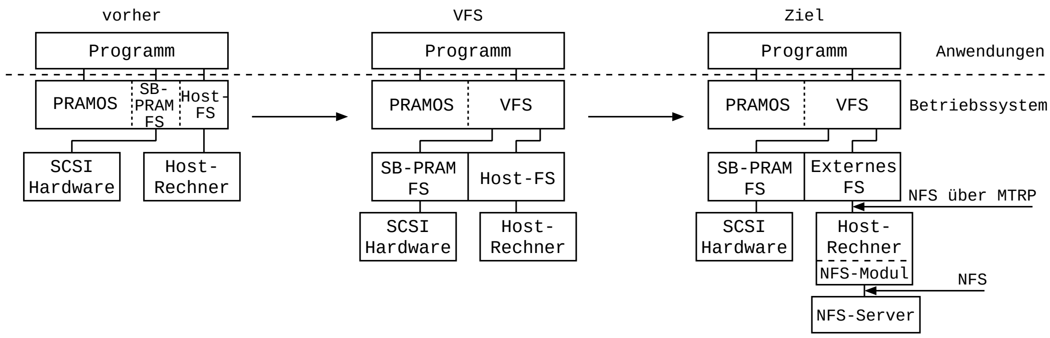

Abbildung 2: VFS Schnittstelle

`/foo/bar` gelöscht wurde. Ein Zugriff mußte deshalb durch absolute Referenzierung von Dateien erfolgen, eine Referenzierung relativ zum aktuellen Verzeichnis war nicht möglich.

Um die oben angesprochenen Nachteile zu beseitigen, wurde parallel zu diesem Fortgeschrittenenpraktikum das Virtual File System (VFS) [8] als Programmierschnittstelle implementiert. Es abstrahiert von der Implementierung und stellt dateisystemübergreifende Funktionen zur Verfügung (UNIX-Dateisystem-Zugriffsfunktionen). Der Applikations-Programmierer hat durch diese Abstrahierung nur die UNIX-Dateisystem-Zugriffsfunktionen zu beachten. Das VFS übernimmt die Aufgabe, die für das jeweilige Dateisystem richtigen Zugriffsfunktionen aufzurufen (Abbildung 2).

Ziel dieses Fortgeschrittenenpraktikums war es, eine Anbindung des Hostrechner-Dateisystems an die VFS-Schicht des PRAMOS zu realisieren. Da eine Änderung des Host-FS wäre ähnlich aufwendig gewesen, wie eine komplette Neuprogramierung, wurde nach einer alternativen Implementierungsmöglichkeit gesucht, die eine größere Funktionalität zur Verfügung stellt. Es wurde das Network File System (NFS) [7] für eine Implementierung gewählt.

Das NFS ermöglicht den Zugriff auf beliebige Dateiserver in einem IP-Netzwerk. Insbesondere kann auch auf den Hostrechner zugegriffen werden. Durch die Wahl des NFS kann, ohne nennenswerten Mehraufwand bei der Implementierung, eine weitaus größere Funktionalität zur Verfügung gestellt werden, da nicht nur auf das Dateisystem des Hostrechners zugegriffen werden kann.

Implementiert wurde der Zugriff auf das NFS in einem Modul des Hostprogrammes. Dabei wurde die Struktur der Module grundlegend überarbeitet. In der ursprünglichen Implementierung der Module mußte für jedes neu programmierte Modul das Hostprogramm in großen Teilen überarbeitet werden. Desweiteren konnte das PRAMOS nicht mit einzelnen Instanzen eines Modules direkt kommunizieren, sondern mußte das Hostprogramm als Vermittler benutzen. Um diese Nachteile zu beheben, entschieden wir uns dazu, die Implementierung der Module dahingehend zu überarbeiten, daß das Hostprogramm keine Kenntnis über die Funktionalität der Module haben, und auch nicht als Vermittler zwischen PRAMOS und einem Modul agieren mußte.

Die vorliegende Arbeit ist wie folgt gegliedert: In Kapitel 2 wird erläutert, wie die Funktionalität der PRAMOS - Hostrechner Kommunikation (MTRP/Module) erweitert wurde. In Kapitel 3 wird die Anbindung des NFS an die SB-PRAM erläutert. In Kapitel 4 wird ein Ausblick über die Ausbaumöglichkeiten der NFS-Anbindung gegeben. Im nachfolgenden Anhang findet sich eine Referenz der neuen bzw. geänderten Funktionen im PRAMOS inklusive einer Kurzbeschreibung der jeweiligen Funktionalität, Übergabeparameter und Rückgabewerte.

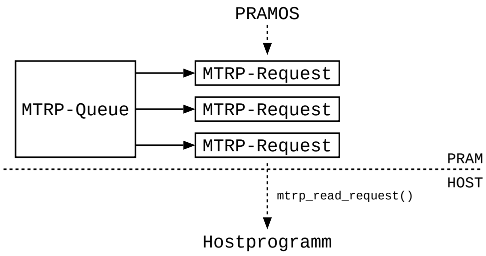

Abbildung 3: Struktur des MTRP

2 PRAMOS - Hostrechner-Kommunikation

2.1 Grundlagen

Die Grundlage der PRAMOS - Hostrechner-Kommunikation ist das Memory Transfer Request Protokoll (MTRP) [4, Kapitel 3]. Es ermöglicht dem PRAMOS das Ausführen von Funktionen auf dem Hostrechner (Remote Procedure Calls, RPC). Dazu werden Funktionsaufrufe und Parameter (MTRP-Requests) vom PRAMOS an die Hostseite des SB-PRAM-Betriebssystems (Hostprogramm/Module) geschickt. Die MTRP-Requests werden vom PRAMOS über eine MTRP-Queue an den Host übergeben. Die MTRP-Queue ist als FIFO-Schlange realisiert, in der Zeiger auf den eigentlichen MTRP-Request-Block abgelegt werden (siehe Abbildung 3). Der MTRP-Request wird von dem Hostprogramm oder einem Modul mittles direktem Speicherzugriff (DSZ) ausgelesen. Nach Abarbeitung solch eines Funktionsaufrufes vom Hostprogramm oder einem Modul wird der Rückgabewert der Funktion in eine vorgegebene Adresse innerhalb des MTRP-Requests im PRAMOS geschrieben, wiederum mittels DSZ. Für den Betriebssystem-Programmierer stehen MTRP-Zugriffsfunktionen zur Verfügung, die sich um die Verwaltung der MTRP-Queue kümmern.

2.2 Modul-Struktur

Die ursprüngliche Implementierung des MTRP sah nur eine einzige MTRP-Queue [4, Kapitel 3] für die Kommunikation zwischen dem PRAMOS und dem Hostrechner vor (siehe Abbildung 4). Falls ein MTRP-Request des PRAMOS für ein Modul bestimmt war, schickte das PRAMOS einen MTRP-Request zum Hostprogramm, das mittels Interprozesskommunikation (IPC) die entsprechende Funktion in dem dafür zuständigen Modul ausführte. Das Modul schrieb dann nach Abarbeitung des MTRP-Requestes den funktionsspezifischen Rückgabewert durch direkten Speicherzugriff in den Speicher der SB-PRAM.

Durch diese Art der Kommunikation konnte das PRAMOS nur über den Umweg über das Hostprogramm mit einem Modul kommunizieren. Außerdem wurden durch diesen Kommunikationsweg unabhängige MTRP-Requests des PRAMOS serialisiert.

Um die oben genannten Beschränkungen aufzuheben, wurde eine neue Implementierung für Module geschrieben, so daß das PRAMOS direkt mit einem Modul kommunizieren kann und unabhängige MTRP-Requests parallel abgearbeitet werden können. Jedes Modul hat in der neuen Implementierung eine eigene MTRP-Queue, damit das PRAMOS mit dem jeweiligen Modul direkt kommunizieren kann. Diese Implementierung impliziert, daß nun das Hostprogramm nur ein Modul unter vielen ist, allerdings mit der Möglichkeit andere Module zu starten.

Solange die bereits vorhandenen Module noch nicht auf die neue Implementierung umgestellt sind (nicht im Rahmen dieser Arbeit), werden beide Implementierungen parallel benutzt.

2.3 Mehrere MTRP-Queues

Die ursprüngliche Implementierung der MTRP-Queue war eine einzige Struktur mit einer zur Compile-Zeit des Betriebssystems fest vorgegebenen Anzahl an Einträgen, deren Adresse im Adressraum des Speichers der SB-PRAM zum Zeitpunkt des Kompilierens des Betriebssystems

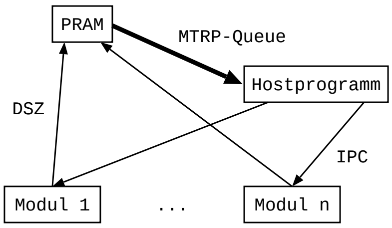

Abbildung 4: Ursprüngliche Implementierung der SB-PRAM - Hostprogramm Kommunikation

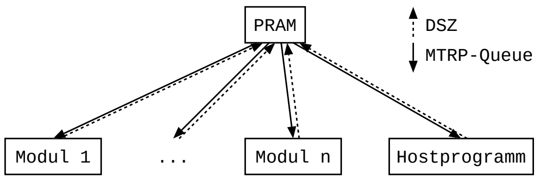

Abbildung 5: Neue Implementierung der SB-PRAM - Hostprogramm Kommunikation

festgelegt wird. Um das MTRP sinnvoll für die Module nutzen zu können, wurde diese Implementierung derart erweitert, daß Speicher für eine MTRP-Queue dynamisch zur Laufzeit des SB-PRAM-Betriebssystems alloziert werden kann und die Anfangsadresse dieser Struktur dem Hostprogramm oder einem Modul bekannt gemacht wird. Durch diese überarbeitete Implementierung des MTRP ist eine Begrenzung der Anzahl der Kommunikationspartner nur noch durch die zur Verfügung stehenden Resourcen gegeben (siehe Abbildung 5).

2.3.1 PRAMOS-Seite

Die neu implementierte MTRP-Funktion `new_mtrp_queue()` legt eine neue MTRP-Queue an. Als Argument übergibt man ihr die maximale Länge der Queue; d.h. die Anzahl der MTRP-Requests [4, S. 19ff] die später gleichzeitig in der MTRP-Queue auf Bearbeitung warten können. Alle bereits vorhandenen MTRP-Funktionen wurden derart überarbeitet, daß sie als zusätzliches Argument eine MTRP-Queue erwarten, über die sie mit dem Host kommunizieren sollen.

2.3.2 Hostseite

Um eine FIFO-Abarbeitung der Anfragen innerhalb einer MTRP-Queue garantieren zu können, wird auf der Hostseite durch die neue Funktion `get_mtrp_queue()` eine Zugriffsstruktur angelegt (im folgenden Hostqueue genannt), in der unter anderem jeweils ein Zeiger auf das nächste/letzte zu bearbeitende Kommando gespeichert ist. Dazu übergibt man der Funktion die Anfangsadresse der MTRP-Queue im Adressraum der SB-PRAM.

Die neue Funktion `mtrp_read_request()`, der man eine Hostqueue als Parameter übergibt, hat als Rückgabewert einen Zeiger auf einen MTRP-Request, der aus der MTRP-Queue ausgelesen wird. Dies kann entweder blockierend oder nicht blockierend geschehen. Bei einem blockierenden Zugriff wartet die Funktion solange, bis sich mindestens ein MTRP-Request in der MTRP-Queue befindet und gibt diesen zurück. Bei einem nicht blockierenden Zugriff liefert die Funktion einen NULL-Zeiger zurück, falls zur Zeit des Funktionsaufrufs kein MTRP-Request auf Abarbeitung wartet.

Nach Abarbeitung eines MTRP-Requests teilt man dem PRAMOS den erfolgreichen oder mißlungenen Abschluß des MTRP-Requests durch den Aufruf der Funktion `mtrp_confirm_job()` mit (siehe auch Abschnitt D.2 und [4, Kapitel 3]).

Es ist nicht möglich, dieselbe MTRP-Queue von mehr als einem Modul auslesen zu lassen, da die für eine MTRP-Queue zuständige Zugriffsstruktur (Hostqueue) lediglich in einem Modul lokal zur Verfügung steht.

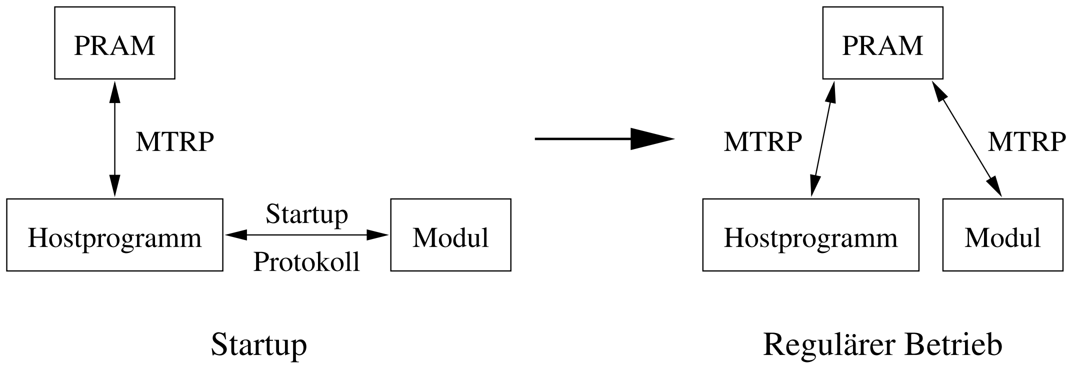

Abbildung 6: Kommunikationskanäle eines Moduls

2.4 Überarbeitung der Module

Da die ursprüngliche Implementierung der Module weder eine direkte Kommunikation des PRAMOS mit einem Modul erlaubte (siehe Abb. 4, S. 6), noch die Konfiguration und Einbindung einzelner Module von dem PRAMOS gesteuert werden konnte, wurde die Implementierung der Module angepasst. Dazu wurde ein neuer MTRP-Request definiert, der das Hostprogramm dazu veranlaßt ein Modul zu starten. Der neue MTRP-Request sieht folgendermaßen aus:

Name:	`MTRP_MAIN_CREATEMODULE`
Parameter:	*par[0]:* Modultyp,
	par[1]: Adresse der MTRP-Queue im Adressraum der SB-PRAM,
	par[2]: Länge des Argumentstrings oder `0`,
	par[3]: Anfangsadresse des Argumentstrings im Adressraum der SB-PRAM.

Das Hostprogramm erwartet als ersten Parameter des Requests den Modultyp[1] (im Falle des NFS-Moduls ist dies `MODULE_PRAM_NFSD`), der die Art des zu startenden Moduls angibt. Außerdem wird für jedes Modul eine eigene MTRP-Queue benötigt, deren Anfangsadresse im Adressraum der SB-PRAM als zweiter Parameter übergeben wird. Erlaubt das Modul die Übergabe von Kommandozeilenargumenten, so ist die Länge der Zeichenkette, welche die Kommandozeilenargumente spezifiziert, als dritter Parameter und die Anfangsadresse im Adressraum der SB-PRAM dieser Zeichenkette als vierter Parameter zu übergeben. Erlaubt das Modul keine Kommandozeilenargumente, oder sollen keine übergeben werden, so ist der dritte Parameter auf 0 zu setzen. Der vierte Parameter wird in diesem Fall ignoriert.

Erhält das Hostprogramm solch einen Request, so versucht es, das spezifizierte Modul mit eventuell übergebenen Kommandozeilenargumenten zu starten. Der Pfadname des Moduls im Dateisystem des Hostrechners ist dabei im Hostprogramm einkompiliert. Die vom Modul benötigte Adresse der MTRP-Queue im Speicherbereich der SB-PRAM wird mittels eines Startupprotokolles übergeben (siehe Abbildung 6).

[1] siehe CVS: PRAM/pramos/common/modules_req.{h,m4}

3 Das NFS-Modul

3.1 Grundlagen

Das Network File System (NFS) [7] ist ein Protokoll, das von einem lokalen Rechner über ein Netzwerk den Zugriff auf Dateien anderer Rechner ermöglicht, sofern die jeweiligen Rechner für diese Dateien den Zugriff erlauben. Dabei werden geöffnete Datei/ein geöffnetes Verzeichnis durch ein NFS-Filehandle eindeutig identifiziert. Der Speicherverbrauch für diese Datenstruktur ist deutlich kleiner (32 Byte), als der Speicherverbrauch bei einer Identifizierung durch den absoluten Pfad einer Datei/eines Verzeichnisses (allein die Speicherung einer Komponente eines Pfadnamens kann 255 Zeichen beanspruchen, wobei auf der SB-PRAM ein Zeichen 4 Byte an Speicherplatzt einnimmt). Auf der SB-PRAM wird das NFS als Subsystem des Virtual File Systems (VFS) [8] realisiert.

3.2 Struktur

Ruft ein Benutzerprogramm auf der SB-PRAM eine Betriebssystemfunktion (syscall) auf, die auf ein Dateisystem zugreift, benutzt die Betriebssystemfunktion dazu eine Funktion des VFS-Layers, der die betreffende dateisystemspezifische Funktion aufruft. Im Falle des NFS schickt diese dateisystemspezifische Funktion einen MTRP-Request an das NFS-Modul auf dem Hostrechner. Das NFS-Modul greift dann mittels des NFS-Protokolls über ein Netzwerk auf ein Dateisystem eines Netzwerkrechners zu. Nach Abarbeitung eines Zugriffs wird das Ergebnis auf dem umgekehrten Weg zurückgeliefert (Abbildung 7).

3.3 Übersicht über die Funktionsweise des NFS-Moduls

Alle Dateisystemanfragen des VFS-Layers der SB-PRAM, für die das NFS-Protokoll Funktionen bereitstellt (z.B. *mkdir*), werden vom NFS-Modul auf dem Hostrechner bearbeitet. Alle anderen VFS-Funktionen (z.B. *sync*)werden auf der SB-PRAM direkt bearbeitet (siehe dazu auch Abschnitt 3.5).

3.3.1 Hostseite (Startup des NFS-Moduls)

Das Startup-Protokoll des NFS-Moduls benutzt die normalen Ein-/Ausgabekanäle (`stdin`, `stdout`) des Hostrechner-Betriebssystems, die mittels einer Pipe in das Hostprogramm umgeleitet werden und so während des Startens des Modules eine Inter-Prozess-Kommunikation (IPC) ermöglichen. Sobald das Modul betriebsbereit ist, werden die Kommunikationskanäle geschlossen und das Modul kann nur noch mit dem PRAMOS kommunizieren. Eine genaue Beschreibung des Startup-Protokolles für das NFS-Modul befindet sich in Anhang B.1 auf Seite 15.

3.3.2 Hostseite (Betrieb)

Damit ein Benutzer auf ein Dateisystem zugreifen kann, muß es erst dem Betriebssystem bekannt gemacht werden. Diese Operation heißt „ein Dateisystem mounten". Die Umkehrung hierzu heißt „ein Dateisystem unmounten", wobei der VFS-Layer dafür zuständig ist, daß ein bei einer Unmount-Operation keine Datei mehr geöffnet ist.

Das NFS-Modul verwaltet auf dem Hostrechner eine Liste gemounteter Dateisysteme. Die Länge der Liste kann dynamisch wachsen und ist somit nur vom zur Verfügung stehenden Speicher beschränkt. Wird ein Dateisystem gemountet (Abschnitt D.1), wird der SB-PRAM eine ID-Nummer (die Position in der Liste der gemounteten Dateisysteme) mitgeteilt, welche dieses Dateisystem eindeutig bestimmt. Ein Listenelement in der Liste der gemounteten Dateisysteme enthält nicht nur Informationen für welches Dateisystem es zuständig ist, sondern auch Authentifizierungsinformationen, die den Zugriff auf dieses Dateisystem erst ermöglichen.

Ein Unmount auf dieses Dateisystem (Abschnitt D.1) löscht zwar alle NFS spezifischen Daten aus der Liste, nicht jedoch das Listenelement, das diese Daten enthielt (Abbildung 8). Da

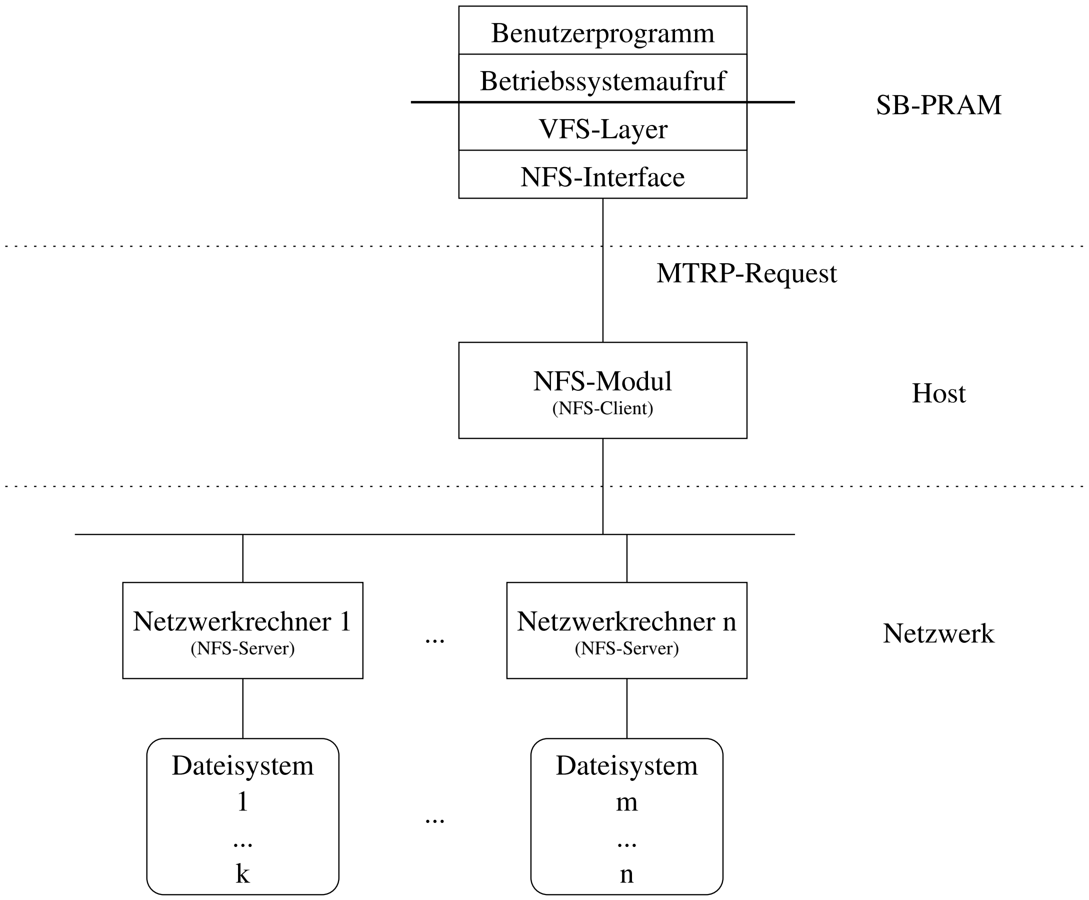

Abbildung 7: Struktur eins NFS Zugriffs

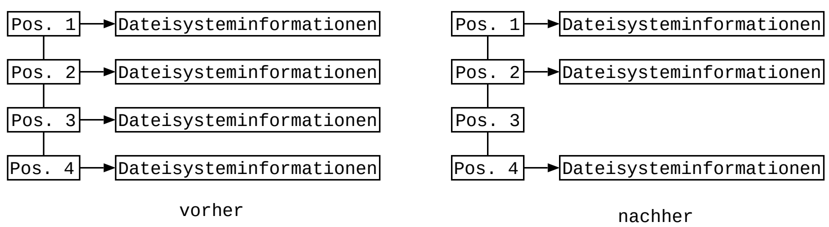

Abbildung 8: Unmount von Dateisystem Nummer 3

NFS-Anfragen von der SB-PRAM die IDs benutzen (welche die Position in der Liste der gemounteten Dateisysteme darstellen), um das Dateisystem anzugeben auf das gerade zugegriffen wird, wird so sichergestellt, daß eine NFS-Operation auf das richtige Dateisystem angewendet wird. Würde auch das Listenelement aus der Liste entfernt, müßten im PRAMOS alle geöffneten Dateien/Verzeichnisse überprüft werden, ob diese noch eine korrekte ID-Nummer enthalten. Enthielten sie keine korrekte ID-Nummer mehr, müßte diese dann korrigiert (dekrementiert) werden.

Strenggenommen stellt die implementierte Art des Unmountens eines Dateisystemes ein Speicherleck dar. Da jedoch das PRAMOS für jedes Programm neu initialisiert wird, und damit auch das Hostprogramm und die Module, wird aus Geschwindigkeitsgründen auf eine „saubere" Implementierung dieser Verwaltung verzichtet. Sollte das PRAMOS jedoch um Mehrbenutzer-Funktionalität erweitert und nicht mehr für jedes Programm neu initialisiert werden, so muß diese Dateisystemverwaltung überarbeitet werden, da sich sonst im Laufe der Zeit (mounten/unmounten) eine größere Anzahl an „toten" Listenelementen ansammeln könnte.

3.3.3 PRAMOS-Seite

Die NFS-Komponente des VFS-Layers im PRAMOS verwaltet eine Liste aktiver `vnode`s [8]. Ein aktiver `vnode` ist hierbei ein `vnode`, der vom VFS-Layer des PRAMOS noch benutzt wird um eine Datei eindeutig identifizieren zu können und somit nicht freigegeben werden kann. Innerhalb eines aktiven `vnode` werden hierbei private Daten des NFS, wie z.B. das zugehörige NFS-Filehandle (Abschnitt D.1) oder die ID-Nummer des zugehörigen Dateisystems (Abschnitt 3.3.2), gespeichert.

Wird auf der SB-PRAM eine Datei oder ein Verzeichnis geöffnet, so wird überprüft ob dafür bereits ein `vnode` existiert. Dies geschieht durch vergleichen des NFS-Filehandles des Verzeichnisses oder der Datei mit den bereits in der Liste der aktiven `vnode`s gespeicherten NFS-Filehandles. Existiert bereits ein `vnode` mit diesem NFS-Filehandle, so wird dieser existierende `vnode` zurückgegeben, ansonsten wird ein neuer `vnode` erzeugt. Dies ist nötig, da der VFS-Layer verlangt, daß lediglich ein `vnode` pro Verzeichnis bzw. Datei existiert.

Da auf der SB-PRAM Aufgaben parallel abgearbeitet werden, muß sichergestellt werden, daß nicht zwei oder mehr Prozessoren gleichzeitig die Liste verändern, damit z.B. nicht zwei `vnode`s angelegt werden, die dieselbe Datei referenzieren. Dies wird durch *Locking* sichergestellt. Die dazu benötigten Funktionen standen dazu bereits zur Verfügung und sind integraler Bestandteil des PRAMOS [6, Kapitel 4]. Hierbei kann durch Zuhilfenahme von *Locks* garantiert werden, daß nur ein Prozessor die Liste bearbeitet.

3.4 Abarbeitung einer NFS-Anfrage der SB-PRAM anhand eines Beispiels

Anhand eines Aufrufs von *mkdir*, der ein neues Verzeichnis im Dateisystem erzeugt, wird im folgenden die Abarbeitung einer NFS-Anfrage verdeutlicht. Zur Vereinfachung wird jedoch nicht im Detail auf die vorhandene Fehlerbehandlung eingegangen. Der VFS-Layer des PRAMOS ruft die Funktion `nfs_mkdir()` mit den Übergabeparametern

- `struct vnode *vp` (Verzeichnis, in dem das neue Verzeichnis erzeugt werden soll)
- `char *nm` (Name des zu erzeugenden Verzeichnisses)
- `mode_t mode` (Zugriffsrechte auf das zu erzeugende Verzeichnis)
- `struct vnode **vpp` (Adresse im Speicher der SB-PRAM, an die der `vnode` des zu erzeugenden Verzeichnisses geschrieben werden soll)

auf [8, Funktion vfs_makedir]. Diese Funktion alloziert nun Speicher für den `vnode` des zu erzeugenden Verzeichnisses. Danach erzeugt sie einen MTRP-Request an das NFS-Modul vom Typ `MTRP_NFS_MKDIR`, dem als Parameter die Anfangsadresse der privaten Daten des `vnodes vp` im Adressraum des Speichers der SB-PRAM, die Länge des Namens des zu erzeugenden Verzeichnisses, die Anfangsadresse des Namens im Adressraum des Speichers der SB-PRAM, die Zugriffsrechte des zu erzeugenden Verzeichnisses und die Anfangsadresse der privaten Daten des neu erzeugten `vnode`s `vpp` im Adressraum des Speichers der SB-PRAM übergeben werden.

Auf dem Hostrechner wird dadurch im NFS-Modul die `mkdir` Funktion aktiviert. Diese Funktion liest die privaten `vnode` Daten des Verzeichnisses (aus dem Speicher der SB-PRAM), in dem das neue Verzeichnis erzeugt werden soll. In diesen Daten steht die ID-Nummer des zugehörigen Verzeichnisbaumes, so daß die zugehörigen NFS-Daten (z.B. Authentifizierung) aus der Liste der gemounteten Dateisysteme des NFS-Moduls ausgelesen werden können. Danach wird der Name des zu erzeugenden Verzeichnisses ausgelesen, vom SB-PRAM-Zeichenkettenformat [5] ins Host-Zeichenkettenformat (ASCII) umgewandelt und zusammen mit den Zugriffsrechten in eine NFS-Anfragestruktur geschrieben.

Nach dem erfolgreichen Abschluß der daraufhin folgenden NFS-Anfrage schreibt das Modul das NFS-Filehandle des erzeugten Verzeichnisses in die privaten Daten des auf der SB-PRAM neu erzeugten `vnode`s.

Auf der SB-PRAM wird nun der `vnode` des neuen Verzeichnisses, nachdem NFS-spezifische Teile des `vnode`s vorher initialisiert wurden, in die Liste der aktiven `vnode`s eingefügt.

3.5 VFS-Funktionen ohne direktes NFS-Äquivalent

Für alle VFS Funktionen ohne Äquivalent im NFS wurden im PRAMOS Funktionen implementiert, die die geforderte Funktionalität, soweit möglich bzw. sinnvoll, durch Benutzung vorhandener Funktionen emulieren:

- `root()` liefert den `vnode` des Wurzelverzeichnisses des Dateisystems zurück.
- `sync()` schreibt noch nicht gespeicherte Daten (Cache) eines gesamten Dateisystems. Da auf der SB-PRAM kein Cache für das NFS existiert, ist diese Funktion immer erfolgreich.
- `access()` überprüft die Zugriffserlaubnis mittels der Zugriffsattribute der Datei bzw. des Verzeichnisses.
- `fsync()` schreibt noch nicht gespeicherte Daten (Cache) einer Datei. Da auf der SB-PRAM kein Cache für das NFS existiert, ist diese Funktion immer erfolgreich .
- `inactive()` markiert eine Datei bzw. ein Verzeichnis als unbenutzt, damit Ressourcen freigegeben werden können. Die Implementierung dieser Funktion entfernt den `vnode` aus der Liste der aktiven `vnode`s.

4 Ausblick

Im folgenden werden ein paar Verbesserungsmöglichkeiten für die vorhandene Implementierung der NFS-Anbindung vorgestellt.

Mehrere Instanzen des NFS-Moduls

Die derzeitige Implementierung des SB-PRAM-NFS benutzt nur ein NFS-Modul und führt somit keine unabhängigen Dateizugriffe parallel durch. Bei einer Erweiterung des SB-PRAM-NFS um die Fähigkeit mehrere NFS-Module auf Dateisysteme arbeiten zu lassen, ist jedoch darauf zu achten, daß in allen NFS-Modulen dieselben Dateisysteme in der gleichen Reihenfolge gemountet und ungemountet werden. Damit wird sichergestellt, daß ein beliebiger (aber fester) NFS-Request an ein NFS-Modul, der das Ziel-Dateisystem mittels einer ID-Nummer referenziert, auch genau dasselbe Dateisystem erreicht wie von einer anderen NFS-Modul aus.

Caching

Die Liste der aktiven `vnode` im PRAMOS kann verbessert werden zu einem `vnode` Cache, so daß das Öffnen von Dateien bzw. Verzeichnissen beschleunigt wird. Dies kann dadurch erreicht werden, daß bei einem Aufruf von `inactive()` ein aktiver `vnode` als inaktiv markiert, jedoch nicht aus der Liste entfernt wird.

Bei einem `lookup()` sollte dabei zusätzlich der zu einem `vnode` zugehörige Pfad in den privaten Daten des `vnodes` gespeichert werden. Ein `lookup()` sucht dann zuerst nach dem Pfad im `vnode` Cache und ruft nur dann die `lookup()` Funktion im NFS-Modul auf, falls der Pfad nicht im `vnode` Cache gefunden wird. Andernfalls braucht nur ein eventuell inaktiv markierter `vnode` als aktiv markiert zu werden. Hierbei muß jedoch beachtet werden, daß bei einem Aufruf von `remove()` bzw. `rmdir()` der jeweilige `vnode` aus dem Cache entfernt wird, da es sonst zu Cacheinkonsistenzen kommt.

Bei dieser Erweiterung kann auch ein Geschwindigkeitsgewinn im normalen Betrieb erwartet werden (wenn z.B. ein bereits geöffnetes Verzeichnis wieder geöffnet wird), da die Erkennung bereits existierender `vnode`s anhand des Pfades realisiert werden würde, und nicht anhand des NFS-Filehandles, welches die `lookup()` Funktion im PRAMOS durch den Aufruf der `lookup()` Funktion des NFS-Moduls erhält.

Desweiteren kann die neuere Version 3 des NFS-Protokolls[2] verwendet werden um Daten zu cachen, da diese Version im Gegensatz zu der von uns benutzten Version 2 des NFS-Protokolls[3] die für einen Datencache benötigten Routinen zur Wahrung der Cachekonsistenz bereitstellt.

Timeoutbehandlung

Eine weitere sinnvolle Erweiterung wäre eine Timeoutbehandlung auf der PRAMOS-Seite. Wird das NFS-Modul auf dem Hostrechner durch irgendwelche äußeren Umstände an der Arbeit gehindert oder terminiert, soll das aufrufende Programm auf der SB-PRAM nicht "endlos" auf das Zurückkehren der Funktion warten.

Geänderte Verwaltung gemounteter Dateisysteme

Sobald das PRAMOS nicht mehr für jedes auszuführende Programm neu initialisiert wird, muß die Verwaltung gemounteter Dateisysteme überarbeitet werden (siehe auch Abschnitt 3.3.2).

Bei einem Unmount müssen in der Liste der aktiven `vnode`s die ID-Nummern der Dateisysteme entsprechend angepaßt werden. Falls die Benutzung mehrerer Instanzen des NFS-Modules implementiert wurde und auch mehrere Instanzen benutzt werden, so muß diese Anpassung in jeder Instanz vorgenommen werden. Danach kann das komplette Listenelement aus der Liste der gemounteten Dateisysteme entfernt werden.

[2] siehe RFC 1813
[3] RFC 1094

A Neue MTRP-Queue Funktionen

A.1 Gemeinsam genutzte Datenstruktur[4]

```
typedef struct pram_mtrp_queue
{
   int queue_len;                         /* must be a power of 2 */
   int free_slots;          /* number of free slots; accessed on pram via tdr/mpadd */
   int write_pointer;   /* pointer to next free slot; accessed on pram via mpadd on host via load
*/
   pram_req_t *volatile *queue;
} pram_mtrp_queue;
```

A.2 SB-PRAM-Seite[5]

Name:	`new_mtrp_queue()`
Parameter:	`int`
Rückgabe:	`pram_mtrp_queue *`
Beschreibung:	Diese Funktion legt eine neue MTRP-Queue an. Sie erwartet als Argument eine Zahl die angibt, wieviele MTRP-Requests gleichzeitig in der neu anzulegenden Queue auf Bearbeitung warten können. Sie liefert einen Zeiger auf die neu erzeugte Queue zurück, falls kein Fehler auftrat, `NULL` sonst.

A.3 Hostseite[6]

A.3.1 Datenstrukturen

```
typedef struct host_mtrp_queue
{
  int queue_len;                          /* must be a power of 2 */
  int next_read;              /* only accessed on host; free_slots is not needed */
  int write_pointer;
  pram_req_t *volatile *queue;
} host_mtrp_queue;

typedef enum { BLOCKING, NON_BLOCKING } mtrp_read_mode;
```

A.3.2 Funktionen

Name:	`get_mtrp_queue()`
Parameter:	`pram_mtrp_queue *`
Rückgabe:	`host_mtrp_queue *`
Beschreibung:	Diese Funktion erwartet die Adresse einer MTRP-Queue auf der SB-PRAM, um eine Struktur anzulegen, die Informationen über die übergebene MTRP-Queue enthält. Mittels dieser Struktur können die nachfolgenden Funktionen auf die Queue auf der SB-PRAM zugreifen. Zurückgegeben wird ein Zeiger auf diese Zugriffsstruktur falls kein Fehler auftrat, `NULL` sonst.

[4] siehe CVS: PRAM/pramos/common/mtrp.h
[5] siehe CVS: PRAM/pramos/pram/kern/hostcommstack.c
[6] siehe CVS: PRAM/pramos/host/mtrp_communication.c

Name:	`mtrp_read_request()`
Parameter:	`host_mtrp_queue *`, `const mtrp_read_mode`
Rückgabe:	`pram_req_t *`
Beschreibung:	Diese Funktion erwartet einen Zeiger auf eine Zugriffsstruktur für eine MTRP-Queue und einen Zugriffsmodus (`BLOCKING`, `NON_BLOCKING`). Wird als Zugriffsmodus `BLOCKING` verwendet, kehrt diese Funktion erst wieder zurück, wenn sich mindestens ein MTRP-Request in der MTRP-Queue befindet. Zurückgegeben wird in diesem Fall die Adresse eines Requests auf der SB-PRAM. Wird als Zugriffsmodus `NON_BLOCKING` benutzt, kehrt diese Funktion auch dann zurück, wenn kein Request auf Bearbeitung wartet. In diesem Fall wird `NULL` zurückgegeben.

Name:	`mtrp_confirm_job()`
Parameter:	`pram_req_t *`, `int`
Rückgabe:	keine
Beschreibung:	Diese Funktion erwartet die Adresse eines Requests auf der SB-PRAM und einen Rückgabewert für diesen Request, der an die SB-PRAM weitergeleitet wird.

B NFS-Modul

B.1 Startupprotokoll zwischen NFS-Modul und Hostprogramm

Die Kommunikation zwischen einem Modul, das in einem Kind-Prozess des Hostprogrammes gestartet wird, und dem Hostprogramm wird über eine Pipe zwischen beiden Prozessen abgewickelt.

NFS-Modul	Hostprogramm
Das NFS-Modul sendet den String "`PRAM-MTRP-Queue?`" an das Hostprogramm.	
	Das Hostprogramm antwortet daraufhin mit der SB-PRAM-Adresse der MTRP-Queue für dieses Modul. Die Adresse wird als String übergeben.
Die Adresse der MTRP-Queue im Adressraum der SB-PRAM wird über `stdin` als String eingelesen.	
Tritt hierbei kein Fehler auf, so gibt es den String "`PNFSD up and running.`" aus.	
	Daraufhin signalisiert das Hostprogramm dem PRAMOS den erfolgreichen Abschluß des `MTRP_MAIN_CREATEMODULE` Requests, baut die IPC-Verbindung ab und das PRAMOS kommuniziert unabhängig vom Hostprogramm mit dem Modul (siehe auch Abbildung 6 auf S. 7).
Tritt jedoch ein Fehler auf, so gibt das Modul stattdessen "`PNFSD failed.`" aus.	
	Dies veranlaßt das Hostprogramm dem PRAMOS eine Fehlermeldung zu übergeben.

B.2 MTRP-Requests für das NFS-Modul

Vorbemerkung zu den MTRP-Requests:

- die Parameter der MTRP-Requests sind angelehnt an die Übergabeparameter des VFS-Layers (`struct vnode`, `struct mounta`, `struct statvfs`, `struct vattr`, `struct uio`), die in [8] detailliert beschrieben sind,
- private `vfs` Daten sind Daten des NFS, die in der Struktur `struct vfs` des VFS-Layers abgelegt sind,
- private `vnode` Daten sind Daten des NFS, die in der Struktur `struct vnode` des VFS-Layers abgelegt sind,
- eine ID-Nummer als Argument wird vor dem Request aus den privaten `vfs` Daten extrahiert,
- Adressenangaben beziehen sich auf den Adressraum auf der SB-PRAM.

Name: `MTRP_NFS_MOUNT`
Parameter: *par[0]:* Länge +1 des Namens des Dateisystem-Hosts,
par[1]: Anfangsadresse des Namens des Dateisystem-Hosts,
par[2]: Länge +1 das Namens des Verzeichnisses,
par[3]: Anfangsadresse des Namens des Verzeichnisses,
par[4]: Adresse der privaten `vfs` Daten (Ergebnis).
Speziell: Die `struct vnode` des VFS-Layers wird nicht benötigt,
Namen werden aus der `struct mounta` extrahiert.

Name: `MTRP_NFS_UNMOUNT`
Parameter: *par[0]:* ID-Nummer des gemounteten Dateisystems.

Name: `MTRP_NFS_STATFS`
Parameter: *par[0]:* ID-Nummer des gemounteten Dateisystems,
par[1]: Adresse einer Struktur `statvfs` (Ergebnis).

Name: `MTRP_NFS_LOOKUP`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Länge des Namens +1,
par[2]: Anfangsadresse des Namens,
par[3]: Adresse der privaten `vnode` Daten (Ergebnis).

Name: `MTRP_NFS_GETATTR`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Adresse einer Struktur `vattr` (Ergebnis).

Name: `MTRP_NFS_SETATTR`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Adresse einer Struktur `vattr`.

Name: `MTRP_NFS_CREATE`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Länge des Namens +1,
par[2]: Anfangsadresse des Namens,
par[3]: Zugriffsrechte für die Datei,
par[4]: Adresse der privaten `vnode` Daten (Ergebnis).

Name: `MTRP_NFS_REMOVE`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Länge das Namens +1,
par[2]: Anfangsadresse des Namens.

Name: `MTRP_NFS_RENAME`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten (Quelle),
par[1]: Kodierung der Längen der beiden Namen (je +1),
par[2]: Anfangsadresse des Namens (Quelle),
par[3]: Adresse der privaten `vnode` Daten (Ziel),
par[4]: Anfangsadresse des Namens (Ziel).
Kodierung: `((strlen(Quellname)+1)<<16)|(strlen(Zielname)+1)`

Name: `MTRP_NFS_MKDIR`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Länge des Namens +1,
par[2]: Anfangsadresse des Namens,
par[3]: Verzeichnisattribute,
par[4]: Adresse der privaten `vnode` Daten (Ergebnis).

Name: `MTRP_NFS_RMDIR`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Länge des Namens +1,
par[2]: Anfangsadresse des Namens.

Name: `MTRP_NFS_READDIR`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Adresse einer Struktur `uio` (Ergebnis),
par[2]: Adresse eines Datentyps `int` (EOF).
Bedingung: `uio_resid` ≥ 64 (NFS Beschränkung).

Name: `MTRP_NFS_LINK`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten (Quelle),
par[1]: Adresse der privaten `vnode` Daten (Ziel),
par[2]: Länge des Namens +1,
par[3]: Anfangsadresse des Namens.

Name: `MTRP_NFS_READ`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Adresse einer Struktur `uio` (Ergebnis).

Name: `MTRP_NFS_WRITE`
Parameter: *par[0]:* Adresse der privaten `vnode` Daten,
par[1]: Adresse einer Struktur `uio`.

C Universell einsetzbare Datenstrukturen

C.1 PRAMOS[7]

Die vorliegende Implementierung der NFS Anbindung benutzt eine Liste um bereits geöffnete Dateien und Verzeichnisse erkennen zu können. Diese Liste wurde als doppelt verkettete Liste implementiert und erlaubt die allgemeine Anwendung durch austauschbare Vergleichsroutinen.

Name:	`List_Init()`
Parameter:	`struct List **`
Rückgabe:	`int`
Beschreibung:	Diese Funktion legt eine neue Liste an. Ein Zeiger auf diese Liste wird an die Adresse geschrieben, die übergeben wird. Bei Erfolg wird `1` zurückgegeben, `0` sonst.

Name:	`List_SetCompare()`
Parameter:	`struct List *`, `int (*)(unsigned long *, unsigned long *)`
Rückgabe:	`void`
Beschreibung:	Diese Funktion weist der Liste eine neue Vergleichsfunktion zu.

Name:	`STD_List_Compare()`
Parameter:	`unsigned long *, unsigned long *`
Rückgabe:	`int`
Beschreibung:	Standard Vergleichsfunktion, falls nicht mittels `List_SetCompare()` eine neue Vergleichsfunktion zugewiesen wird.
Bemerkung:	Diese Funktion ist nur der Funktion `List_Init()` bekannt, da `STD_List_Compare()` als `static` deklariert ist.

Quelltext:	`if (first == second) return 0;`
	`if (first < second) return -1;`
	`return 1;`

Name:	`List_Add()`
Parameter:	`struct List *, unsigned long *`
Rückgabe:	`struct List_Node *`
Beschreibung:	Diese Funktion fügt einen Zeiger auf beliebige Daten in die übergebene Liste ein. Zurückgegeben wird `NULL` im Fehlerfall, ein Zeiger auf ein Listenelement, das den Zeiger auf das Datum enthält, sonst.

Name:	`List_Head()`
Parameter:	`struct List *`
Rückgabe:	`struct List_Node *`
Beschreibung:	Diese Funktion gibt einen Zeiger auf das Kopfelement der Liste zurück, falls die Liste nicht leer ist, `NULL` sonst.

Name:	`List_Tail()`
Parameter:	`struct List *`
Rückgabe:	`struct List_Node *`
Beschreibung:	Diese Funktion gibt einen Zeiger auf das Endelement der Liste zurück, falls die Liste nicht leer ist, `NULL` sonst.

[7] siehe CVS: PRAM/pramos/pram/vfs/list.c

Name: `List_Next()`
Parameter: `struct List *`, `struct List_Node *`
Rückgabe: `struct List_Node *`
Beschreibung: Diese Funktion gibt einen Zeiger auf das Nachfolgerelement des übergebenen Elementes zurück., falls ein Nachfolgerelement existiert, `NULL` sonst.

Name: `List_Previous()`
Parameter: `struct List *`, `struct List_Node *`
Rückgabe: `struct List_Node *`
Beschreibung: Diese Funktion gibt einen Zeiger auf das Vorgängerelement des übergebenen Elementes zurück, falls ein Vorgängerelement existiert, `NULL` sonst.

Name: `List_Search()`
Parameter: `struct List *`, `unsigned long *`
Rückgabe: `struct List_Node *`
Beschreibung: Diese Funktion sucht, unter Zuhilfenahme einer vorher spezifizierten Vergleichsfunktion (`List_SetCompare()`), nach einem Datum in der Liste, das zu dem übergebenen Datum äquivalent ist („äquivalent" deshalb, da dies von der speziellen Suchfunktion abhängt, siehe `STD_List_Compare()`). Zurückgegeben wird das Listenelement, welches das gesuchte Datum enthält, oder `NULL`, falls das gesuchte Datum nicht in der Liste ist.

Name: `List_Data()`
Parameter: `struct List *`, `struct List_Node *`
Rückgabe: `unsigned long *`
Beschreibung: Diese Funktion gibt das in einem Listenelement enthaltene Datum zurück.

Name: `List_SetData()`
Parameter: `struct List *`, `struct List_Node *`, `unsigned long *`
Rückgabe: `unsigned long *`
Beschreibung: Diese Funktion ersetzt das Datum eines Listenelementes, zurückgegeben wird das alte Datum.

Name: `List_Size()`
Parameter: `struct List *`
Rückgabe: `unsigned int`
Beschreibung: Diese Funktion gibt die Anzahl der in der Liste enthaltenen Elemente zurück.

Name: `List_Delete()`
Parameter: `struct List *`, `struct List_Node *`
Rückgabe: `unsigned long *`
Beschreibung: Diese Funktion entfernt ein Listenelement aus der Liste, zurückgegeben wird das in dem entfernten Listenelement enthaltene Datum.

C.2 Host[8]

Das NFS-Modul auf dem Hostrechner benutzt eine Liste um bereits gemountete Dateisysteme verwalten zu können. Diese Liste ist analog zu der Liste im PRAMOS realisiert, wobei eine erweiterte Funktionalität wie folgt implementiert ist:

[8] siehe CVS: PRAM/pramos/host/nfs/list.c

Name: `List_SetConstruct()`
Parameter: `struct List *`, `int (*)(unsigned long **)`
Rückgabe: `void`
Beschreibung: Diese Funktion weist der Liste einen neuen Konstruktor zu, der auf jedes neu hinzugefügte Datum angewendet wird.

Name: `STD_List_Construct()`
Parameter: `unsigned long **`
Rückgabe: `int`
Beschreibung: Standard Konstruktor, falls nicht mittels `List_SetConstruct()` ein neuer Konstruktor zugewiesen wird. Zurückgegeben wird `0` bei Erfolg.
Bemerkung: Diese Funktion ist nur der Funktion `List_Init()` bekannt, da `STD_List_Construct()` als `static` deklariert ist.
Quelltext: `return 0;`

Name: `List_SetDestruct()`
Parameter: `struct List *`, `unsigned long *(*)(unsigned long *)`
Rückgabe: `void`
Beschreibung: Diese Funktion weist der Liste einen neuen Destruktor zu, der auf jedes zu entfernende Datum angewendet wird.

Name: `STD_List_Destruct()`
Parameter: `unsigned long *element`
Rückgabe: `unsigned long *`
Beschreibung: Standard Destruktor, falls nicht mittels `List_SetDestruct()` ein neuer Destruktor zugewiesen wird. Zurückgegeben wird `0` bei Erfolg.
Bemerkung: Diese Funktion ist nur der Funktion `List_Init()` bekannt, da `STD_List_Destruct()` als `static` deklariert ist.
Quelltext: `return element;`

Name: `List_GetItem()`
Parameter: `struct List *`, `unsigned long pos`
Rückgabe: `struct List_Node *`
Beschreibung: Diese Funktion gibt das Listenelement zurück, das sich an Position `pos` befindet, sofern `pos` $\in \{1, ..., List\_Size()\}$ und `List_Size()` > 0.

Name: `List_Remove()`
Parameter: `struct List **`
Rückgabe: `void`
Beschreibung: Diese Funktion entfernt die Liste komplett aus dem Speicher, es kann nun nicht mehr mit dieser Liste gearbeitet werden.

D Glossar

D.1 Dateisystem-spezifisch

Virtual File System (VFS)

Das Virtual File System [8] stellt eine einheitliche Schnittstelle zwischen Betriebssystem und verschiedenen Dateisystemen zur Verfügung, so daß man mittels einer einheitlichen Benutzerschnittstelle auf die verschiedenen Dateisysteme zugreifen kann.

Network File System (NFS)

Das Network File System [7] ist ein Protokoll, das den Zugriff auf Dateien anderer Rechner in einem Netzwerk ermöglicht, sofern die jeweiligen Rechner für diese Dateien den Zugriff erlauben.

NFS-Filehandle

Ein NFS-Filehandle ist eine Datenstruktur, die ein Verzeichnis oder eine Datei auf die mittels des NFS zugegriffen wird, eindeutig bestimmt.

Dateisysteme mounten

Ein Dateisystem mounten heißt, das Dateisystem dem Betriebssystem bekannt zu machen, damit ein Benutzer darauf zugreifen kann.

Dateisysteme unmounten

Ein Dateisystem unmounten ist die inverse Funktion zu Dateisysteme mounten.

D.2 SB-PRAM-spezifisch

Memory Transfer Request Protokoll (MTRP)

Das Memory Transfer Request Protokoll [4, Kapitel 3] ermöglicht dem PRAMOS auf dem Hostrechner Funktionen auszuführen (Remote Procedure Calls, RPC).

MTRP-Queue

Die MTRP-Queue [4, Kapitel 3] ist eine FIFO-Schlange, in die MTRP-Requests des PRAMOS an die Hostseite des SB-PRAM Betriebssystems eingefügt werden.

MTRP-Request

Ein MTRP-Request ist ein innerhalb einer MTRP-Queue gespeicherter Funktionsaufruf des PRAMOS an die Hostseite des SB-PRAM Betriebssystems.

MTRP-Funktionen

Die MTRP-Funktionen stellen eine einheitliche Schnittstelle zur Verfügung, um das MTRP zu benutzen. Sie ermöglichen das Erzeugen von MTRP-Queues und MTRP-Requests

Hostprogramm

Das Hostprogramm [4] regelt die Kommunikation der SB-PRAM mit dem Benutzer und ist für das Starten der Programme auf der PRAM sowie der Module auf dem Hostrechner verantwortlich.

Modul

Es gibt zwei Arten von Modulen, die alte und die neue Implementierung. Diese Beschreibung bezieht sich auf die neue Implementierung. Module sind vom PRAMOS aufrufbare Programme auf dem Hostrechner, die nach dem Starten durch das Hostprogramm autonom agieren.

Abbildungsverzeichnis

Literatur

[1] P. Bach, M. Braun, A. Formella, J. Friedrich, Th. Grün, C. Lichtenau. Building the 4 Processor SB-PRAM Prototype. Hawaii International Conference on System Sciences, Hawaii, USA, 1996.

[2] Jörg Keller. Zur Realisierbarkeit des PRAM Modells. Dissertation, FB 14 Informatik, Universität des Saarlandes, 1992.

[3] Abhiram G. Ranade, Sandeep N. Bhatt, S. Lennart Johnson. The Fluent Abstract Machine. In *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, pages 71-93, Cambridge, MA, 1988. MIT Press.

[4] Stefan Franziskus. Entwurf und Implementation der Hostseite des PRAM-Betriebssystems PRAMOS. Diplomarbeit, FB14 Informatik, Universität des Saarlandes, Mai 1997.

[5] Andreas Crauser. Entwurf und Implementierung des SB-PRAM-Betriebssystems PRAMOS: SB-PRAM-Seite. Diplomarbeit, FB 14 Informatik, Universität des Saarlandes, 1996.

[6] Jochen Röhrig. Implementierung der P4-Laufzeitbibliothek auf der SB-PRAM. Diplomarbeit. FB 14 Informatik, Universität des Saarlandes, 1996.

[7] SUN Microsystems Inc., Request for comment (RFC) 1094, Network File System V2, 1989.

[8] Daniel Rock. VFS. Manuskript, Lehrstuhl Prof. Paul, FB14 Informatik, Universität des Saarlandes, 1998.