Humboldt-Universität zu Berlin
Institut für Informatik

Effiziente dynamische Verwaltung von gemeinsam genutztem Speicher in einem verteilten System

Diplomarbeit

Niklas Hahn

Betreuung: Frank Müller, Ph.D.

Berlin, 27. März 2001

Inhaltsverzeichnis

Einleitung **1**

1 Grundlagen **3**
1.1 Verteilte Systeme 3
1.2 Distributed Shared Memory 4
1.3 DSM-Threads 7

2 Die Verwaltung von DSM **9**
2.1 Statischer DSM 10
2.2 Dynamischer DSM 11
2.2.1 Zeiger im dynamischen DSM 13
2.2.2 Zugriff auf Objekte im dynamischen DSM 14

3 Semi-Dynamischer DSM **15**
3.1 Der Aufbau des DSM-Heap 15
3.2 Verwaltung des Speichers im DSM-Heapblock 16
3.3 Die Speicherverwaltung von Kernighan/Ritchie 17
3.3.1 Veränderungen zur Verwendung im DSM-Heap 19
3.3.2 Vorteile bei der Verwendung im DSM-Heap 20
3.4 Konsistenz des Speichers im DSM-Heap 21
3.5 Vor- und Nachteile des semi-dynamischen DSM 22

4 Dynamischer DSM **24**
4.1 Veränderung des Aufbaus des DSM-Heap 25
4.2 Verwaltung der DSM-Heapblöcke 26
4.2.1 Anforderungen an den Aufbau des DSM-Heap 27
4.2.2 Informationen über die DSM-Heapblöcke 27
4.2.3 Zeitpunkt der Vervollständigung des DSM-Heap 28
4.3 Verwaltung des Speichers in den Heapblöcken 29
4.4 Zeiger in die DSM-Heapblöcke 29
4.4.1 Die Zeigertabelle 30
4.4.2 Übersetzung der Zeiger 32
4.5 Vor- und Nachteile dieser Lösung 33

5 Effizienter dynamischer DSM **36**
5.1 Erstellung von DSM-Heapblöcken 36
5.1.1 Plazierung der DSM-Heapblöcke im Speicher 37
5.1.2 Risiken von `mmap` 38
5.2 Verwaltung der DSM-Heapblöcke 40
5.2.1 Transfer der Informationen über die Heapblöcke 40
5.2.2 Vollständigkeit der Heap-Informationen 44
5.2.3 Zeitpunkt der Vervollständigung des DSM-Heap 47
5.3 Die Seitentabelle 48
5.3.1 Konsistenz der Seitentabelle 49
5.3.2 Aktualisierung der Seitentabelle 50
5.4 Verwaltung des Speichers in den Heapblöcken 52
5.4.1 Auslagerung der Freelist 53
5.5 Verwaltungsblöcke 55
5.5.1 Verwaltung des Speichers in den Verwaltungsblöcken . 56
5.6 Vor- und Nachteile des effizienten dyn. DSM 58
5.7 Zusammenfassender Vergleich der Lösungen 59

6 Performancetests **60**
6.1 FFT und LU 60
6.2 Ein spezielles Testprogramm 62

Zusammenfassung **69**

A Schnittstellenbeschreibung **72**

Literaturverzeichnis **74**

Einleitung

Das Ziel dieser Diplomarbeit ist es, eine möglichst effiziente dynamische Verwaltung für einen gemeinsam genutzten Speicher in einem verteilten System zu entwerfen und zu implementieren.

Die Speicherverwaltung soll dabei im Rahmen der *DSM-Threads* [1, 2] realisiert werden. DSM-Threads sind ein Laufzeitsystem, in dem Threads in einem homogenen, verteilten System ausgeführt werden können. DSM-Threads basieren auf POSIX Threads (Pthreads) [3] und verfügen über einen virtuellen, verteilten, gemeinsam genutzten Speicher (DSM - distributed shared memory).

Der Begriff Speicherverwaltung bezieht sich hierbei nicht nur auf die Verwaltung des Speichers innerhalb des DSM, sondern ebenso auf die Erstellung und Organisation des DSM selbst.

In der ursprünglichen Implementation der DSM-Threads handelte es sich bei dem verteilten, gemeinsamen Speicher ausschließlich um statischen Speicher, d.h. es war nicht möglich diesen Speicher während der Laufzeit in seiner Größe zu verändern. Vom Autor der vorliegenden Diplomarbeit wurde dieser Speicher bereits im Rahmen einer Studienarbeit [4] um eine dynamische Komponente ergänzt, d.h. dem Benutzer der DSM-Threads wurde die Möglichkeit gegeben, während der Laufzeit verteilten, gemeinsamen Speicher vom System anzufordern sowie zurückzugeben zu können. Auf dieser Grundlage soll nun innerhalb dieser Arbeit eine bzgl. der Effizienz verbesserte Methode zur Realisierung eines dynamischen, verteilten, gemeinsamen Speichers entwickelt werden.

Zu verteilten Systemen mit gemeinsamem Speicher und DSM-Threads wird in Kapitel 1 eine kurze Einführung gegeben. Kapitel 2 deutet die Schwierigkeiten an, die sich aus der Verteilung von gemeinsamem Speicher für dessen Verwaltung ergeben. In Kapitel 3 wird ein zwar einfacher und effizienter, dafür allerdings nur „fast“ dynamischer Lösunganſatz zur Verwaltung von verteiltem Speicher gezeigt, bevor in Kapitel 4 die in der oben erwähnten Studienarbeit entwickelte Lösung mit ihren Nachteilen bzgl. der Effizienz vorgestellt wird. Schließlich wird in Kapitel 5 eine in der Effizienz gegenüber den in den vorhergehenden Kapiteln dargestellten Verwaltungsmethoden optimierte Lösung entwickelt. Performancetests sollen in Kapitel 6 den Gewinn an Effizienz belegen.

Mein Dank geht in allererster Linie an Frank Müller, für die Anregung zu dieser Diplomarbeit und die tatkräftige Unterstützung, zu der er im Verlauf ihrer Erstellung jederzeit bereit war.

Kapitel 1

Grundlagen

1.1 Verteilte Systeme

„*Ein verteiltes System ist eine Sammlung voneinander unabhängiger Computer, die dem Benutzer des Systems den Eindruck vermitteln, es handele sich um einen einzigen Computer*“ [5].

Als verteilte Systeme kann man dabei sowohl Systeme verstehen, in denen eine gewisse Anzahl von einzelnen Rechnern über ein Netzwerk miteinander verbunden sind (Abbildung 1.1 A), als auch Systeme in denen eine gewisse Anzahl von CPUs miteinander zu einem einzigen Rechner verbunden sind (Abbildung 1.1 B). Eine Anwendungsmöglichkeit solcher verteilter Systeme kann darin bestehen, zu berechnende Aufgaben in Teilaufgaben zu zerlegen und diese Teilaufgaben von den einzelnen Komponenten eines verteilten Systems lösen zu lassen. Solch eine Teilung und Verteilung von Aufgaben macht dabei natürlich erst ab einer bestimmten Größe und Komplexität eines gegebenen Problems Sinn.

Die Zusammenarbeit der einzelnen Rechner bzw. CPUs in einem verteilten System muß über spezielle Software gesteuert werden. Solche Software soll zum einen die verteilte Berechnung von Problemen überhaupt erst ermöglichen und zum anderen soll es dem Anwender möglichst einfach gemacht werden, seine Probleme verteilt berechnen zu lassen. D.h. die Internas der Verteilung der Teilprobleme auf die einzelnen Rechner bzw. CPUs, dies betrifft z.B. den eigentlichen Transfer der Teilaufgaben, oder den Austausch von Daten zwischen diesen, sollen vor dem Anwender so weit wie möglich verborgen werden.

Hinsichtlich der Schwierigkeit und des Aufwands, solche Steuerungssoftware für verteilte Systeme zu erstellen, muß unterschieden werden zwischen Systemen in denen einzelne Rechner zusammenarbeiten und Systemen in denen mehrere CPUs in einem Rechner zusammenarbeiten (siehe Abbildung 1.1). Einfacher zu entwickeln ist Steuerungssoftware für letztgenannte

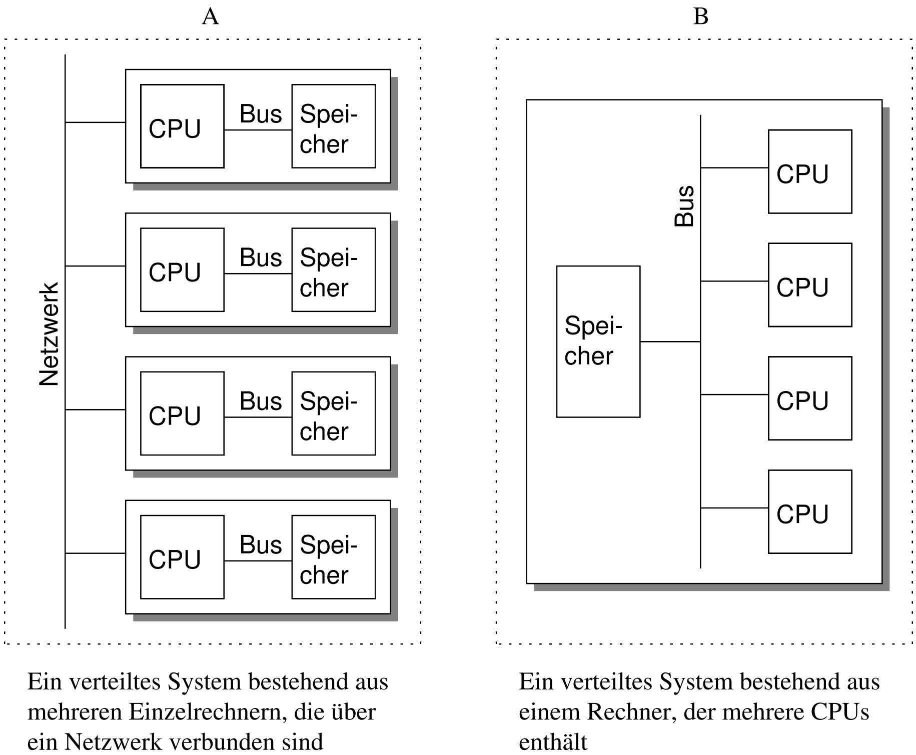

Abbildung 1.1: Beispiele für Verteilte Systeme

Systeme. Jedoch ist die Hardware für solche Systeme bedeutend aufwendiger und kostspieliger als für erstgenannte Systeme. Rechnernetzwerke dieser Art, d.h. verteilte Systeme, die aus Einzelcomputern bestehen, welche über ein Netzwerk miteinander verbundenen sind, sind heutzutage allerorten anzutreffen. Dabei muß hinzugefügt werden, daß eine sinnvolle verteilte Ausführung von Programmen nur dann möglich ist, wenn das dabei zum Einsatz kommende Netzwerk gewisse Mindestanforderungen an die Übertragungsgeschwindigkeit und Bandbreite der zwischen den Knoten auszutauschenden Nachrichten erfüllt. Mit dem DSM-Thread-System wird eine Entwicklungsumgebung für verteilte Programmierung in ebensolchen verteilten Systemen (Abbildung 1.1 A) realisiert.

1.2 Distributed Shared Memory

Ein Konzept zur Entwicklung obengenannter Steuerungssoftware für verteilte Systeme, in denen es keinen physisch gemeinsam genutzten Speicher gibt (Abbildung 1.1 A), stellt die softwaretechnische Realisierung eines virtuellen, verteilten, gemeinsam genutzten Speichers dar. Anstatt davon auszugehen, daß Programme innerhalb derartiger Systeme auf den verschiedenen Knoten (mit Knoten wird hier wie üblich ein einzelner Rechner innerhalb des verteilten Systems bezeichnet) grundsätzlich in komplett verschiedenen Adreßräumen ablaufen, entwarfen Li/Hudak [6, 7] die Methode, die Adreßräume der einzelnen Knoten zu einem einzigen, virtuellen, gemeinsamen Adreßraum - dem DSM [8, 9, 10] - zusammenzufassen (siehe Abbildung 1.2).

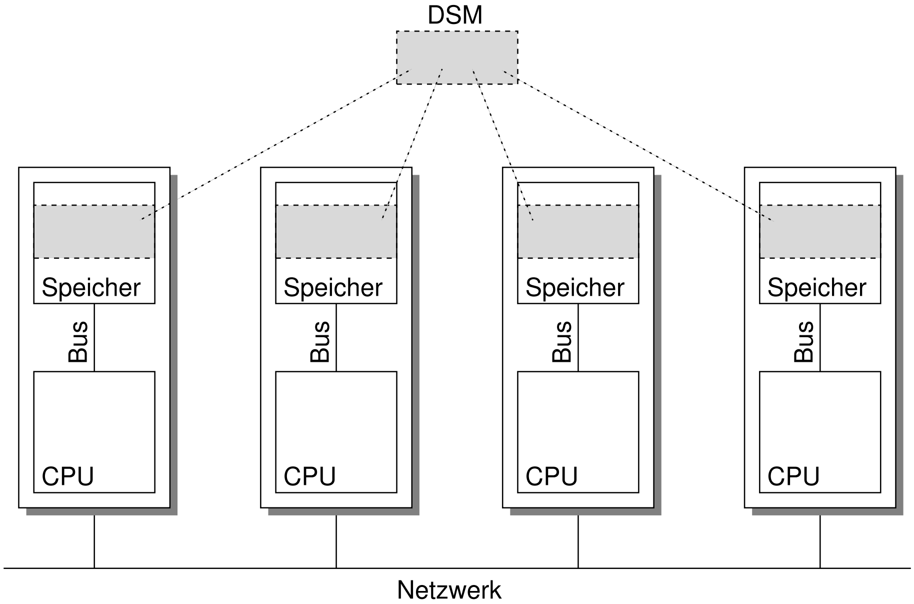

Abbildung 1.2: Virtueller, verteilter, gemeinsam genutzter Speicher

Der verteilte virtuelle Speicher ist dabei wie „normaler“ virtueller Speicher seitenbasiert und ist diesem auch in weiterer Hinsicht nicht unähnlich. Jede einzelne Seite des verteilten Speichers ist zu jedem Zeitpunkt virtuell auf genau einem Knoten vorhanden. D.h. von jeder verteilten Speicherseite gibt es zu jedem Zeitpunkt auf genau einem Knoten eine gültige Kopie (siehe Abbildung 1.3). Erfolgt auf einem Knoten ein Zugriff auf eine verteilte Speicherseite, von der sich keine gültige Kopie auf diesem Knoten befindet, so löst dies einen Hardware-Seitenzugriffsfehler aus. Aufgrund dieses Zugriffsfehlers wird dann ein Mechanismus ausgelöst, in dessen Verlauf die betreffende Seite von dem Knoten geholt wird, der gerade eine gültige Kopie dieser Seite besitzt und danach der Zugriff auf ersterem Knoten wiederholt. Der Ablauf ist also dem innerhalb des „normalen“ virtuellen Speichers ganz ähnlich, nur daß bei diesem die betreffende Speicherseite normalerweise von einer Festplatte geholt wird und nicht wie hier über ein Netzwerk von einem anderen Rechner [5].

Hinsichtlich der Anzahl von Knoten, auf denen es gleichzeitig eine gültige Kopie einer bestimmten DSM-Seite geben darf, kann es je nach angewende-

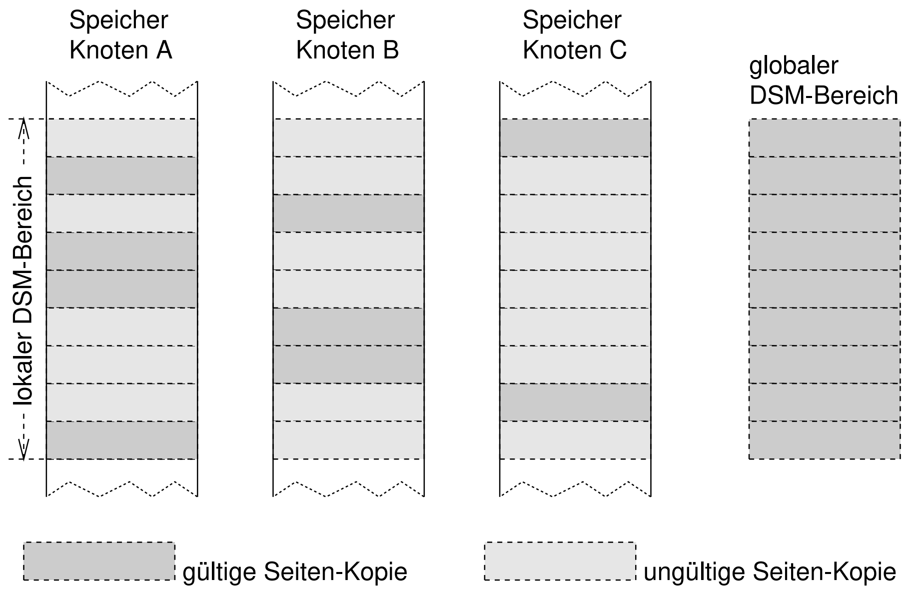

Abbildung 1.3: Verteilung des DSM

tem Seitenzugriffsprotokoll unterschiedliche Anforderungen geben Die oben genannte Bedingung gilt, wenn zu jedem Zeitpunkt nur auf einem einzigen Knoten auf eine bestimmte Speicherseite lesend oder schreibend zugegriffen werden darf (Zugriffsprotokoll „SingleReaderSingleWriter"). Darf hingegen von mehreren Knoten gleichzeitig auf eine bestimmte Speicherseite lesend oder schreibend zugegriffen werden, so muß diese Seite auch auf den entsprechenden Knoten gleichzeitig gültig sein (Zugriffsprotokolle „MultiReaderSingleWriter" und „MultiReaderMultiWriter"). D.h. es gibt dann mehrere gültige Lese- oder Schreibkopien der betreffenden Seite.

Neben DSM-Systemen in denen der gesamte Adreßraum eines jeden Knotens als DSM genutzt wird (z.B. IVY [6, 7] oder Mirage [11]), gibt es auch Systeme in denen der DSM nur einen Teil des Adreßraumes einnimmt (wie in Abbildung 1.2 zu sehen), weil nur ein bestimmter Teil der Variablen gemeinsam genutzt werden soll (z.B. Munin [12, 13] oder Midway [14, 15]). Dies geschieht aus Effektivitätsgründen, und zwar in dem Fall hauptsächlich, um die Menge der zu transferierenden Daten einzuschränken und damit den notwendigen Netzwerkverkehr zu reduzieren. Der gemeinsame Adreßraum kann jedoch noch stärker strukturiert werden, indem nicht einzelne Variablen, sondern Objekte (wie man sie aus der objektorientierten Programmierung kennt) gemeinsam genutzt werden. Man spricht dann, im Gegensatz zu seitenbasierenden DSM (alle bisher genannten Beispiele), von objektbasierenden DSM (z.B. Linda [16, 17] oder Orca [18, 19]).

1.3 DSM-Threads

DSM-Threads setzen die in den vorhergehenden Abschnitten gemachten Aussagen um. Auf der Grundlage der Pthreads bilden sie ein Laufzeitsystem, in welchem der Anwender Threads verteilt ausführen kann und realisieren dies unter anderem mit Hilfe eines seitenbasierenden, virtuellen, verteilten, gemeinsam genutzten Speichers. Dabei wird nur ein Teil des Adreßraumes der einzelnen Knoten als DSM genutzt.

Die grundlegende Funktionsweise der DSM-Threads soll hier nur kurz vereinfacht dargestellt werden. Auf jedem Knoten innerhalb des verteilten Systems, auf dem DSM-Threads laufen, wird ein Prozeß gestartet, welcher ein Pthread-System [20], DSM-Systemthreads und Anwenderthreads enthält. Alle Seiten des gemeinsamen, verteilten Speichers sind auf den einzelnen Knoten entweder validiert oder invalidiert. Als validiert auf einem Knoten wird eine DSM-Seite bezeichnet, deren Besitzer dieser Knoten momentan ist, was bedeutet, daß auf ihm auf sie zugegriffen werden kann. Ist hingegen ein Knoten nicht der momentane Besitzer einer Speicherseite, so wird diese Seite als auf diesem Knoten invalidiert bezeichnet und aus einem Zugriff von diesem Knoten auf diese Seite resultiert ein Speicherzugriffsfehler (nämlich das Signal `SIGSEGV`). Die Validierung bzw. Invalidierung wird über den UNIX-Systemruf `mprotect` mittels der Attribute `PROT_READ`, `PROT_WRITE`, `PROT_NONE` vorgenommen.

Zur Startzeit sind alle DSM-Seiten auf dem Startknoten validiert und auf allen anderen, d.h. den entfernten Knoten, invalidiert. Tritt nun auf einem Knoten, beispielsweise innerhalb eines Anwenderthreads, weil der Anwender auf verteilte Daten zugegriffen hat, ein Zugriffsfehler auf eine DSM-Seite auf, so übergibt der Signalhandler dem Seitenmanager (ein DSM-Systemthread mit der zweithöchsten Priorität innerhalb des Prozesses) dieses Knotens eine Anfrage für diese Seite und legt sich schlafen. Der Seitenmanager beauftragt sodann einen Dienstthread (innerhalb der DSM-Systemthreads gibt es einen Pool von Dienstthreads, die unterschiedliche Aufgaben übernehmen) damit, die benötigten Zugriffsrechte für diese Seite zu beschaffen. Dieser Dienstthread erfüllt diese Aufgabe beispielsweise, indem er an einen anderen Knoten, nämlich an den wahrscheinlichen Besitzer der entsprechenden Seite, eine Seitenanfrage sendet. Empfängt dann der Kommunikationsmanager (ein DSM-Systemthread mit der höchsten Priorität, der einen Kommunikationskanal offen hält und auf eintreffende Nachrichten wartet) eines Knotens eine derartige Seitenanfrage und dieser Knoten ist tatsächlich der aktuelle Besitzer der entsprechenden Seite, so beauftragt er einen Dienstthread damit, diese Seite zu invalidieren und an den anfragenden Knoten zu schicken. Anderenfalls, d.h. wenn dieser Knoten nicht der aktuelle Besitzer dieser Seite ist, schickt der Kommunikationsmanager die Anfrage an den scheinbaren Besitzer weiter. Empfängt der Kommunikationsmanager des ursprünglich anfragenden Knotens dann die Seite, so aktiviert er einen Dienstthread, der die Aufgabe hat, die Seite zu validieren und dann den Anwenderthread wieder aufzuwecken. Vom Anwenderthread kann der Zugriff sodann mit Erfolg wiederholt werden.

Kapitel 2

Die Verwaltung von DSM

Die Verteilung von Speicher auf mehrere Knoten und dessen gemeinsame Nutzung durch mehrere Knoten stellt an die Verwaltung dieses Speichers spezielle Anforderungen. Eine besondere Herausforderung stellt dabei die Aufrechterhaltung der Konsistenz des Speichers auf allen Knoten dar. Diese Konsistenz muß dabei zum einen auf der Ebene der Speicherseiten sichergestellt werden. Wie in Abschnitt 1.2 angedeutet wurde, muß garantiert werden, daß nur genau die erlaubte Anzahl (in Abhängigkeit von dem verwendeten Seitenzugriffsprotokoll) von Knoten gleichzeitig auf eine DSM-Seite lesend oder schreibend zugreift. Zu den Details der Realisierung der Speicherkonsistenz auf Seitenebene sei auf die angegebene Literatur verwiesen ([1, 2, 5, 8, 21]), da dies die eigentliche Speicherverwaltung nur indirekt betrifft. Hier von Interesse ist die Konsistenz des verteilten Speichers auf der Ebene innerhalb der DSM-Seiten.

Auf dieser Ebene geht es um die einzelnen DSM-Objekte. Mit Objekt wird innerhalb dieser Arbeit ein „Stück Speicher" bezeichnet. Entsprechend bezeichnet DSM-Objekt ein Stück aus dem DSM. Erfolgt auf einem Knoten ein Zugriff auf solch ein verteiltes, gemeinsam genutztes Objekt und die Seite, auf der sich dieses Objekt befindet, ist auf diesem Knoten nicht validiert, so wird, da es sich um seitenbasierenden DSM handelt, diese Seite vollständig, nach dem in Abschnitt 1.3 beschriebenen Schema, transferiert. Daraus ergibt sich die Notwendigkeit, daß sich die DSM-Objekte erstens auf allen Knoten auf der jeweils gleichen DSM-Seite befinden müssen, und daß sie sich zweitens jeweils an der gleichen Stelle innerhalb der entsprechenden DSM-Seite befinden müssen. DSM-Objekt x muß also z.B. auf allen Knoten auf der DSM-Seite i stehen und muß darüber hinaus jeweils den gleichen Abstand zur Anfangsadresse von DSM-Seite i haben. Weiterhin muß sichergestellt werden, daß die Speicherseiten, die für den DSM benutzt werden, komplett für die DSM-Objekte reserviert sind. D.h. auf den DSM-Seiten dürfen sich keine anderen, „normalen" Objekte befinden, da diese, ebenfalls aufgrund der Seitenbasierung des DSM, im Falle eines Transfers der entsprechenden DSM-Seite, ebenfalls transferiert und auf dem Empfängerknoten überschrieben werden würden.

Die Verwaltung des DSM wird auf allen Ebenen wesentlich vereinfacht, wenn erreicht wird, daß alle DSM-Seiten in einem zusammenhängenden Bereich im Speicher stehen, der im Idealfall auf allen Knoten dieselbe Anfangsadresse hat.

Die Forderung nach einem zusammenhängenden Speicherbereich für die dynamischen DSM-Objekte ist jedoch noch für einen weiteren Aspekt von großer Bedeutung. Dürften die DSM-Objekte und die „normalen" Objekte in beliebiger Reihenfolge im Speicher stehen, so würde daraus unter Umständen eine enorme Verschwendung von Speicher resultieren. Aufgrund der Seitenbasierung des DSM müßte jeder Bereich im Speicher, an dem DSM-Objekte stehen, auf Seitengrenze ausgerichtet werden. Bei alternierender Erzeugung von DSM-Objekten und „normalen" Objekten, müßte demzufolge für jedes einzelne DSM-Objekt mindestens eine komplette Speicherseite reserviert werden. Es ist leicht abzusehen zu welch unnötig umfangreicher Speicherbelegung dies im allgemeinen Fall führen würde.

Die Einhaltung dieser Bedingungen ist eine der Hauptschwierigkeiten bei der Verwaltung von verteiltem, gemeinsamem Speicher. Dabei hängt der Schwierigkeitsgrad stark davon ab, welche Art Objekte als DSM-Objekte zugelassen werden und wo sich diese Objekte im Speicher eines Prozesses befinden dürfen.

2.1 Statischer DSM

Eine Möglichkeit, die eben beschriebenen Bedingungen auf recht einfache Weise erfüllen zu können, ist es, sämtliche DSM-Objekte an jeweils derselben Stelle im statischen Teil des Speichers der einzelnen Prozesse auf den unterschiedlichen Knoten abzulegen. D.h. in dem Fall werden ausschließlich Objekte mit statischer Lebensdauer als DSM-Objekte zugelassen.

Statische Objekte sind jene Objekte, die während der gesamten Laufzeit vorhanden sind. In der Programmiersprache C sind dies die Objekte in Variablen, welche extern (d.h. außerhalb von Funktionsblöcken) definiert werden sowie die Objekte in Variablen, welche intern (d.h. innerhalb von Funktionsblöcken) mit dem Schlüsselwort `static` definiert werden. Als DSM-Objekte sind dabei jedoch nur erstere geeignet, da nur diese Objekte für alle Funktionen im Anwenderprogramm sichtbar sind. Statische Objekte werden vom Laufzeitsystem vor der eigentlichen Programmausführung auf jedem Knoten im Datensegment des virtuellen Speichers des jeweiligen Prozesses erzeugt, sofern sie vom Anwender initialisiert werden und anderenfalls, d.h. wenn sie vom Anwender nicht initialisiert werden, im BSS-Segment („Block Started by Symbol“) erzeugt, wobei sie mit binären Nullen initialisiert werden. Die Anordnung der Objekte im Datensegment wird von der Reihenfolge ihrer Initialisierung durch den Anwender bestimmt, ist auf allen Knoten dieselbe und bleibt während der gesamten Laufzeit unverändert. Sofern also alle DSM-Objekte unmittelbar nacheinander extern initialisiert werden, stehen sie in einem zusammenhängenden Bereich an jeweils derselben Stelle im Speicher eines jeden Knotens des verteilten Systems (siehe Abbildung 2.1). Damit ist ein Großteil der erforderlichen Bedingungen erfüllt.

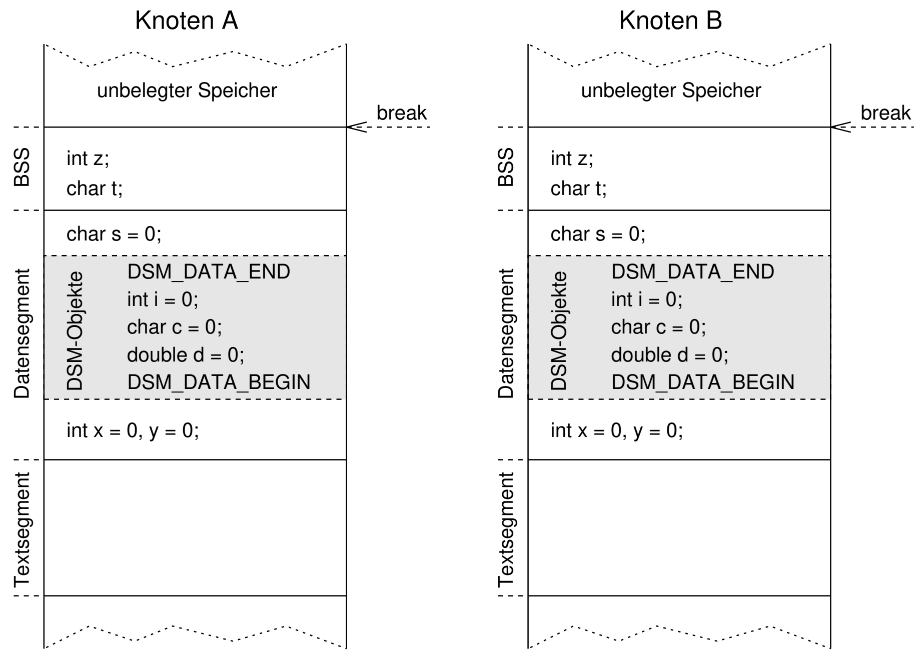

Abbildung 2.1: Speicherlayout der Prozesse auf allen Knoten zur Startzeit

In der ursprünglichen Implementierung der DSM-Threads wird dieser Ansatz verfolgt. Dem Anwender wurde auferlegt, sämtliche DSM-Objekte extern zwischen zwei Makros (`DSM_DATA_BEGIN` und `DSM_DATA_END`) zu initialisieren. Diese Makros haben u.a. die Aufgabe den Speicherbereich, in dem sich dann alle DSM-Objekte befinden, auf Seitengrenze auszurichten. Mit dieser Ausrichtung kann garantiert werden, daß auf den betreffenden Speicherseiten ausschließlich DSM-Objekte stehen.

2.2 Dynamischer DSM

Sollen jedoch auch dynamische DSM-Objekte zugelassen werden, d.h. wenn es möglich sein soll, Speicher für DSM-Objekte auch zur Laufzeit bereitzustellen und wieder freizugeben, so sind obengenannte Bedingungen ungleich schwerer zu erfüllen.

Anfragen nach zusätzlichem Speicher durch einen Prozeß werden vom Betriebssystem (UNIX) aus dem unbelegten Speicherbereich oberhalb des Datensegments bzw. des BSS-Segments bedient (siehe Abbildung 2.1). Dazu kann der Prozeß die Grenze („break“) zwischen dem Datensegment bzw. dem BSS-Segment und dem unbelegten Speicher nach oben verschieben. Der so entstehende Speicherbereich wird Heap genannt. Im Heap werden sämtliche Objekte abgelegt, die erst während der Laufzeit erzeugt werden. Dort stehen also nicht nur die dynamischen DSM-Objekte, sondern auch die „normalen“ dynamischen Objekte. In C sind dies gewöhnlich die Objekte, für die `malloc` und `free` aufgerufen wird. Die Reihenfolge, in der die dynamischen Objekte im Heap erzeugt und wieder freigegeben werden können, kann dem Anwender nicht vorgeschrieben werden, da dies eine nicht hinnehmbare Einschränkung für den Anwender wäre. Weiterhin kann nicht vorgeschrieben werden, welche dynamischen Objekte auf welchen Knoten erzeugt werden, d.h. auf den einzelnen Knoten kann sich eine unterschiedliche Anzahl von unterschiedlich großen Objekten im Heap befinden. Im Gegensatz zum Datensegment kann der Heap also auf den unterschiedlichen Knoten sehr wohl unterschiedlich strukturiert sein und er kann sich darüberhinaus in seiner Struktur während der Laufzeit ständig ändern (siehe Abbildung 2.2).

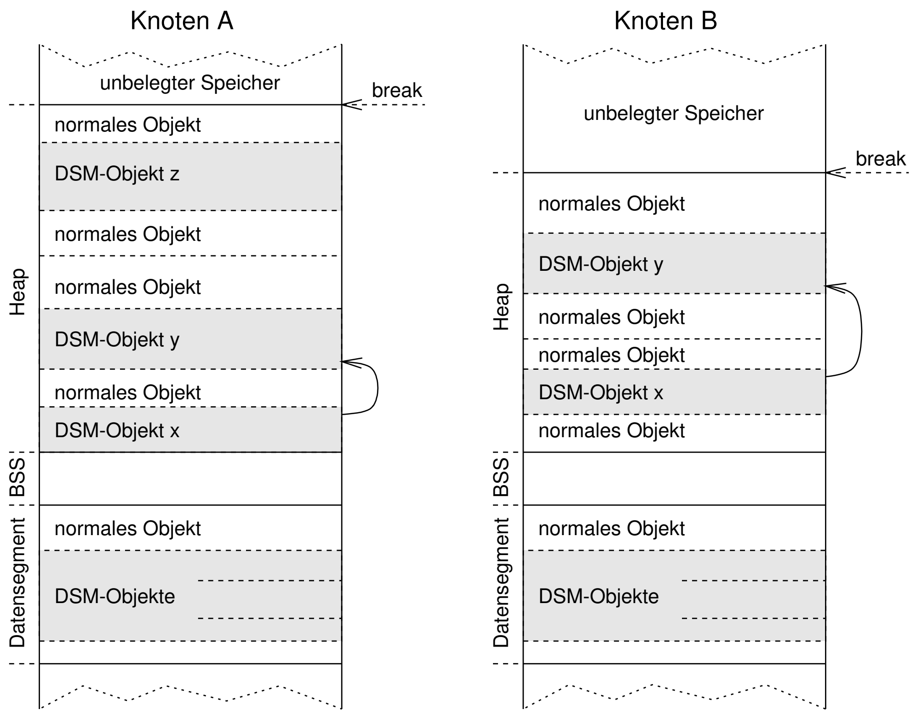

Abbildung 2.2: Speicherlayout auf unterschiedlichen Knoten zur Laufzeit

Für DSM-Objekte, die im Heap erzeugt werden, ist also keine der genannten Bedingungen an das Layout des DSM erfüllt, sofern nicht zusätzliche Maßnahmen zu deren Erfüllung ergriffen werden. Aus der dynamischen Erzeugung von DSM-Objekten ergeben sich jedoch noch weitere Probleme.

Im Gegensatz zu den statischen Objekten entstehen die dynamischen Objekte erst während der Laufzeit. Sie sind also nicht automatisch auf allen Knoten vorhanden. Dies ist aber notwendig, da sie gemeinsam, d.h. von allen Knoten, genutzt werden sollen. Das bedeutet, daß, wenn auf einem Knoten ein neues DSM-Objekt erzeugt wird, dies allen anderen Knoten bekannt gemacht werden muß und diese das Objekt ebenfalls erzeugen müssen. Dies betrifft beispielsweise in Abbildung 2.2 das Objekt z auf Knoten A. Gleiches gilt für DSM-Objekte, die während der Laufzeit auf einem Knoten wieder freigegeben werden.

2.2.1 Zeiger im dynamischen DSM

Die einzelnen dynamischen DSM-Objekte können auf den unterschiedlichen Knoten an unterschiedlichen Stellen im Heap liegen. Dementsprechend haben sie unterschiedliche Adressen. Daraus ergeben sich weitreichende Konsequenzen für die Verwendung von Zeigern. Auf Zeiger kann nicht verzichtet werden, da z.B. die Objekte aus dem Heap über Zeiger an den Anwender übergeben werden. Selbst wenn sichergestellt wird, daß alle DSM-Objekte auf allen Knoten auf der jeweils gleichen DSM-Seite liegen und jeweils den gleichen Abstand zur Anfangsadresse der Seite haben, garantiert dies nicht, daß diese Objekte tatsächlich die gleichen Adressen auf allen Knoten haben, da die einzelnen DSM-Seiten auf den unterschiedlichen Knoten unterschiedliche Anfangsadressen haben können. Es stellt sich also das Problem, daß Zeiger auf DSM-Objekte auf den einzelnen Knoten auf unterschiedliche Adressen zeigen müssen, um auf die richtigen Objekte zu verweisen. So ist in Abbildung 2.2 ein Zeiger zu sehen, der in Objekt x steht und auf Objekt y verweist. Objekt y hat jedoch auf den Knoten A und B eine unterschiedliche Adresse. Wird nun die Seite, auf der sich Objekt x befindet, von Knoten A zu Knoten B transferiert, ohne dabei die Tatsache zu behandeln, daß Objekt y auf den beiden Knoten verschiedene Adressen hat, so verweist der Zeiger in Objekt x nach dem Transfer auf Knoten B auf dieselbe Adresse wie er es auf Knoten A getan hat. Er zeigt auf Knoten B also nicht auf Objekt y. Wie ebenfalls in Abbildung 2.2 zu sehen, kommt hinzu, daß auch die Adresse des Zeigers, in dem Fall Objekt x, auf den verschiedenen Knoten unterschiedlich sein kann. Den eventuell unterschiedlichen Adressen der DSM-Objekte muß also bei einem Transfer von DSM-Seiten, die Objekte mit Zeigern enthalten, entsprechend Rechnung getragen werden.

Bei der Threadmigration tritt dieses Problem in sehr ähnlicher Weise auf [22, 23]. Es kann auf sehr unterschiedliche Möglichkeiten gelöst werden. Es gibt z.B. Systeme, in denen zugelassen wird, daß Zeiger nach einem Transfer undefiniert sind [24]. Andere Systeme benutzen Zeiger, welche zusätzliche

Angaben über den Ort der Daten beinhalten, auf die sie verweisen [25]. Die Zeiger bei einem Transfer zu aktualisieren, ist eine weitere Lösungsmöglichkeit [26, 27, 28]. Ein gänzlich verschiedener Ansatz besteht darin, für gewisse Daten eines jeden Threads einen Bereich an der jeweils gleichen Stelle im virtuellen Adreßraum auf jedem Knoten zu reservieren [23, 29, 30]. Einige Aspekte dieser verschiedenen Herangehensweisen an das Problem der unterschiedlichen Objektadressen (und zwar vornehmlich aus den beiden zuletzt genannten) werden in dieser Arbeit als Anregung genutzt.

2.2.2 Zugriff auf Objekte im dynamischen DSM

Dem Anwender werden Abschnitte aus dem DSM in Form von Zeigern auf diese Speicherabschnitte übergeben. Ein Objekt als DSM-Objekt anzulegen, macht nur Sinn, wenn alle Knoten auf dieses Objekt zugreifen können. Der Zugriff auf ein dynamisches DSM-Objekt ist nur dann auf allen Knoten möglich, wenn eine Variable, aus der auf dieses Objekt verwiesen wird, auf allen Knoten zur Verfügung steht. Darum sollten diese Variablen DSM-Variablen sein, also entweder im statischen oder im dynamischen DSM selbst stehen.

Letztlich bleibt es zwar dem Anwender überlassen, ob er den DSM sinnvoll einsetzt, jedoch kann gegenteiliges durchaus auch zu Schwierigkeiten in der Verwaltung führen. Dies ist der Fall, wenn er z.B. einer Variable, die sich nicht im DSM befindet, den Rückgabewert der Speicherverwaltungsfunktion zuweist, die einen Zeiger auf einen neuen DSM-Abschnitt übergibt. Dies steht in Zusammenhang mit den im letzten Abschnitt genannten Problemen, wobei der Umfang der entstehenden Schwierigkeiten davon abhängt, auf welche Art und Weise diese Probleme gelöst werden. In den Kapiteln 4 und 5 wird darauf näher eingegangen.

Kapitel 3

Semi-Dynamischer DSM

Es soll nun ein Modell für den Aufbau des DSM-Heap (mit DSM-Heap werden hier alle Bereiche im Heap bezeichnet, welche DSM-Objekte enthalten) dargestellt werden, mit dem sich die im letzten Kapitel aufgezeigten Probleme auf einfache Weise lösen lassen, das dafür jedoch Einschränkungen erfordert, die dem Anwender im allgemeinen Fall nicht abverlangt werden können. Der DSM ist innerhalb dieses Lösungsansatzes nicht wirklich dynamisch und wird hier darum semi-dynamisch genannt.

3.1 Der Aufbau des DSM-Heap

Eine Möglichkeit zu erreichen, daß sich alle dynamischen DSM-Objekte in einem zusammenhängenden Speicherbereich befinden, der auf allen Knoten an der gleichen Adresse beginnt, besteht darin, bereits während der Initialisierungsphase des DSM-Systems auf allen Knoten für die dynamischen DSM-Objekte eine gewisse Menge Speicher vom Betriebssystem anzufordern. D.h. bevor im Heap irgendwelche anderen Objekte erzeugt werden, wird dort ein Bereich, im folgenden DSM-Heapblock genannt, reserviert, der dann alle dynamischen DSM-Objekte aufnehmen soll (siehe Abbildung 3.1). Denn das erste Objekt, das im Heap erzeugt wird, hat auf allen Knoten die gleiche Anfangsadresse, da es überall direkt an der Grenze zum Datensegment bzw. zum BSS-Segment liegt. Somit hat der DSM-Heapblock auf allen Knoten die gleiche Anfangsadresse.

Der eigentliche Beginn des DSM-Heapblocks innerhalb des Speicherblocks muß zuvor aufgrund der Seitenbasierung des DSM auf Seitengrenze ausgerichtet werden. Aus demselben Grund muß dessen Größe ein Vielfaches der Seitengröße sein.

Analog zum Verfahren für den statischen DSM werden alle Seiten im DSM-Heap auf dem Startknoten validiert und auf allen anderen Knoten invalidiert.

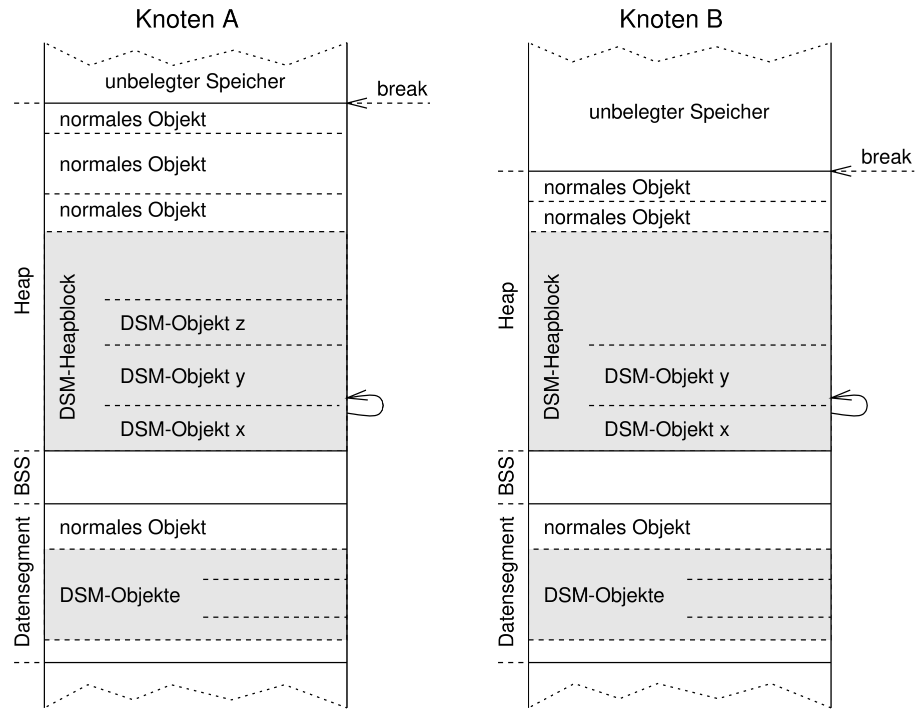

Abbildung 3.1: Speicherlayout mit einem DSM-Heapblock

Möchte dann der Anwender zur Laufzeit ein DSM-Objekt erzeugen, so bekommt er dafür ausschließlich Speicher aus dem DSM-Heapblock geliefert. Gibt der Anwender ein DSM-Objekt wieder frei, so wird dieser Speicher wieder in den DSM-Heapblock eingefügt und kann dem Anwender im folgenden erneut zur Verfügung gestellt werden.

Sofern dann sichergestellt wird, daß alle DSM-Objekte auf allen Knoten in der gleichen Reihenfolge erzeugt werden, haben sie auf allen Knoten die gleichen Adressen. Damit ist garantiert, daß alle DSM-Objekte auf allen Knoten auf der jeweils gleichen DSM-Seite stehen und jeweils denselben Abstand zur Anfangsadresse der entsprechenden Seite haben. Weiterhin stellt sich innerhalb dieses Lösungansatzes das Problem unterschiedlicher Adressen bzgl. der zu transferierenden Zeiger nicht. Auch ist sichergestellt, daß sich auf den DSM-Seiten ausschließlich DSM-Objekte befinden.

3.2 Verwaltung des Speichers im DSM-Heapblock

Der Speicher innerhalb des DSM-Heap muß geeignet verwaltet werden. Dies wird durch die Tatsache, daß die DSM-Objekte auf allen Knoten die gleichen Adressen haben und in einem zusammenhängenden Speicherbereich stehen, erheblich vereinfacht. An die Verwaltungsmethode werden dabei im wesentlichen keine besonderen Anforderungen oder Bedingungen gestellt. Trotzdem sind bei deren Auswahl die speziellen Gegebenheiten des seitenbasierenden DSM zu bedenken, um negative Auswirkungen auf die Performance des DSM-Systems zu weit wie möglich zu minimieren.

Die zur Speicherverwaltung notwendigen Datenstrukturen sollten selbst so wenig Speicherplatz wie möglich beanspruchen und sollten sich weiterhin nicht über unnötig viele Speicherseiten verteilen. Beide Anforderungen ergeben sich daraus, daß die Struktur des Speichers im DSM-Heap auf allen Knoten die gleiche sein muß (siehe Abschnitt 3.4). Die Daten, die diese Struktur beschreiben, müssen also auf allen Knoten zur Verfügung stehen, denn wenn diese Daten auf einem Knoten ergänzt oder verändert werden, weil ein dynamisches DSM-Objekt erzeugt oder wieder freigeben wird, so muß diese Veränderung auf allen Knoten vorgenommen werden. Weiterhin müssen diese Daten dynamisch erweiterbar sein. Es muß sich also bei den Daten zur Verwaltung des DSM-Heap selbst ebenfalls um dynamische DSM-Objekte handeln. Damit aus dem notwendigen Transfer der Verwaltungsdaten nur ein möglichst geringer Transfer von DSM-Seiten resultiert, werden die beiden oben genannten Anforderungen an diese Daten gestellt. Denn der Transfer der DSM-Seiten kostet in einem verteilten System, in dem die Knoten über ein Netzwerk verbunden sind (siehe Abschnitt 1.1), sehr viel an Performance. Ein Hauptziel, wenn es um die Performanceverbesserung in einem solchen System geht, ist es demzufolge, den notwendigen Seitentransfer so gering wie möglich zu halten.

Bei einer dynamischen Speicherverwaltung wie sie hier besprochen wird geht es im Wesentlichen darum, im Falle einer entsprechenden Anfrage des Anwenders, einen geeigneten freien Speicherabschnitt innerhalb des verwalteten Speicherbereichs zu finden bzw. vom System anzufordern und dem Anwender zur Verfügung zu stellen sowie einen vom Anwender wieder zurückgegeben Speicherabschitt als freien Abschnitt in die Verwaltung aufzunehmen oder an das System zurückzugeben. Um dies möglichst effizient tun können, werden dabei innerhalb vieler Methoden zur Speicherverwaltung mehrere Listen verwendet. Dieser Weg soll hier aus oben genannten Gründen nicht beschritten werden. Für die Verwaltung des DSM-Heap wird eine Methode nach Kernighan/Ritchie [31] gewählt, bei der nur eine vorwärtsverkettete Liste verwendet wird.

3.3 Die Speicherverwaltung von Kernighan/Ritchie

In dieser Liste (im folgenden mit Freelist bezeichnet) stehen alle freien Speicherabschnitte. Die Elemente der Freelist enthalten den freien Speicherabschnitt selbst sowie einen Kopf, der die Größe dieses Abschnitts und einen

Zeiger auf den nächsten freien Abschnitt enthält (siehe Abbildung 3.2). Dabei ist der jeweils nächste Abschnitt immer derjenige, welcher die nächst höhere Adresse im Speicher hat, mit einer Ausnahme, und zwar wird von dem letzten Element zurück auf das erste verwiesen. Der Zugriff auf die Freelist erfolgt über einen zusätzlichen Zeiger (`start_search_p`), der immer auf jene Stelle in der Freelist verweist, an welcher zuletzt ein Element bearbeitet wurde. Die verwalteten Speicherabschnitte liegen ausnahmslos im Heap. Sie müssen sich dabei nicht in einem zusammenhängenden Bereich befinden, da auch andere Programmteile während Laufzeit Speicher aus dem Heap anfordern können, also andere Objekte zwischen den verwalteten Abschnitten liegen können.

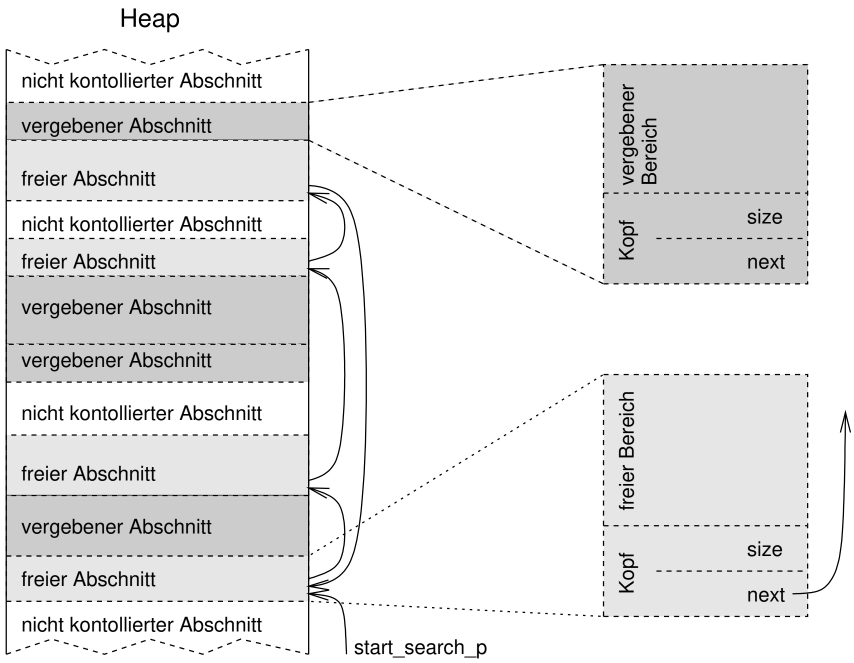

Abbildung 3.2: Die Freelist in der Speicherverwaltung von Kernighan/Ritchie

Bei einer Speicheranforderung durch den Anwender (entspricht `malloc`) wird die Freelist nach einem freien Abschnitt mit ausreichender Größe durchsucht, d.h. es werden die Größenangaben in den Köpfen der Freelist-Elemente mit der Größe des angeforderten Speicherabschnitts verglichen. Die Suche wird gestoppt, sobald ein Abschnitt gefunden wird, der groß genug ist, um die Anwenderforderung bedienen zu können („first-fit“-Methode). Dann wird das entsprechende Element entweder aus der Freelist entfernt, nämlich wenn der gefundene Abschnitt genau die richtige Größe hat, oder es wird, nämlich wenn dieser Abschnitt größer als benötigt ist, der hintere Teil dieses Abschnitts an den Anwender vergeben und lediglich die Angabe über die

Größe des freien Abschnitts im Kopf des Freelist-Elements entsprechend der verbleibenden Größe neu berechnet.

Wurde jedoch nach einem kompletten Durchlauf der Freelist kein ausreichend großer freier Speicherabschnitt gefunden, so wird über den UNIX-Systemruf `sbrk(n)` („break"-Verschiebung um `n` Bytes, siehe Abschnitt 2.2) ein neuer Speicherabschnitt vom System angefordert, dieser Abschnitt dann am Ende der Freelist eingefügt und schließlich die Anwenderforderung aus diesem Abschnitt bedient.

Gibt der Anwender einen Speicherabschnitt wieder frei (entspricht `free`) wird wiederum die Freelist durchsucht, diesmal mit dem Ziel jene Stelle in der Freelist zu finden, an welcher der Abschnitt eingefügt werden muß, d.h. es werden die Speicheradressen der bereits in der Freelist befindlichen Abschnitte mit der des einzufügenden Abschnitts verglichen. Nachdem die richtige Stelle gefunden wurde, brauchen lediglich die „next"-Zeiger in den Köpfen der betroffenen Freelist-Elemente und des einzufügenden Elements (dessen „size"-Angabe ist während seiner Benutzung durch den Answender erhalten geblieben) korrigiert zu werden, um den freigegebenen Abschnitt in die Freelist einzufügen. Grenzen dabei Abschnitte direkt aneinander, so werden sie zu einem einzigen zusammengefaßt, wobei zusätzlich zu den „next"-Zeigern lediglich die „size"-Angaben der entsprechenden Elemente korrigiert werden müssen.

3.3.1 Veränderungen zur Verwendung im DSM-Heap

Neben den im nächsten Abschnitt aufgeführten Vorteilen bietet die Speicherverwaltungsmethode von Kernighan/Ritchie jenen, daß sie für ihre Anwendung im DSM-Heap nur minimal verändert werden muß.

Die wesentlichste Veränderung betrifft den Aufruf von `sbrk`. Da sämtliche dynamischen DSM-Objekte im dem bereits während der Initialisierungsphase allozierten Speicherblock (der DSM-Heapblock, siehe Abschnitt 3.1) erzeugt werden sollen, soll während der Laufzeit kein zusätzlicher Speicher via `sbrk` vom Betriebssystem angefordert werden. Darum wird der Aufruf von `sbrk` durch eine Funktion ersetzt, die einen Speicherabschnitt aus dem DSM-Heapblock liefert. Diese Funktion verschiebt die Grenze zwischen dem bereits vergebenen Bereich im DSM-Heapblock (`dsm_break`, siehe Abbildung 3.3) und dessen Obergrenze solange nach oben wie noch ausreichend Platz in ihm ist, um die jeweilige Anforderung zu bedienen zu können und schlägt anderenfalls fehl.

Eine weitere notwendige Veränderung bzw. Anpassung der Methode von Kernighan/Ritchie betrifft die Variablen `dsm_break` und `start_search_p`. Da diese Variablen auf allen Knoten zur Verfügung stehen müssen (siehe Abschnitt 3.4), muß es sich dabei um DSM-Objekte handeln. Und da diese Variablen weiterhin bereits zur Verfügung stehen müssen, wenn das erste

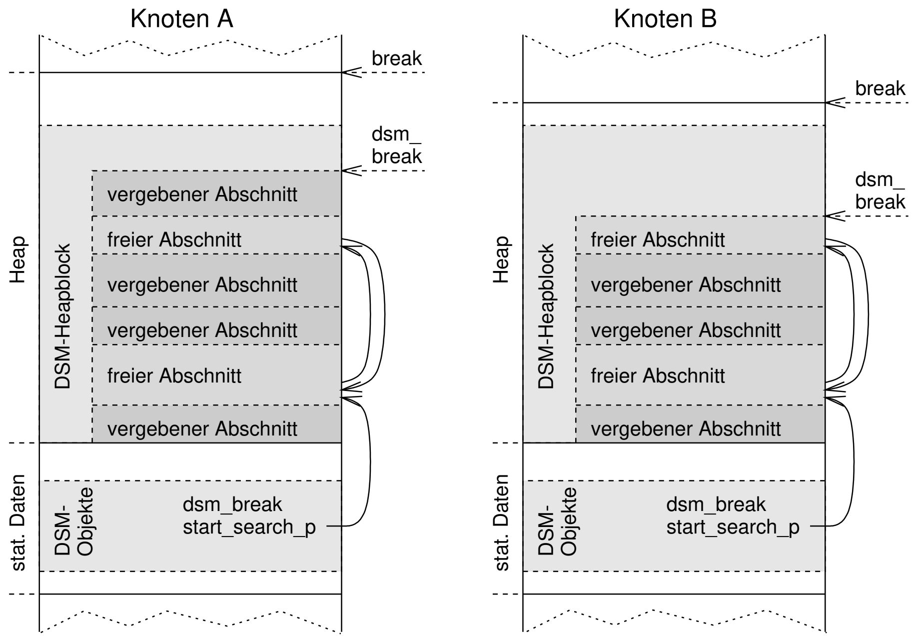

Abbildung 3.3: Die Kernighan/Ritchie-Methode im DSM-Heap

Objekt im DSM-Heap erzeugt werden soll, d.h. wenn die erste Operation mit der Freelist ausgeführt wird, können sie nicht selbst im DSM-Heap stehen. Darum werden `dsm_break` und `start_search_p` im statischen DSM-Bereich (siehe Abschnitt 2.1) angelegt.

3.3.2 Vorteile bei der Verwendung im DSM-Heap

Der Hauptvorteil der Kernighan/Ritchie-Methode besteht bei ihrer Verwendung im DSM-Heap in ihrer Einfachheit. Ihr daraus im allgemeinen Fall resultierender Nachteil, bei den Operationen mit der Freelist nicht sehr effizient vorzugehen, ist hier also von Vorteil. Denn in einem verteilten System läßt sich die Effizienz über ein Reduzieren des notwendigen Seitentransfers sehr viel wirkungsvoller steigern als über die Optimierung einzelner Rechenvorgänge.

Zwar müssen, wenn auf einem Knoten die Freelist durchsucht bzw. berarbeitet wird, alle Seiten zu diesem Knoten transferiert werden, auf denen sich jene Freelist-Elemente befinden, auf die dabei zugegriffen wird und deren Besitzer er nicht zufällig schon ist. Jedoch ist dies die mindestens notwendige bzw. kleinstmögliche Anzahl von Seiten, da die Anzahl der Elemente auf die dabei notwendigerweise zugegriffen werden muß aufgrund der größtmöglichen Einfachheit der Struktur der Freelist so gering wie möglich gehalten wird. In einem späteren Kapitel wird gezeigt wie dieser Seitentransfer weiter reduziert werden kann.

Die Anwendung der „first-fit“-Methode beim Durchsuchen der Freelist nach einem freien Speicherabschnitt kommt, z.B. im Gegensatz zur „best-fit“-Methode, ebenfalls dem Ziel zugute, den notwendigen Seitentransfer so gering wie möglich zu halten.

Auch zwei weitere Eigenschaften der Kernighan/Ritchie-Methode, die bei ihrer Anwendung im allgemeinen Fall eigentlich als Nachteil gewertet werden müssen, kommen hier nicht negativ zum Tragen. Dies betrifft zum einen die Tatsache, daß nicht mehr benötigter Speicher nie an das Betriebssystem zurückgegeben wird (wie es z.B. durch den Aufruf von `sbrk(-n)` möglich wäre). Dies ist innerhalb des DSM nicht von Nachteil, da dies nur möglich wäre, wenn der DSM-Heap verkleinert werden würde. Angesichts des hohen Aufwands der dazu allerdings innerhalb des DSM-Laufzeitsystems an zahlreichen Stellen notwendig wäre, soll hier bewußt auf diese Möglichkeit verzichtet werden. Und zum anderen betrifft dies die Tatsache, daß ausschließlich `sbrk` genutzt wird, um mehr Speicher vom Betriebssystem zu erhalten. Beispielsweise benutzen andere Speicherverwaltungsmethoden aus Effizienzgründen ab einer gewissen Größe eines vom Anwender angeforderten Speicherabschnitts dazu den UNIX-Systemruf `mmap`. Jedoch ist auch dies innerhalb des semi-dynamischen DSM nicht von Nachteil, da während der Laufzeit kein weiterer Speicher für den DSM-Heap vom Betriebssystem angefordert wird.

3.4 Konsistenz des Speichers im DSM-Heap

Es muß sichergestellt werden, daß die Struktur des Speichers im DSM-Heap auf allen Knoten identisch ist. Diesbezüglich gilt eine Abschwächung, und zwar muß diese Struktur auf einem Knoten lediglich bis zur der Stelle im DSM-Heap identisch zu der auf anderen Knoten sein, bis zu der bisher auf diesem Knoten auf den DSM-Heap zugegriffen wurde. Z.B. ist in Abbildung 3.3 ein gültiger Zustand des DSM-Heap zu sehen, trotz seiner unterschiedlichen Speicherstruktur auf den beiden Knoten ab einer gewissen Stelle. Mit der Struktur des Speichers im DSM-Heap ist hierbei ausschließlich die Anordnung und Größe der freien und vergebenen Speicherabschnitte im DSM-Heap gemeint. Für die Konsistenz der Objekte, die der Anwender in den Abschnitten erzeugt, die an ihn vergeben wurden, ist dieser selbst verantwortlich.

Diese Struktur des DSM-Heap wird ausschließlich von jenen Operationen der Verwaltung des Speichers innerhalb des DSM-Heap bestimmt, welche die Freelist oder die Variablen `dsm_break` und `start_search_p` manipulieren. Da es sich dabei um DSM-Objekte handelt, werden sie über den normalen Seitentransfermechanismus des DSM-Systems zwischen den Knoten ausgetauscht und so die Speicherstruktur des DSM-Heap unter den Knoten vererbt. Um die Konsistenz des DSM-Heap zu garantieren, müssen diese Operationen allerdings vor der gleichzeitigen Ausführung auf mehreren Knoten geschützt werden. Dies wird realisiert, indem die Funktionsaufrufe zur Speicherverwaltung mit einem Mutex [32], im folgenden Heap-Mutex genannt, umgeben werden. Es muß sich dabei um ein und denselben Mutex für alle diese Funktionsaufrufe handeln, da alle diese Funktionen die Struktur des Speichers im DSM-Heap verändern. Damit sind die Daten zur Speicherverwaltung vor gleichzeitiger Bearbeitung durch mehrere Knoten geschützt.

3.5 Vor- und Nachteile des semi-dynamischen DSM

Als Hauptnachteil des semi-dynamischen DSM ist zu nennen, daß hierbei der DSM-Heap während der Laufzeit „voll" werden kann und der Anwender dann keine weiteren DSM-Objekte mehr erzeugen kann, sofern er nicht vorher solche wieder freigibt. D.h. der Anwender kann nicht beliebig viele, beliebig große DSM-Objekte während der Laufzeit erzeugen wie er es normalerweise von einer dynamischen Speicherverwaltung im Rahmen der physischen Gegebenheiten des eingesetzten Systems erwarten würde. Demgegenüber steht jedoch die Tatsache, daß der semi-dynamische DSM mit einem sehr geringen Aufwand verwaltbar ist. Jede Erweiterung in Richtung echter Dynamik ist mit höherem Verwaltungaufwand verbunden, geht also zu Lasten der Effizienz.

Aufgrund seiner Vorteile bzgl. der Effizienz könnte der semi-dynamische DSM in einem verteilten System in einer eingeschränkten Anzahl von Anwendungsfällen eingesetzt werden, nämlich wenn der Anwender bereits vor der Laufzeit in der Lage ist abschätzen zu können, wieviel „dynamischen" DSM er insgesamt benötigen wird. Dem Anwender könnte dann überlassen werden, die Größe des DSM-Heapblocks am Anfang seines Programms festzulegen.

Für einen solchen Einsatz spricht die Tatsache, daß bei einer Speicherreservierung durch einen Prozeß unter UNIX gewöhnlich dieser Speicherbereich vorerst nur anonym in den virtuellen Speicher eingeblendet wird (z.B. beim Aufruf von `sbrk`) und dann das „Copy-On-Write"-Verfahren angewendet wird [33]. Dabei werden physische Speicherseiten erst dann alloziert, wenn der Prozeß auf die entsprechende Seite im virtuellen Speicher schreibend zugreift. D.h. aufgrund der Reservierung von Speicher (sofern dies z.B. über `sbrk` geschieht) wird ersteinmal nur Speicher aus dem virtuellen Adreßraum eines Prozesses verbraucht und noch kein physischer Speicher. Da der virtuelle Adreßraum gewöhnlich sehr viel größer als der physische Speicher ist (z.B. umfaßt der Nutzerbereich des virtuellen Adreßraumes eines Prozesses unter einem 32-bit Betriebssystem wie Linux insgesamt 3 GB), können ohne weiteres recht große Mengen an Speicher von einem Prozeß reserviert werden, ohne daß dies negative Auswirkungen auf die Gesamtmenge des auf dem System noch zur Verfügung stehenden physischen Speichers hat. Hier bedeutet dies, daß der Anwender im konkreten Fall die Schätzung der Menge des von ihm benötigten dynamischen DSM nach oben hin mit recht großem Spielraum vornehmen kann. Er könnte z.B. ohne weiteres festlegen, daß für den DSM-Heap die doppelte Menge Speicher reserviert wird als er ungefähr als von ihm benötigt eingeschätzt hat.

Weiterhin spricht für einen solchen Einsatz des semi-dynamischen DSM, daß allzugroße Mengen an gemeinsamen Daten in einem verteilten System aufgrund des dafür notwendigen Netzwerkverkehrs normalerweise nicht sinnvoll sind. Es bietet sich also im allgemeinen ohnehin nicht an, beliebig groß werdende Datenmengen gemeinsam zu nutzen, so daß es in vielen Fällen möglich sein sollte, mit der eben beschriebenen möglichen „Übertreibung“ nach oben, eine Obergrenze für den Umfang der DSM-Daten vor der Laufzeit festzulegen.

Im allgemeinen Fall kann dem Anwender jedoch nicht zugemutet werden, die maximale Menge des von ihm benötigten dynamischen DSM vor dem Programmstart abschätzen zu müssen. Weiterhin ergibt sich aus einem zu groß gewählten DSM-Heapblock für zahlreiche Teile im DSM-System ein unnötiger Mehraufwand, was bei einem allzu großem DSM-Heapblock zu erheblichen Performanceeinbußen führt. Eine adäquate Lösung für eine dynamische Verwaltung von verteiltem Speicher bietet dieser Ansatz demzufolge nicht.

Kapitel 4

Dynamischer DSM

In diesem Kapitel wird die innerhalb der bereits in der Einleitung erwähnten Studienarbeit [4] entwickelte Lösung für einen dynamischen DSM vorgestellt. Es handelt sich dabei um „echt“ dynamischen DSM, dessen Verwaltung jedoch bzgl. der Effizienz in mancherlei Hinsicht Nachteile beinhaltet. „Echt“ dynamisch bedeutet hierbei, daß der DSM zur Laufzeit bis an die physischen Grenzen des Speichers des verwendeten Systems vergrößerbar ist. Wie bereits in Abschnitt 3.3.2 erwähnt, wird aus Performancegründen für den DSM-Heap auf die Eigenschaft verzichtet, zur Laufzeit auch verkleinerbar zu sein, also nicht mehr benötigten Speicher wieder an das Betriebssystem zurückgeben zu können,

Um von dem semi-dynamischen DSM zu „echt“ dynamischen DSM zu kommen, muß der DSM-Heap während der Laufzeit vergrößerbar sein. Damit dies möglich ist, muß zum einen der Anspruch aufgegeben werden, daß der DSM-Heap einen zusammenhängenden Bereich im Speicher umfaßt, und zum anderen, daß der DSM-Heap auf allen Knoten an der selben Stelle im Speicher steht.

Ersterer Anspruch könnte beibehalten werden, wenn die im letzten Kapitel vorgestellte Lösung um die Möglichkeit erweitert wird, daß für den DSM-Heapblock, sobald dieser während der Laufzeit „voll“ wird, ein neuer, entsprechender größerer, zusammenhängender Bereich im Speicher erstellt wird und der alte DSM-Heapblock dann in den neuen kopiert wird. Ohne erheblichen zusätzlichen Aufwand zu betreiben, kann dann allerdings nicht mehr garantiert werden, daß der neue DSM-Heapblock auf allen Knoten an der gleichen Stelle im Speicher steht. Damit ist ein Hauptvorteil des semi-dynamischen DSM hinfällig, nämlich daß alle DSM-Objekte auf allen Knoten an der gleichen Stelle im Speicher stehen. Zudem wäre das komplette Kopieren des alten DSM-Heapblock bzgl. der Effizienz ein bedeutender Nachteil. Daher ist dies kein Lösungsansatz, der der Zielsetzung dieser Arbeit entsprechen würde.

In der im folgenden vorgestellten Verwaltungsmethode für dynamischen DSM werden beide erwähnten Ansprüche fallen gelassen, womit ein beträchtlicher Mehraufwand in vielerlei Hinsicht verbunden ist, der hier jeweils nur kurz dargestellt werden soll.

4.1 Veränderung des Aufbaus des DSM-Heap

Der Aufbau des semi-dynamischen DSM wird dahingehend geändert, daß ein DSM-Heapblock nunmehr immer erst dann erstellt wird, wenn während der Laufzeit Bedarf nach mehr DSM auftritt. Für die Größe des anzulegenden Heapblocks wird dabei ein Standardwert genommen. Das bedeutet einerseits, daß der erste DSM-Heapblock nicht mehr während der Initialisierungsphase des DSM-Systems erstellt wird, sondern erst zur Laufzeit, wenn der Anwender das erste Mal ein DSM-Objekt erzeugen will. Und das bedeutet andererseits, daß während der Laufzeit bei Bedarf, d.h. wann immer im zuletzt erstellten DSM-Heapblock kein Platz mehr ist, um eine Anforderung nach mehr DSM zu befriedigen, ein zusätzlicher DSM-Heapblock erstellt wird. Die Standardgröße kann dabei natürlich auch beliebig überschritten werden und die Anzahl der DSM-Heapblöcke ist unbeschränkt. In Abbildung 4.1 ist der veränderte Aufbau des DSM-Heap zu sehen.

Die Größe der Heapblöcke muß wiederum ein Vielfaches der Seitengröße sein und deren Anfangsadresse auf Seitengrenze ausgerichtet sein. Auch werden wie bisher alle DSM-Seiten innerhalb der Heapblöcke zunächst validiert, wenn es sich beim dem Knoten, auf dem sie erstellt werden, um den Startknoten handelt und invalidiert, falls es sich dabei um einen der anderen Knoten handelt.

Diese Lösung beinhaltet den entscheidenden Vorteil, daß der Anwender nicht mehr die Menge des vom ihm maximal benötigten DSM vor der Programmausführung festlegen muß, da der DSM-Heap nun während der Laufzeit am Umfang zunehmen kann. Die Festlegung der Standardgröße für den ersten anzulegenden DSM-Heapblock und die im darauffolgenden, inkrementell anzulegenden DSM-Heapblöcke kann trotzdem, diesmal jedoch optional, dem Anwender überlassen werden. Denn eine auf die Größe der DSM-Objekte abgestimmte Größe der DSM-Heapblöcke kann über die Gesamtgröße des DSM-Heap die Verschwendung von Speicher vermeiden und darüber auch durchaus die Performance des DSM-Systems beeinflussen, da für jede einzelne DSM-Seite irgendwann bestimmte Aktionen ausgeführt werden müssen. Weiß der Anwender beispielsweise, daß er während der Programmausführung oftmals DSM-Abschnitte anfordern wird, die knapp anderthalb Seiten groß sind, so wäre es sinnvoll für die DSM-Heapblöcke eine Größe von z.B. drei oder sechs Seiten zu wählen. Denn wenn diese standardmäßig z.B. zwei Seiten groß wären, würde jeweils eine halbe Seite verschwendet werden.

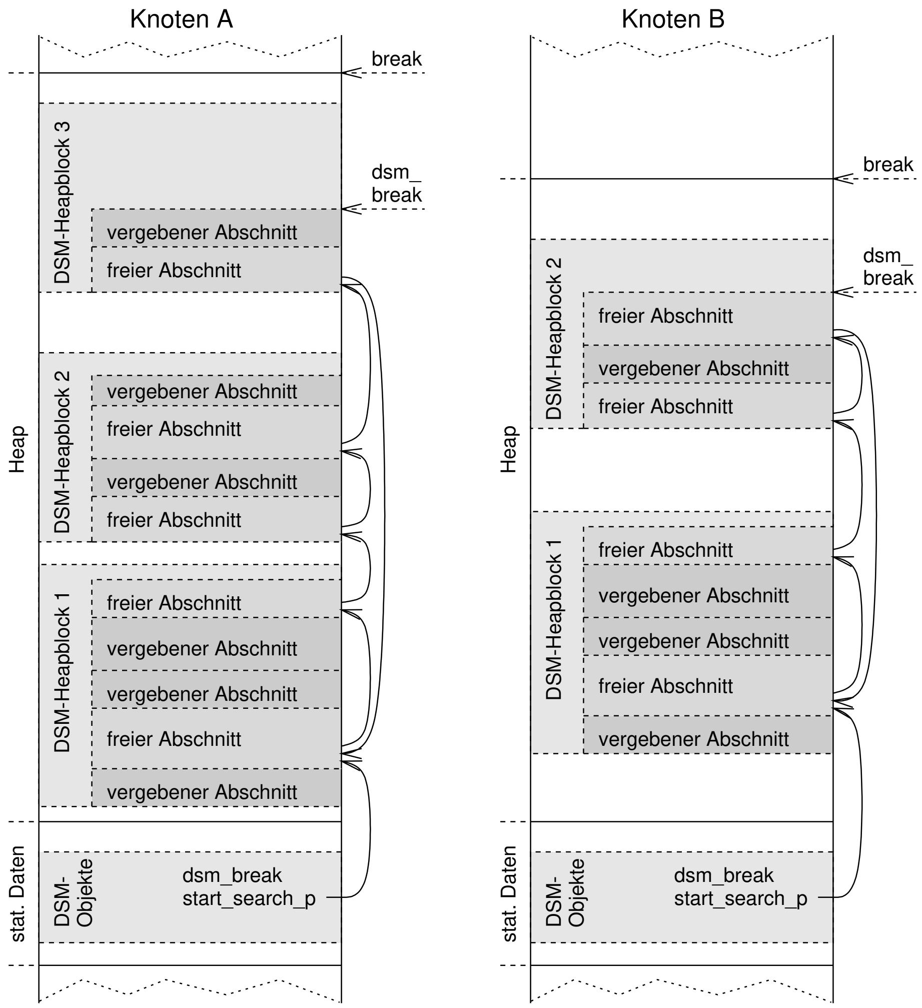

Abbildung 4.1: Veränderter Aufbau des DSM-Heap

Weiterhin ist es für den Aufwand der Verwaltung der Heapblöcke günstig, wenn von ihnen so wenige wie möglich erstellt werden müssen.

Von der Möglichkeit der Festlegung der Standardgröße für die Heapblöcke abgesehen, bleibt diese Aufteilung des DSM-Heap in mehrere Blöcke dem Anwender verborgen, d.h. sie führt zu keinerlei Einschränkungen für ihn.

4.2 Verwaltung der DSM-Heapblöcke

Die einzelnen DSM-Heapblöcke müssen auf den unterschiedlichen Knoten in der richtigen Größe, Anzahl und Reihenfolge sowie zum richtigen Zeitpunkt erstellt werden.

4.2.1 Anforderungen an den Aufbau des DSM-Heap

Größe und Reihenfolge der DSM-Heapblöcke müssen auf allen Knoten übereinstimmen. Wenn z.B. auf Knoten A DSM-Heapblock 2 mit den DSM-Seiten 4, 5 und 6 erstellt wird, so muß dieser Heapblock im folgenden auch auf allen anderen Knoten genau diese und keine weiteren Seiten beinhalten. Das ist für zahlreiche Abläufe im DSM-System, auf die hier nicht näher eingegangen werden soll, von großer Bedeutung. Im wesentlichen betrifft dies die Effizienz, jedoch wären einige dieser Abläufe ohne umfangreiche Erweiterungen nicht funktionsfähig. Z.B. existiert für jede DSM-Seite ein Eintrag in einer Seitentabelle, deren Verwaltung enorm verkompliziert werden würde, wenn die einzelnen DSM-Seiten in den Seitentabellen der verschiedenen Knoten in unterschiedlicher Reihenfolge stehen dürften. Die Lösung der in Abschnitt 2.2.1 genannten Probleme würde wesentlich erschwert, wenn die Größe oder die Reihenfolge der einzelnen Heapblöcke auf den unterschiedlichen Knoten variieren dürfte. In dem Fall würde auch die Verwaltung der Freelist erheblich erschwert werden.

Die Anzahl der Heapblöcke und damit der Zeitpunkt ihrer Erzeugung kann jedoch auf den einzelnen Knoten verschieden sein. Es müssen nicht immer auf jedem Knoten alle Heapblöcke erstellt werden, die bisher auf allen anderen Knoten erstellt wurden. Ein gänzlich neuer Heapblock wird immer dann erstellt, wenn der Platz im DSM-Heap nicht mehr ausreicht (Heapblock 3 auf Knoten A in Abbildung 4.1). Dies geschieht vorläufig nur auf dem Knoten, auf dem die Anfrage nach mehr DSM gestellt wurde. Erst wenn auf einem anderen Knoten (Knoten B in Abbildung 4.1) ein Zugriff in diesen Heapblock stattfinden soll, muß dieser auf diesem Knoten ebenfalls erstellt werden. Aufgrund der Forderung nach der Einhaltung der gleichen Reihenfolge aller Heapblöcke auf allen Knoten, müssen in dem Fall auf diesem Knoten auch alle vor diesem Heapblock liegenden Heapblöcke erstellt werden, sofern dies noch nicht geschehen ist, der DSM-Heap also bis zu dieser Stelle vervollständigt werden. Aus demselben Grund müssen auf einem Knoten, auf dem ein gänzlich neuer Heapblock erstellt werden soll, zuvor alle Heapblöcke erstellt werden, die bisher auf anderen Knoten erstellt wurden, der DSM-Heap auf diesem Knoten also komplett vervollständigt werden.

4.2.2 Informationen über die DSM-Heapblöcke

Um den DSM-Heap auf einem Knoten komplett oder teilweise vervollständigen zu können, werden auf diesem Knoten Informationen über den Aufbau des bisher erstellten DSM-Heap benötigt. Diese müssen die Anzahl und Größe der erstellten Heapblöcke und einige weitere Angaben beinhalten. Um kein aufwendiges Protokoll zur Synchronisation dieses Informationsaustausches verwenden zu müssen, werden diese Information bei jedem Seitentransfer verschickt. D.h. wann immer von einem Knoten eine Seite verschickt wird, werden die Informationen über den Aufbau des DSM-Heap auf diesem Knoten in der Nachricht mitgeschickt, die die Seite enthält. Dies ist möglich, weil diese Informationen selbst bei einer großen Anzahl erstellter Heapblöcke nur recht wenig Platz beanspruchen und die Nachrichten daher nicht wesentlich vergrößert werden.

Über den Seitentransfer, der immer stattfindet, wenn auf einem Knoten in einen Heapblock zugegriffen wird, der auf diesem Knoten noch nicht existiert, oder wenn auf einem Knoten ein gänzlich neuer Heapblock erstellt werden soll, ist garantiert, daß dieser Knoten mindestens die Informationen erhält, die notwendig sind, um den DSM-Heap bis zu der entsprechenden Stelle vervollständigen zu können. Denn dabei wird in jedem Fall mindestens eine Seite von einem Knoten, auf dem der DSM-Heap bereits mindestens bis zu dieser Stelle erstellt wurde, zu ersterem Knoten gesandt.

4.2.3 Zeitpunkt der Vervollständigung des DSM-Heap

Es muß sichergestellt werden, daß ein DSM-Heapblock, der bereits auf einem oder mehreren Knoten erstellt wurde, auf einem Knoten, auf dem dieser Heapblock noch nicht existiert, erstellt wird, bevor auf diesem Knoten tatsächlich auf diesen Heapblock zugegriffen wird.

Abgesehen von einer Ausnahme, deren Behandlung hier nicht dargelegt werden soll, kann dies über den Mechanismus der Zeigerübersetzung (Abschnitt 4.4) realisiert werden. Dabei werden u.a. die Werte aller Zeiger (also die Adressen auf die sie zeigen) übersetzt, die auf einer DSM-Seite, die gerade von einem Knoten empfangen wurde, stehen und in den DSM-Heap verweisen. Dabei müssen auch jene Zeiger übersetzt werden, die in einen Heapblock verweisen, der auf diesem Knoten noch nicht existiert. Dies ist der frühste Zeitpunkt zu dem mit einer Adresse aus einen solchen Heapblock gearbeitet wird und zu dem er darum spätestens erstellt werden muß.

Der Zugriff auf die Objekte im DSM-Heap, also auf die Heapblöcke, erfolgt ausschließlich über Zeiger. Diese Zeiger stehen in Objekten, die sich auf DSM-Seiten befinden. Wird nun auf einem Knoten auf einen noch nicht existierenden Heapblock zugegriffen, so erfolgt dieser Zugriff über eines dieser Objekte. Da der Heapblock, in den aus diesem Objekt verwiesen wird, auf diesem Knoten noch nicht existiert, wurde auf diesem Knoten auch bisher noch nicht auf dieses Objekt mit seinem aktuellen Inhalt zugegriffen. Dieser Knoten kann also nicht der aktuelle Besitzer der DSM-Seite sein, in der sich dieses Objekt befindet. Demzufolge findet ein Transfer dieser Seite von ihrem aktuellen Knoten zu diesem Knoten statt. Darauffolgend werden die Zeiger, die auf dieser Seite stehen, auf dem Empfängerknoten übersetzt, einschließlich des Zeigers, der in den nicht existierenden Heapblock verweist. Dazu muß dieser Heapblock erstellt werden (siehe Abschnitt 4.4.2).

4.3 Verwaltung des Speichers in den Heapblöcken

Aus der Aufteilung des DSM-Heap in mehrere Blöcke ergeben sich keine wesentlichen neuen Anforderungen an die Verwaltung des Speichers in diesen Blöcken. Es kann darum die Speicherverwaltung des semi-dynamischen DSM (siehe Abschnitte 3.2, 3.3, 3.4) fast ohne Änderungen übernommen werden.

Da in der Speicherverwaltung von Kernighan/Ritchie sowieso nicht davon ausgegangen wird, daß die verwalteten Speicheranschnitte in einem zusammenhängenden Bereich liegen, spielt die Aufteilung des DSM-Heap hierbei keine Rolle. D.h. die Freelist und die Funktionen der Speicherverwaltung sind ohne weitere Anpassung auch im aufgeteilten DSM-Heap einsatzfähig (zur Freelist siehe Abbildung 4.1). Die einzige diesbzgl. notwendige Änderung betrifft die Zeiger der Freelist. Auch diese Zeiger müssen während eines Transfers der DSM-Seite, auf der sie stehen, übersetzt werden. Dazu ist es notwendig, daß sie in einer Zeigertabelle registriert werden (siehe nächsten Abschnitt). Darum müssen die Funktionen der Speicherverwaltung um einige Funktionsaufrufe zur Zeigerregistrierung ergänzt werden.

Eine weitere Änderung betrifft die Variable `dsm_break`. In dieser Variable wird nun nicht mehr wie beim semi-dynamischen DSM der Abstand zwischen der Anfangsadresse des DSM-Heap und dem Ende des bereits an die Speicherverwaltung vergebenen Bereichs aus dem DSM-Heap festgehalten (Abbildung 3.3), sondern der Abstand zwischen der Anfangsadresse des zuletzt gänzlich neu erstellten DSM-Heapblocks und dem Ende des aus ihm vergebenen Bereichs. Erreicht `dsm_break` die Endadresse dieses Heapblocks, wird ein neuer Heapblock erstellt und `dsm_break` auf `0` gesetzt.

4.4 Zeiger in die DSM-Heapblöcke

Die einzelnen DSM-Heapblöcke und damit die Objekte in diesen Blöcken können auf den unterschiedlichen Knoten verschiedene Adressen haben. In so einem Fall müssen die Zeiger auf diese Objekte auf den unterschiedlichen Knoten entsprechend verschiedene Werte haben, damit sie auf dem jeweiligen Knoten auf die richtige Stelle im Speicher verweisen, d.h. auf die Stelle, an der sich dieses Objekt im Speicher dieses Knotens tatsächlich befindet.

Bei einem Transfer einer DSM-Seite können also die Werte der Zeiger, die sich auf dieser Seite befinden und in den DSM-Heap verweisen, nicht einfach durch den Empfängerknoten vom Senderknoten übernommen werden, sondern müssen an die möglicherweise unterschiedlichen Adressen der Objekte, auf die sie verweisen, angepaßt werden. Diese Anpassung wird hier realisiert, indem diese Zeigerwerte auf dem Senderknoten in eine Form übersetzt werden, aus der sich auf allen Knoten die richtige Adresse des Objekts, auf das sie verweisen, berechnen läßt und dann auf dem Empfängerknoten aus dieser Form zurückübersetzt werden, d.h. in Werte übersetzt werden, die den tatsächlichen Adressen der betreffenden Objekte auf diesem Knoten entsprechen.

Übertragen werden die übersetzten Zeigerwerte, indem sie direkt an die Adressen der Zeiger, die diese in der Nachricht haben, welche die zu transferierende Seite enthält, geschrieben werden.

4.4.1 Die Zeigertabelle

Damit die Zeigerwerte übersetzt werden können, müssen die Stellen, an denen sie auf der betreffenden DSM-Seite stehen, bekannt sein. Dazu wird hier eine Tabelle verwendet, in der die Adressen dieser Zeiger (d.h. die Adressen an denen diese Zeiger im Speicher stehen) registriert werden wie es z.B. in [28] vorgeschlagen wird.

Registrierung und Freigabe der Zeiger in der Tabelle

In dieser Tabelle müssen all jene Zeiger, die im DSM stehen und in den DSM-Heap verweisen (diese werden im folgenden mit DSM-Zeiger bezeichnet), eingetragen werden. Dies betrifft die DSM-Zeiger, die der Anwender verwendet, sowie jene, die vom DSM-System selbst verwendet werden. Gibt der Anwender einen Speicherabschnitt aus dem DSM-Heap wieder frei, so müssen alle DSM-Zeiger, die in diesen Abschnitt zeigen, wieder aus der Zeigertabelle freigegeben werden.

Dazu ist es notwendig, an geeigneten Stellen im Programm entsprechende Registrierungs- und Freigabefunktionen für die DSM-Zeiger zu plazieren. Da es aus unterschiedlichen Gründen dem Anwender nicht selbst überlassen werden soll, diese Funktionsaufrufe in sein Programm zu übernehmen, muß die Registrierung und Freigabe der DSM-Zeiger des Anwenders vom DSM-System automatisch vorgenommen werden.

Ein DSM-Zeiger wird frühstens benutzt, wenn das Objekt, auf das er verweist, erzeugt wird. Demzufolge wird er dann frühstens zwischen den Knoten transferiert und dazu übersetzt. Also muß er spätestens zu diesem Zeitpunkt in der Zeigertabelle registriert werden. Aus der Zeigertabelle wieder freigegeben werden muß ein Zeiger spätestens, wenn das Objekt, auf das er verweist, wieder freigegeben wird.

Die Objekte des Anwenders werden über die Funktionen zur Verwaltung des Speichers im DSM-Heap erzeugt und wieder freigegeben. Um die Zeiger, die auf diese Objekte verweisen, automatisch in der Zeigertabelle zu registrieren und wieder freizugeben, werden diese Funktionen um einen Parameter ergänzt, in dem die Adresse dieser Zeiger übergeben werden muß. Mit dieser Adresse wird dann innerhalb der Speicherverwaltungsfunktion die Registrierungs- bzw. Freigabefunktion aufgerufen.

Vom DSM-System selbst werden DSM-Zeiger ausschließlich innerhalb der Freelist verwendet. Zur Registrierung bzw. Freigabe dieser Zeiger werden die Registrierungs- bzw. Freigabefunktionen an den entsprechenden Stellen in den Speicherverwaltungsfunktionen aufgerufen.

Transfer der Einträge in der Zeigertabelle

Über diesen Mechanismus werden die DSM-Zeiger allerdings jeweils nur auf dem Knoten in der Zeigertabelle registriert bzw. aus ihr wieder freigegeben, auf dem die entsprechenden Funktionen der Speicherverwaltung vom Anwender aufgerufen werden. Die Zeigertabelle muß jedoch auf allen Knoten, auf denen diese Zeiger benutzt werden, für die Seite, auf der diese Zeiger stehen, die gleichen Einträge haben, da die Werte dieser Zeiger auf all diesen Knoten übersetzt werden müssen. Es müssen also Informationen über diese Einträge zwischen den Knoten ausgetauscht werden. Dies erfolgt über die gleiche Methode wie sie für den Transfer der Informationen über den Aufbau des DSM-Heap angewendet wird (siehe Abschnitt 4.2.2).

Bei jedem Transfer einer DSM-Seite werden also Informationen über die Zeigertabelle mitgeschickt und mit diesen die Zeigertabelle auf dem Empfängerknoten aktualisiert. Damit die Nachrichten dabei nicht größer werden als unbedingt nötig, werden bei einem Transfer einer DSM-Seite immer nur die Einträge der Zeigertabelle mitgeschickt, welche diese Seite betreffen.

Die Einträge der Zeigertabelle enthalten die Adressen der Zeiger. Da die Objekte, in denen diese Zeiger stehen, ebenso wie die Objekte auf die sie verweisen, auf den einzelnen Knoten an unterschiedlichen Stellen im DSM-Heap stehen können, müssen die diesbzgl. Einträge in der Zeigertabelle, ebenso wie die Werte der Zeiger, bei einem Transfer entsprechend übersetzt werden.

Es muß sichergestellt werden, daß die transferierten Informationen aus der Zeigertabelle vollständig sind. Es müssen jeweils die Einträge aller Zeiger transferiert werden, welche auf der entsprechenden Seite stehen und auf dem Empfängerknoten benutzt werden sollen. Dies wird ohne weiteren Aufwand bereits über den Seitentransfermechanismus des DSM-Systems und das dabei verwendete Seitenzugriffsprotokoll sowie die Verwendung des Heap-Mutex garantiert.

Konsistenz der Zeigertabelle

Die Zeigertabelle darf nicht auf mehreren Knoten gleichzeitig verändert werden und muß, wenn sie verändert wird, auf dem entsprechenden Knoten sämtliche Änderungen beinhalten, die zuvor auf anderen Knoten an ihr vorgenommen worden sind. Diese Anforderungen können insofern abgeschwächt werden, als daß sie nur für die Einträge über jeweils eine bestimmte Seite in der Zeigertabelle gelten müssen.

Erstere Anforderung wird über die Verwendung des Heap-Mutex erfüllt, da die Zeigertabelle ausschließlich auf dem Knoten bearbeitet wird, der der aktuelle Besitzer des Heap-Mutex ist.

Damit die zweite Anforderung erfüllt wird, müßte ein Knoten, auf dem die Zeigertabelle für eine bestimmte DSM-Seite verändert werden soll, diese Seite von dem Knoten, der die Zeigertabelle für diese Seite als letzter bearbeitet hat, oder einem Knoten, der diese Seite danach besessen hat, geschickt bekommen, und zwar nachdem er der Besitzer des Heap-Mutex geworden ist. Nur so wäre sichergestellt, daß die Zeigertabelle auf diesem Knoten alle bisherigen Veränderungen für diese Seite enthält und auf keinem anderen Knoten mehr verändert werden kann. Um dies zu gewährleisten, wird auf diesem Knoten, nachdem er den Heap-Mutex erhalten hat, über die Adresse des zu registrierenden bzw. freizugebenden Zeigers auf die entsprechende Seite zugegriffen, woraufhin eine aktuelle Kopie dieser Seite zu diesem Knoten transferiert wird, zzgl. der sie betreffenden aktuellen Einträge in der Zeigertabelle.

4.4.2 Übersetzung der Zeiger

Die Werte aller DSM-Zeiger auf einer Seite sowie die dazugehörigen Adreßeinträge in der Zeigertabelle, müssen bei einem Seitentransfer in eine Form übersetzt werden (im folgenden globale Adresse genannt), aus der auf allen Knoten die tatsächlichen Adressen (im folgenden lokale Adresse genannt) der betreffenden Objekte berechnet werden kann. Innerhalb der DSM-Threads wird diese Übersetzung wesentlich dadurch erleichtert, daß auf jedem Knoten eine Seitentabelle vorhanden ist, in der u.a. die Anfangsadressen aller DSM-Seiten gespeichert sind.

Aus einer zu übersetzenden lokalen Adresse wird auf dem Senderknoten die globale Adresse berechnet, indem aus dem Abstand der lokalen Adresse zur Anfangsadresse der Seite, auf der sich die lokale Adresse befindet, sowie dem Seitentabellenindex dieser Seite, der mit der Seitengröße multipliziert wird, eine Summe gebildet wird. Auf dem Empfängerknoten wird dann aus der globalen Adresse durch Division durch die Seitengröße zuerst der Seitentabellenindex der betreffenden Seite berechnet und dann mit Hilfe der Seitentabelle die Anfangsadresse dieser Seite ermittelt. Die lokale Adresse wird dann berechnet, indem der Seitentabellenindex, der wiederum mit der Seitengröße multipliziert wird, von der globalen Adresse abgezogen wird und das Ergebnis zur Anfangsadresse der Seite addiert wird. In Abbildung 4.2 ist dieser Algorithmus dargestellt.

Bevor dabei auf dem Empfängerknoten in der Seitentabelle die Anfangsadresse der Seite ermittelt wird, muß überprüft werden, ob diese Seite auf diesem Knoten bereits existiert, und falls dies nicht der Fall ist, der betreffende Heapblock erstellt werden (siehe Abschnitt 4.2.3).

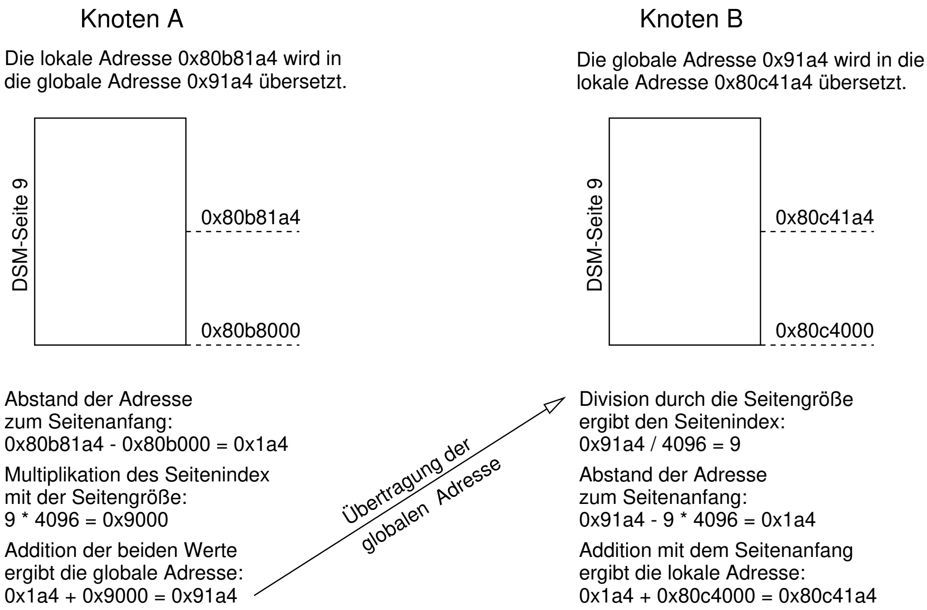

Abbildung 4.2: Übersetzung einer zu übertragenden Adresse

4.5 Vor- und Nachteile dieser Lösung

Der entscheidende Vorteil dieser Lösung gegenüber dem semi-dynamischen DSM liegt in seiner „echten“ Dynamik, d.h. in seiner unbegrenzten Größe.

Dem stehen allerdings einige Nachteile gegenüber, die hauptsächlich die Effizienz des DSM-Systems, aber auch gewisse Einschränkungen bei der Benutzung des DSM-Heap betreffen. Beide Punkte hängen im wesentlichen mit der notwendigen Zeigerübersetzung zusammen.

Die Aufteilung des DSM-Heap in mehrere Blöcke führt an sich noch nicht zu irgendwelchen Einschränkungen bei dessen Benutzung und der Aufwand für die Verwaltung der Heapblöcke verschlechtert die Performance des DSM-Systems nur unwesentlich, da die damit zusammenhängenden Operationen nur geringen Rechenaufwand erfordern und nur mit einer recht geringen Frequenz ausgeführt werden.

Die Übersetzung der Adressen und Werte der DSM-Zeiger kann sich hingegen durchaus recht negativ auf die Performance auswirken. Pro Übersetzung sind zwar nur wenige einfache Rechenschritte nötig, es ist aber nicht unwahrscheinlich, daß sehr viele Übersetzungen durchgeführt werden müssen. Verwendet der Anwender beispielsweise eine lange Liste, für deren einzelne Elemente er jeweils nur einige Bytes aus dem DSM-Heap anfordert, so passen auf eine DSM-Seite mehrere hundert Elemente dieser Liste. Dann stehen auf dieser Seite mehrere hundert DSM-Zeiger. Für jeden dieser Zeiger sind pro Seitentransfer je vier Übersetzungen notwendig (Adresse und Wert auf dem Sender- sowie dem Empfängerknoten). Es können also durchaus über tausend Übersetzungen für eine DSM-Seite pro Transfer anfallen.

Die Behandlung der DSM-Zeiger verlangt in der hier vorgestellten Form, daß dem Anwender Einschränkungen bei deren Benutzung auferlegt werden. Dies hängt mit der Zeigerregistrierung zusammen, die nur eingeschränkt automatisch funktioniert. Es werden nur jene Zeiger in der Zeigertabelle registriert bzw. aus ihr wieder freigegeben, denen über die Funktionen der Speicherverwaltung im DSM-Heap ein Speicherabschnitt zugewiesen wird bzw. für die die entsprechende Speicherfreigabefunktion aufgerufen wird. Dies ist im allgemeinen jedoch nicht ausreichend, da der Anwender eigentlich innerhalb dieser Speicherabschnitte weitere DSM-Zeiger erzeugen kann, die von der hier praktizierten Methode der Zeigerregistrierung aber nicht erfaßt werden würden. Der Anwender darf hier demzufolge ausschließlich Zeiger, denen er über die Speicherverwaltungsfunktionen ihre Werte zugewiesen hat, als DSM-Zeiger nutzen.

Die Funktionen zur Registrierung und Freigabe von DSM-Zeigern könnten zwar dem Anwender zur Verfügung gestellt werden, damit er solche an beliebigen Stellen im DSM-Heap erzeugen kann, jedoch wird darauf verzichtet, da es dabei leicht zu Anwendungsfehlern mit z.T. schwer wiegenden Folgen kommen kann. Z.B. führen Unvollständigkeiten oder Reihenfolgevertauschungen bei der Zeigerregistrierung bzw. -freigabe dazu, daß nicht alle Zeiger auf einer zu transferierenden DSM-Seite übersetzt werden oder daß an falschen Stellen auf dieser Seite versucht wird, Adressen zu übersetzen. Solche Fehler könnten zusätzlich Inkorrektheiten bei der Vervollständigung des DSM-Heap (Abschnitt 4.2.3) hervorrufen. Auch führt es z.B. zu schweren Folgefehlern, wenn ein Zeiger registriert wird, der nicht im DSM steht.

Wenn diese Funktionen dem Anwender verfügbar gemacht werden würden, würde dies weiterhin dem Ziel bei der Schnittstellen-Implementierung der DSM-Threads entgegenlaufen, diese möglichst ähnlich zu den Pthreads zu gestalten.

Jedoch wird das DSM-System auch ohne diese Schnittstellenerweiterung durch die verwendete Methode der Zeigerregistrierung recht anfällig für Anwenderfehler im Zusammenhang mit dem Einsatz des dynamischen DSM. Darum muß von diesem verlangt werden, „sinnvoll“ mit dem DSM-Heap umzugehen, da anderenfalls nicht nur beispielsweise Platz im DSM-Heap verschwendet werden würde, sondern durchaus auch Abstürze des DSM-Systems möglich sind. So führt z.B. der in Abschnitt 2.2.2 geschilderte „Anwenderfehler“ hier dazu, daß eine Adresse übersetzt wird, die sich nicht im DSM befindet, was in jedem Fall fehl schlägt.

Das in Abschnitt 3.3.2 geschilderte Problem, daß ein Durchsuchen der Freelist einen umfangreichen Seitentransfer nach sich ziehen kann, bleibt bei diesem Lösungsansatz voll erhalten, und dies ist weiterhin ein Hauptnachteil hinsichtlich der Effizienz. Aufgrund der notwendigen Zeigerübersetzung wirkt es sich diesbzgl. jedoch positiv aus, daß bei der Speicherverwaltungsmethode von Kernighan/Ritchie nur so viele Zeiger verwendet werden wie unbedingt notwendig.

Als weiterer Effizienz-Nachteil ist die Vergrößerung der Nachrichten, die eine zu transferierende DSM-Seite enthalten, anzusehen, welche durch das Mitschicken der Informationen über den Aufbau des DSM-Heap und über die Zeiger auf dieser Seite verursacht wird. Zwar benötigen erstere Informationen, so lange der DSM-Heap nicht übermäßig groß wird, nur recht wenig Platz, so daß die diesbezügliche Nachrichtenvergrößerung akzeptabel ist, für die Zeigerinformationen kann dies jedoch nicht vorrausgesetzt werden. Diese Informationen bestehen im wesentlichen aus den Adressen der Zeiger und benötigen demzufolge pro Zeiger in der Nachricht genauso viel Platz wie der Zeiger selbst im Speicher. Stehen auf einer DSM-Seite sehr viele DSM-Zeiger, wird die entsprechende Nachricht beträchtlich größer. Im diesbzgl. schlimmsten Fall belegen Zeiger den gesamten Platz auf einer Seite. Dann wächst die Nachricht um die Größe einer Seite.

Die in dieser Lösung implementierte Übersetzung der DSM-Zeiger kann jedoch in einer Hinsicht, die bisher noch nicht angesprochen wurde, einen erheblichen funktionalen Vorteil mit sich bringen. In der Einleitung wurde erwähnt, daß DSM-Threads auf einem homogenen, verteilten System ausgeführt werden können. Aus diesem Grund wird in dieser Arbeit ein dynamischer DSM für homogene Systeme entworfen. Wenn dieses für heterogene Systeme getan werden soll, sind einige zusätzliche Aspekte zu beachten. So könnte dann, selbst wenn der DSM-Heap mit der beim semi-dynamischen DSM angewendeten Methode im virtuellen Speicher der einzelnen Knoten erstellt wird, nicht mehr garantiert werden, daß alle Objekte im DSM auf allen Knoten die gleichen Adressen haben. Denn das Speicherlayout kann in einem heterogenen System auf den verschiedenen Knoten unterschiedlich sein. Für die notwendige „Extra"-Behandlung der Zeiger auf DSM-Objekte in einem heterogenen System kann die in diesem Kapitel vorgestellte Methode der Zeigerübersetzung verwendet bzw. auf einfache Art und Weise entsprechend erweitert werden.

Kapitel 5

Effizienter dynamischer DSM

Um dem Anspruch dieser Arbeit gerecht zu werden, sollen nun die beiden in den Kapiteln 3 und 4 vorgestellten Lösungsansätze für einen DSM-Heap in einer Lösung vereinigt werden, mit dem Ziel deren jeweilige Hauptvorteile zu übernehmen und so deren Nachteile weitestgehend zu beseitigen. Von dem semi-dynamischen DSM soll der Effizienzvorteil übernommen werden, daß keine Zeiger übersetzt werden müssen, und vom dynamischen DSM soll der funktionale Vorteil übernommen werden, daß er während der Laufzeit vergrößert werden kann. Dabei bleibt das grundlegende Design der Lösung aus dem letzten Kapitel im wesentlichen erhalten, es werden nun jedoch einige Modifikationen an ihr vorgenommen, welche zu einer Verbesserung der Performance führen sollen.

Zuerst erfolgt ein erneuter Umbau des DSM-Heap, um die Zeigerübersetzung überflüssig machen zu können, bevor weitere Bereiche der so entstehenden Lösung bzgl. der Effizienz optimiert werden.

Die DSM-Zeiger müssen nur dann nicht übersetzt werden, wenn alle dynamischen DSM-Objekte wie beim semi-dynamischen DSM auf allen Knoten die gleichen Adressen haben. Dies kann bei einem DSM, der während der Laufzeit vergrößert werden soll, nur erreicht werden, wenn die hinzukommenden DSM-Heapbereiche nicht wie beim dyn. DSM im letzten Kapitel an beliebigen Stellen im virtuellen Speicher der einzelnen Knoten stehen dürfen, sondern wenn diese Stellen vom DSM-System festgelegt werden, so daß sich jeder Heapblock auf allen Knoten an genau der gleichen Stelle im Speicher befindet. Ein solches Konzept wird z.B. beim „isomalloc memory allocator“ [34] verwendet.

5.1 Erstellung von DSM-Heapblöcken

Die DSM-Heapblöcke sollen an genau definierten Stellen im virtuellen Speicher stehen. Dies kann realisiert werden, indem die Heapblöcke über den

UNIX-Systemruf `mmap` erstellt werden, wobei die Anfangsadresse des einzublendenden Speicherbereichs vorgegeben wird (unter Angabe der Adresse und des `MAP_FIXED`-Flags). Dabei werden für die einzelnen Heapblöcke auf allen Knoten jeweils die gleichen Anfangsadressen gewählt. Dann haben alle DSM-Objekte auf allen beteiligten Knoten identische Adressen und die DSM-Zeiger bleiben nach einem Seitentransfer auch ohne eine Anpassung gültig.

Der Auswahl geeigneter Stellen, an denen die DSM-Heapblöcke im virtuellen Speicher erstellt werden, kommt dabei entscheidende Bedeutung zu. Es muß garantiert werden, daß die gewählten Speicherbereiche auf allen Knoten jederzeit verfügbar sind, d.h. wann immer auf einem Knoten ein Heapblock erstellt werden soll, muß der entsprechende Bereich auf diesem Knoten frei sein. Dazu ist es nötig, daß dort zu keinem Zeitpunkt von einem anderen Programmteil Objekte erzeugt werden.

5.1.1 Plazierung der DSM-Heapblöcke im Speicher

Im virtuellen Speicher eines UNIX-Prozesses befindet sich zwischen dem Ende des Datensegments bzw. des BSS-Segments des Programms und dem Beginn der Shared Libraries (d.h. dem Beginn des Bereichs in dem die Text-, Hilfs-, Daten- und BSS-Segmente der Shared Libraries stehen) ein großer, zum Zeitpunkt des Programmstarts ungenutzter Bereich. Dieser Bereich umfaßt z.B. unter Linux ca. 850 MB, wenn die Binärdatei des Programms vom heutzutage gebräuchlichen ELF-Format ist. Beim älteren `a.out`-Format sind es noch einmal ca. 600 MB mehr. Unter anderen UNIX-Systemen gelten ähnliche oder noch größere Beträge. Beginnend an der Untergrenze (d.h. dem Ende des Daten- bzw. BSS-Segments des Programms), wird in diesem Bereich der Heap des Prozesses angelegt (siehe Abschnitt 2.2). Im Programmverlauf kann der Heap theoretisch bis zur Obergrenze des Bereichs (d.h. dem Beginn der Shared Libraries) anwachsen. Aufgrund der Größe dieses Bereichs ist dies jedoch äußerst unwahrscheinlich. Der obere Teil des Bereichs, d.h. der Teil der über der Obergrenze des Heap liegt, wird vom Betriebssystem nicht genutzt, solange der Heap nicht entsprechend wächst.

Um den DSM-Heap auf allen Knoten an der gleichen Adresse im virtuellen Speicher anlegen zu können, wird dieser im oberen Teil des eben besprochenen Speicherbereichs erstellt. Als Obergrenze für den DSM-Heap wird die Anfangsadresse des Bereichs im virtuellen Speicher definiert, welcher die Daten der Shared Libraries enthält. Von dieser Obergrenze wird der DSM-Heap bei Bedarf nach unten gehend vergrößert, d.h. die DSM-Heapblöcke werden, von der Obergrenze des DSM-Heap ausgehend, untereinander angelegt (siehe Abbildung 5.1). Dies kann solange getan werden wie sich die Untergrenze des DSM-Heap und die Obergrenze des Prozeß-Heap nicht berühren bzw. überschneiden.

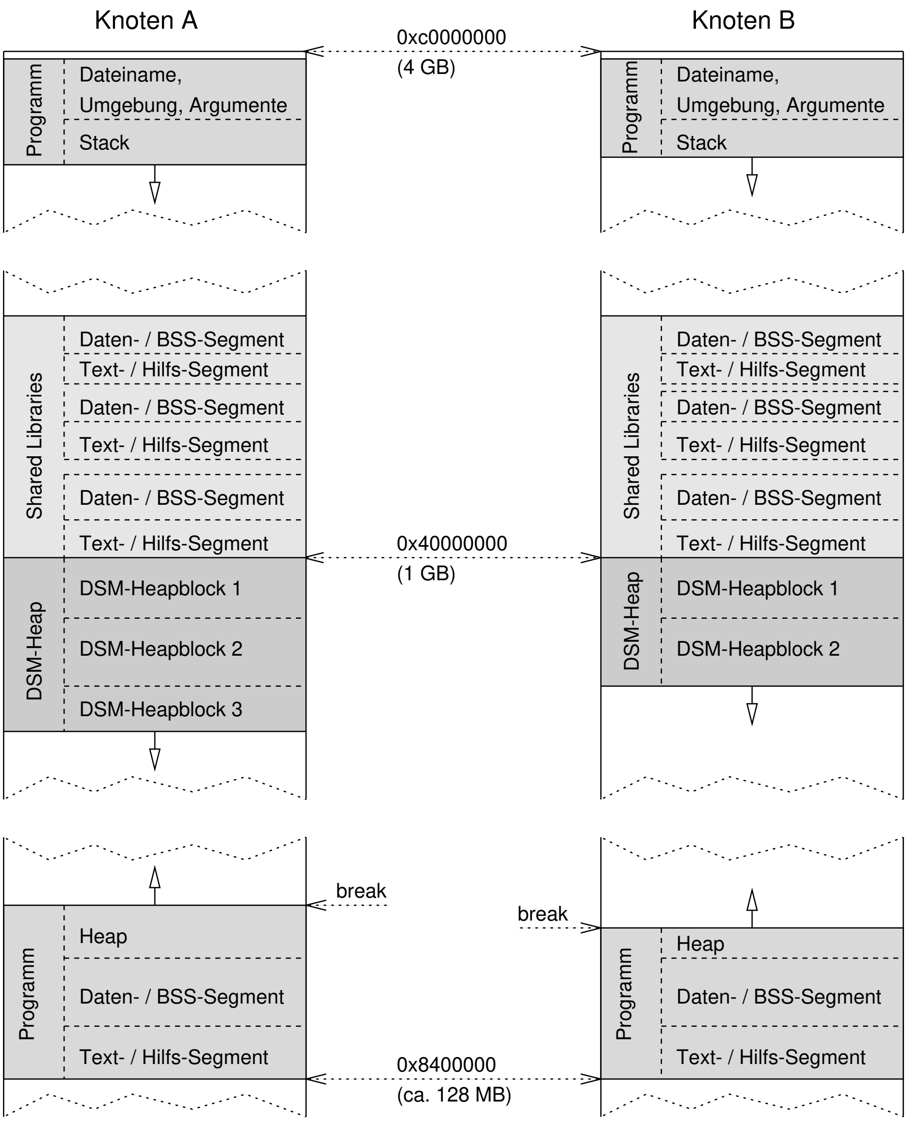

Abbildung 5.1: Speicherlayout mit neu plaziertem DSM-Heap (Linux, ELF)

5.1.2 Risiken von `mmap`

Für die Erstellung der DSM-Heapblöcke wird `mmap` unter fester Vorgabe der Anfangsadresse des Speicherbereichs, der eingeblendet werden soll, verwendet. Mit dieser Anwendung von `mmap` sind jedoch gewisse Risiken verbunden, die nicht unbeachtet bleiben dürfen.

Ein Problem kann sich dabei aus der Tatsache ergeben, daß `mmap` nicht fehlschlägt, wenn ein Speicherbereich eingeblendet werden soll, der bereits zuvor eingeblendet wurde. D.h. es ist möglich, mit `mmap` ein und den selben Speicherbereich mehrmals einzublenden. Dieses Verhalten von `mmap` wird bei bestimmten Anwendungen gezielt genutzt, kann jedoch auch zu Problemen führen, wenn man sich dieser Eigenschaft nicht bewußt ist. Wenn ein Speicherbereich wie dies bei der Erzeugung der DSM-Heapblöcke geschieht mit `mmap` anonym eingeblendet wird, dann wird dieser Speicherbereich von `mmap` komplett mit Nullen initialisiert. Wurde dieser Speicherbereich, oder ein Teil von ihm, bereits zuvor eingeblendet, so sind alle Daten, die sich möglicherweise vor dem zweiten Einblenden in ihm, bzw. dem betreffenden Teil von ihm, befunden haben, danach komplett verloren.

Das bedeutet, daß ein Anwender, der `mmap` in der Art nutzt wie es bei der Erzeugung der DSM-Heapblöcke getan wird sich selbst darum kümmern muß, daß Speicherbereiche nicht mehrmals eingelbendet werden, sofern dies nicht explizit gewünscht wird. Da ein Anwender der DSM-Threads im Normalfall nicht wissen muß, an welcher Stelle sich der DSM-Heap im virtuellen Speicher befindet, muß ihm verboten werden, `mmap` in der hier beschriebenen Weise zu verwenden, damit er nicht aus Versehen einen Bereich aus dem DSM-Heap nochmals einblendet, oder in einem von ihm bereits eingeblendeten Bereich ein DSM-Heapblock erstellt wird.

Eine Benutzung von `mmap` ohne feste Vorgabe der Adresse des einzublendenden Speichersbereichs durch einen Anwender der DSM-Threads stellt hingegen kein Problem dar, da solche Speicherbereiche oberhalb der Anfangsadresse der Stelle, an der die Daten der Shared Libraries stehen (siehe Abbildung 5.1), eingeblendet werden und es demzufolge in dem Fall nicht zu Überschneidungen mit dem DSM-Heap kommen kann.

Ein weiteres Problem kann sich aus der geschilderten Anwendungsweise von `mmap` zur Erstellung der DSM-Heapblöcke für den Heap des Prozesses ergeben, da `mmap` auch erfolgreich ist, wenn als Anfangsadresse für den einzublendenden Speicherbereich eine Adresse angegeben wird, welche im Prozeß-Heap liegt. Auch in diesem Fall würden alle Daten, die sich bis dahin in dem entsprechenden Bereich im Prozeß-Heap befanden, mit Nullen überschrieben werden. Dieser Fall kann eintreten, wenn der DSM-Heap bzw. der Prozeß-Heap im Programmverlauf derart anwachsen, daß sich die Untergrenze des DSM-Heap und die Obergrenze des Prozeß-Heap („break", siehe Abbildung 5.1) überschneiden. Obwohl dieses Problem, aufgrund der enormen Größe, die dabei ein oder beide Heap annehmen müßten (siehe Abschnitt 5.1.1) und aufgrund der zu den Speichermengen in Abschnitt 3.5 gemachten Aussagen, eher von theoretischer Natur ist, kann nicht grundsätzlich von vornherein ausgeschlossen werden, daß es auftritt.

Um dies zu verhindern, wird immer, wenn der DSM-Heap auf einem Knoten vergrößert werden soll, zuvor mit Hilfe des Aufrufs von `sbrk(0)` (liefert die Adresse, an der sich der „break" aktuell befindet) getestet, ob der anzulegende Heapblock auf diesem Knoten in den Prozeß-Heap hineinragen würde und wenn dies der Fall ist, die Erzeugung des Heapblocks entsprechend abgewiesen. Betrifft dies einen gänzlich neuen DSM-Heapblock (siehe Abschnitt 4.2.1), so wird lediglich die betreffende Anfrage des Anwenders nach mehr DSM negativ beantwortet und im Anwenderprogramm kann entsprechend darauf reagiert werden, indem z.B. ein entsprechend großer, nicht mehr benötigter Bereich aus dem DSM wieder freigegeben und die Anfrage wiederholt wird oder z.B. das Anwenderprogramm kontrolliert beendet wird. Betrifft dies hingegen einen DSM-Heapblock, der infolge einer Vervollständigung des DSM-Heap erstellt wird (siehe Abschnitt 4.2), so ist keine weitere Ausführung des Programms mehr möglich und es wird sofort mit einer entsprechenden Fehlermeldung beendet.

5.2 Verwaltung der DSM-Heapblöcke

Bzgl. der Anforderungen an den Aufbau des DSM-Heap auf den einzelnen Knoten gelten die in Abschnitt 4.2.1 gemachten Aussagen zu dem dynamischen DSM aus dem letzten Kapitel in gleichem Maße für dessen effizientere Version, die in diesem Kapitel dargestellt wird. Unterschiede ergeben sich allerdings hinsichtlich der auszutauschenden Informationen über den Aufbau des DSM-Heap und hinsichtlich des Zeitpunktes der Vervollständigung des DSM-Heap.

5.2.1 Transfer der Informationen über die Heapblöcke

Die zwischen den Knoten auszutauschenden Informationen über den Aufbau des DSM-Heap (im folgenden mit Heapinformationen bezeichnet), d.h. über die einzelnen DSM-Heapblöcke, enthalten zwar z.T. andere Angaben als beim DSM-Heap in Kapitel 4, haben jedoch den gleichen Umfang. Für jeden Heapblock muß dessen Größe sowie der Index der ersten Seite dieses Heapblocks in der Seitentabelle (siehe Abschnitt 5.3) übertragen werden. Es gibt zahlreiche Möglichkeiten, diesen Informationsaustausch zu realisieren. Die Auswahl einer dieser Möglichkeiten muß hier, dem Ziel der Arbeit entsprechend, hauptsächlich von deren Effizienzmerkmalen bestimmt werden. Der notwendige Nachrichtentransfer soll hinsichtlich der Anzahl und des Umfangs der Nachrichten so gering wie möglich gehalten werden.

Beim dyn. DSM aus Kapitel 4 werden die Heapinformationen bei jedem Versenden einer Seite mitgeschickt, unabhängig davon, ob die Informationen auf dem Empfängerknoten benötigt werden oder nicht. Dabei entsteht zwar keinerlei zusätzlicher Nachrichtentransfer, jedoch werden alle Nachrichten, mit welchen Seiten verschickt werden, um ein gewisses Maß vergrößert. Dies betrifft potentiell unnötig viele Nachrichten, da die Heapinformationen dabei auf dem Empfängerknoten wahrscheinlich oftmals bereits in gleichem oder größerem Umfang zur Verfügung stehen wie auf dem Senderknoten.

Hinsichtlich der Anzahl der Nachrichten, mit denen die Heapinformationen verschickt werden, besteht also ein gewisses Potential zur Verbesserung der Effizienz.

Die Heapinformationen als DSM-Daten

So wie im „Normalfall“ mit dynamischen Daten verfahren wird, die auf allen Knoten zur Verfügung stehen sollen, könnten die Heapinformationen im DSM-Heap abgelegt und so über den „normalen“ Zugriffsmechanismus für DSM-Daten zwischen den Knoten transferiert werden. Daraus würden sich jedoch einige recht schwerwiegende Probleme ergeben.

Einerseits ließe sich dies nur mit erheblichem Mehraufwand realisieren. Die Heapinformationen müssen auf einem Knoten u.a. dann vollständig vorliegen, wenn auf ihm auf ein Objekt zugegriffen wird, welches sich in einem Heapblock befindet, der auf diesem Knoten noch nicht erstellt wurde, weil der DSM-Heap spätestens dann entsprechend aktualisiert werden muß (siehe Abschnitt 5.2.3). Da der Zugriff auf ein solches Objekt über eine Variable erfolgte, welche sich im DSM befindet (siehe Abschnitt 2.2.2), befindet sich das DSM-System auf diesem Knoten zu diesem Zeitpunkt möglicherweise gerade in der Abarbeitung eines Zugriffsfehlers auf die Seite, auf der sich die betreffende Variable befindet. Aufgrund des dann erfolgenden Zugriffs auf die Heapinformationen würde dann auf diesem Knoten mindestens ein weiterer Seitenzugriffsfehler auftreten, wenn dieser Knoten zu diesem Zeitpunkt nicht zufällig gerade der Besitzer der betreffenden Seite bzw. Seiten ist. Das Auftreten derartiger rekursiver Seitenzugriffsfehler ist im DSM-System jedoch nicht gestattet, d.h. an den DSM-Threads müßten dazu entsprechende Erweiterungen vorgenommen werden.

Ein Mehraufwand ergebe sich aber noch in anderer Hinsicht. Die Heapinformationen werden immer dann erweitert, wenn ein gänzlich neuer Heapblock erstellt wird. Dies geschieht innerhalb der Funktionen zur Verwaltung des Speichers im DSM-Heap, wenn vom Anwender ein entsprechend großer Speicherabschnitt angefordert wird. Diese Funktionen müßten zu diesem Zeitpunkt jedoch ein zweites Mal aufgerufen werden, um einen Speicherabschnitt für die Informationen über den neuen Heapblock anzufordern. Solche rekursiven Aufrufe der Speicherverwaltungsfunktionen sind jedoch nicht möglich, da damit die Freelist zerstört werden würde. Diese Rekursion ließe sich zwar durchbrechen, z.B. indem die Heapinformationen erst nach der Beendigung der Funktionen zur Bereitstellung eines DSM-Abschnitts für den Anwender erweitert werden würden. Allerdings ist der Aufwand dafür nicht gering, da möglicherweise auch dafür wieder ein neuer Heapblock erstellt werden müßte und wiederum neue Informationen anfallen würden.

Der entscheidende Nachteil besteht bei diesem Ansatz allerdings darin, daß die benötigten Heapinformationen zwar gezielt geholt werden und diese somit nicht unnötig oft transferiert werden, daß damit aber ein nicht unbeträchtlicher, zusätzlicher Seitentransfer verbunden ist. Wann immer auf einem Knoten auf ein DSM-Objekt zugegriffen wird, das auf einer DSM-Seite steht, deren Besitzer dieser Knoten nicht gerade ist, müssen auf diesem Knoten Informationen über alle bisher erstellten Heapblöcke vorliegen. Denn es kann jederzeit sein, daß aus diesem Objekt in einen Heapblock verwiesen wird, der auf diesem Knoten noch nicht erstellt wurde und der DSM-Heap demzufolge entsprechend aktualisiert werden muß. D.h. wann immer auf einem Knoten ein Zugriffsfehler auf eine DSM-Seite auftritt, müssen zusätzlich zu dieser Seite die Seiten mit den Heapinformationen zu diesem Knoten transferiert werden, sofern dieser Knoten nicht über aktuelle Kopien dieser Seiten verfügt. Dieses Verhalten kann zwar teilweise optimiert werden und der notwendige Seitentransfer so reduziert werden, aber er kann nicht völlig abgeschaltet werden.

Andere Transfermethoden für die Heapinformationen

Die Heapinformationen können auch über eigenständige Nachrichten unter den Knoten ausgetauscht werden. Z.B. könnte von einem Knoten, auf dem ein gänzlich neuer Heapblock erstellt wurde, ein Broadcast mit den Informationen über diesen Heapblock an alle anderen Knoten verschickt werden (siehe Abbildung 5.2, A). Dies hat den Vorteil, daß diese Informationen immer nur Angaben über einen einzelnen Heapblock enthalten müssen und nicht über alle bisher erstellten. Allerdings würden diese Informationen auch immer an Knoten verschickt, die sie möglicherweise niemals benötigen. Noch schwerer wiegt allerdings der Nachteil, daß dabei garantiert werden muß, daß die Informationen von allen Knoten tatsächlich empfangen wurden und demzufolge immer von allen Empfängern eine Bestätigung an den Sender des Broadcast geschickt werden müßte.

Eine etwas bessere Möglichkeit wäre, die Heapinformationen nur auf Anfrage zu verschicken. D.h. wann immer auf einem Knoten der DSM-Heap aktualisiert werden muß und darum die Informationen über die bisher erstellten Heapblöcke benötigt werden, wird von diesem Knoten eine Anfrage nach diesen Informationen verschickt (siehe Abbildung 5.2, B). Die Anfrage müßte dabei an einen Knoten gerichtet werden, welcher über vollständige Heapinformationen verfügt. Dies muß allerdings erst festgestellt werden, was wiederum mit gewissem Aufwand verbunden ist. Desweiteren geht auch mit dieser Transfermöglichkeit ein erhöhter Nachrichtenaustausch einher.

Verbesserung der bisher angewendeten Transfermethode

Es gibt zahlreiche weitere Methoden wie sie im letzten Abschnitt beispielhaft angegeben wurden die für den Transfer der Heapinformationen genutzt

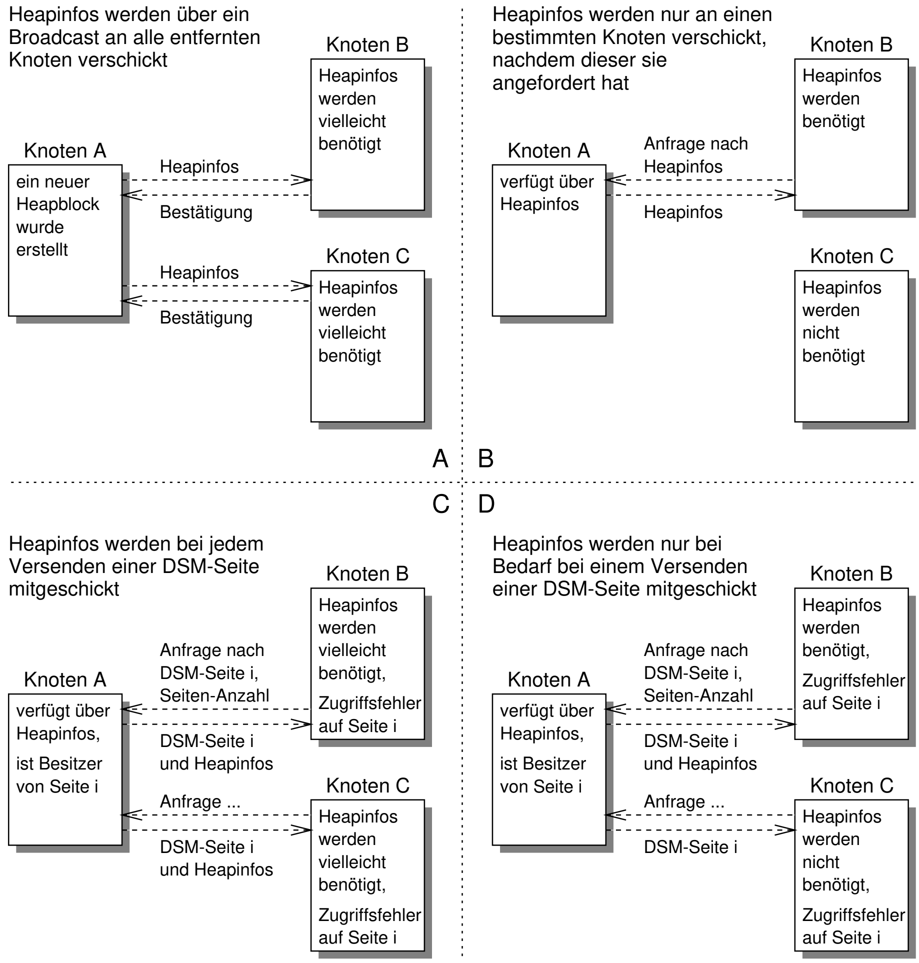

Abbildung 5.2: Transfermöglichkeiten für die Heapinformationen

werden könnten, jedoch sind alle diese Methoden mit zusätzlichem Nachrichtentransfer verbunden. Hinsichtlich der Effizienz, d.h. in dem Fall hinsichtlich der Nachrichtenmenge, wird darum die für den dyn. DSM in Kapitel 4 genutzte Transfermethode (siehe Abbildung 5.2, C) im wesentlichen als durchaus sinnvoll erachtet und wird demzufolge für den effizienten dyn. DSM übernommen. Es wird allerdings eine Optimierung an ihr vorgenommen.

Die Anzahl der Nachrichten, die um die Heapinformationen ergänzt werden, kann reduziert werden, wenn die Informationen nicht wie bisher grundsätzlich, sondern nur bei Bedarf verschickt werden. D.h. von dem Sender einer DSM-Seite sollen die Heapinformationen nur dann mitgeschickt werden, wenn diese von dem Empfänger der DSM-Seite auch tatsächlich benötigt werden. Um diesen Bedarf auf dem Senderknoten feststellen zu können, wird von dem Sender der Seitenanfrage, d.h. dem Knoten auf dem die Heapinformationen eventuell benötigt werden, in der Nachricht, die die Seitenanfrage enthält, die Anzahl der auf ihm bisher erstellten DSM-Seiten mitgeschickt. Die Größe dieser Nachrichten nimmt dadurch nur minimal zu. Auf dem Knoten, der die Seitenanfrage letztlich erhält und der dann die Seite verschickt, wird dann anhand der Anzahl der DSM-Seiten auf beiden Knoten überprüft, ob der Sender der Seitenanfrage, also der Empfänger der DSM-Seite, den DSM-Heap aktualisieren muß, also die Heapinformationen benötigt. Nur in diesem Fall, d.h. wenn auf dem Knoten, dem die Seite geschickt wird, weniger DSM-Seiten existieren als auf dem Knoten, der die Seite verschickt, werden die entsprechenden Heapinformationen mit der Seite mitgeschickt (siehe Abbildung 5.2, D).

5.2.2 Vollständigkeit der Heap-Informationen

Spätestens zu dem Zeitpunkt, zu welchem der DSM-Heap auf einem Knoten vervollständigt werden soll, müssen auf diesem Knoten vollständige Heapinformationen zur Verfügung stehen, d.h. sie müssen Angaben über alle Heapblöcke bis zu der Stelle im DSM-Heap beinhalten, bis zu der dieser vervollständigt werden soll.

Um dies zu garantieren, wäre es hinreichend, wenn dieser Knoten bis zu diesem Zeitpunkt eine Seitenanfrage an einen Knoten geschickt hat, auf dem der DSM-Heap bereits mindestens bis zu dieser Seite erstellt wurde, und er diese Seite dann geschickt bekommen hat. Dieses Verhalten ist bereits gewährleistet, ohne daß dazu extra Schritte unternommen werden müßten.

Der DSM-Heap muß auf einem Knoten spätestens dann vervollständigt werden, wenn auf ihm auf ein Objekt zugegriffen werden soll, welches sich in einem Heapblock befindet, der auf diesem Knoten noch nicht erstellt wurde, oder wenn auf diesem Knoten ein gänzlich neuer Heapblock erstellt werden soll, weil der Speicher in den bereits erstellten Heapblöcken nicht mehr ausreicht. Im ersten Fall muß der DSM-Heap mindestens bis zu diesem bestimmten Heapblock vervollständigt werden und im zweiten Fall muß der DSM-Heap komplett vervollständigt werden (siehe Abschnitt 4.2.1). In beiden Fällen ist die Vollständigkeit der Heapinformationen über den Zugriff auf DSM-Seiten sichergestellt, deren aktueller Besitzer ein Knoten ist, der über die erforderlichen Informationen verfügt.

Nämlich findet im ersten Fall der Zugriff auf das Objekt über eine Variable statt, welche sich auf einer Seite befindet, deren aktueller Besitzer ein Knoten ist, auf dem der entsprechende Heapblock unbedingt bereits erstellt worden sein muß. Denn auf dem Knoten, auf dem diese Variable ihren Wert erhalten hat, nämlich der Verweis auf das entsprechende Objekt, muß der Heapblock, in dem sich dieses Objekt befindet, unbedingt bereits existieren. Wird nun auf einem anderen Knoten auf diese Variable zugegriffen, erhält er die Seite mit dieser Variable von ersterem Knoten, zuzüglich der Informationen über alle bisher auf diesem Knoten erstellten Heapblöcke. Diese Informationen werden dann mindestens zu jedem Knoten weitergegeben, der auf diese Variable zugreift. So wird auf Knoten B in Abbildung 5.3 über die Variable c auf ein Objekt in Heapblock 2 zugegriffen, der auf Knoten B noch nicht existiert und die Heapinformationen dann von Knoten A, der eine aktuelle Kopie der Seite besitzt, auf der sich c befindet, zu Knoten B transferiert. Die Informationen über den Heapblock 3 werden dabei nicht benötigt, da der DSM-Heap auf Knoten B zu diesem Zeitpunkt lediglich bis zum Heapblock 2 vervollständigt werden muß.

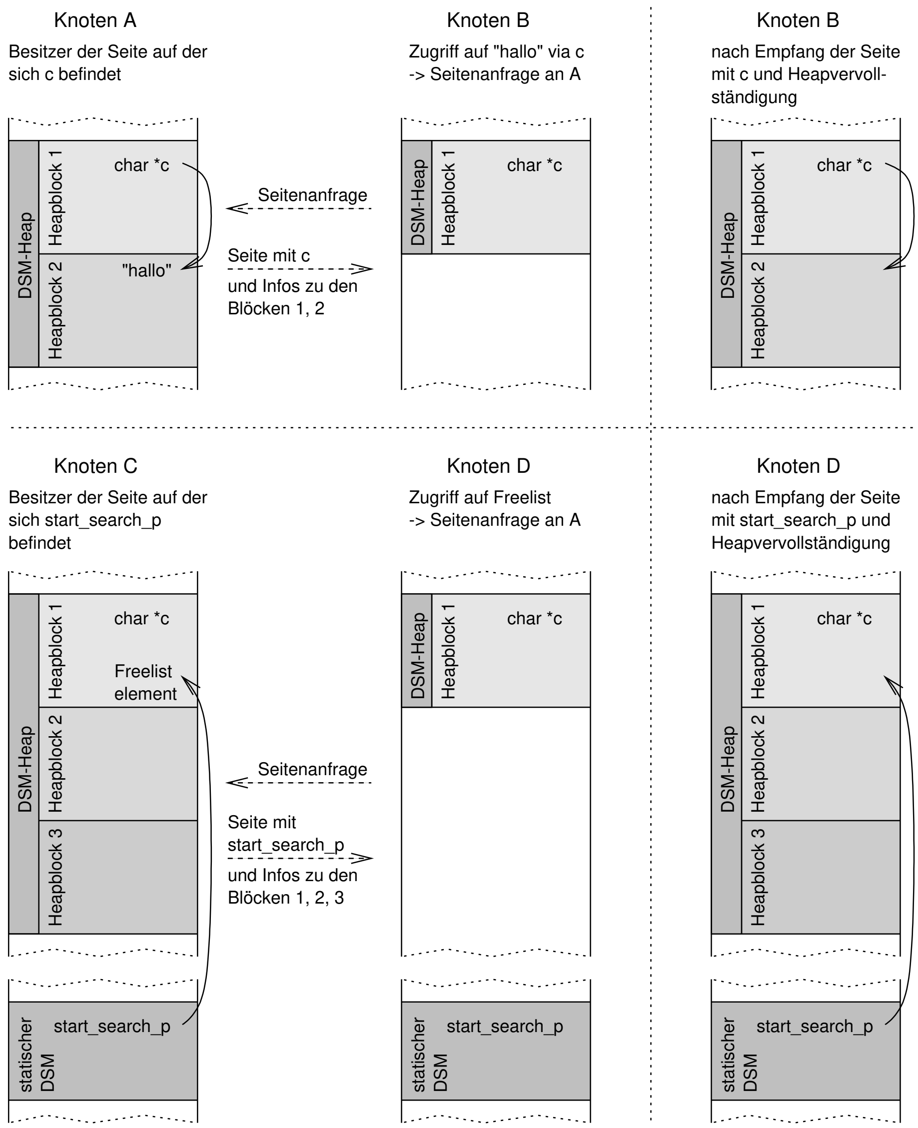

Abbildung 5.3: Vollständigkeit der Heapinformationen

Im zweiten Fall erhält ein Knoten, auf dem ein gänzlich neuer Heapblock erstellt wird, durch den Zugriff auf die Freelist unbedingt Informationen über alle bisher irgendwo erstellten Heapblöcke. Neue Heapblöcke werden immer nur auf dem Knoten erstellt, der gerade der Besitzer des Heap-Mutex (siehe Abschnitt 3.4) ist, d.h. nie von zwei Knoten gleichzeitig, und desweiteren wird dabei nach der Erstellung eines Heapblocks und vor der Freigabe des Heap-Mutex in jedem Fall schreibend auf `start_search_p` (siehe Abschnitt 3.3.1) zugegriffen. Bevor nun auf einem anderen Knoten eventuell ein weiterer, gänzlich neuer Heapblock erstellt wird, wird unbedingt vorher auf die Freelist, also zuerst auf `start_search_p`, zugegriffen. Mit der Seite, auf der sich `start_search_p` befindet, erhält dieser Knoten auch die benötigten Informationen über alle bisher erstellten Heapblöcke. Die Informationen über gänzlich neu erstellte Heapblöcke werden also von einem Besitzer dieser Seite zu ihren nächsten Besitzer weitergegeben, und damit auch die Informationen über alle bisher erstellten Heapblöcke.

Z.B. wird auf Knoten D in Abbildung 5.3 vom Anwender ein Speicherabschnitt aus dem DSM angefordert, wobei ein Durchsuchen der Freelist im weiteren ergeben würde, daß in dem Heapblock 1 nicht mehr genug Platz dafür vorhanden ist, weswegen ein gänzlich neuer Heapblock auf Knoten D erstellt werden müßte. Dies muß insgesamt Heapblock 4 sein und nicht etwa Heapblock 2 auf Knoten D. Dazu muß zuvor auf Knoten D eine Vervollständigung des DSM-Heap stattfinden, also die Heapblöcke 2 und 3 erstellt werden, wozu die entsprechenden Informationen benötigt werden. Diese bekommt er rechtzeitig von Knoten C, da auf Knoten D dabei als erstes auf `start_search_p` zugegriffen wird und demzufolge die entsprechende Seite von deren aktuellen Besitzer Knoten C zu Knoten D transferiert wird. Damit bekommt Knoten D auch die Informationen über die Heapblöcke 2 und 3 von Knoten C.

Konsistenz der Heapinformationen

Die Vollständigkeit der Heapinformationen beinhaltet auch, daß diese auf den verschiedenen Knoten keine unterschiedliche Angaben beinhalten dürfen. Diesbzgl. gelten für die Heapinformationen die gleichen Bedingungen wie für die DSM-Heapblöcke selbst (siehe Abschnitt 4.2.1), d.h. sie dürfen zwar auf den einzelnen Knoten unterschiedlichen Umfang haben, je nachdem wie weit der DSM-Heap jeweils bereits erstellt wurde, müssen bis zu dieser Stelle allerdings auf allen Knoten identisch sein. Wie für die DSM-Heapblöcke selbst ist dies auch für die Heapinformationen durch den Austausch der Heapinformationen unter den Knoten sowie die Verwendung des Heap-Mutex für die Funktionen der Speicherverwaltung sichergestellt. Ausschließlich innerhalb dieser Funktionen werden gänzlich neue Heapblöcke erstellt und die Heapinformationen entsprechend erweitert.

5.2.3 Zeitpunkt der Vervollständigung des DSM-Heap

Der DSM-Heap muß auf einem Knoten spätestens dann teilweise oder komplett vervollständigt werden, wenn auf ihm eine DSM-Seite empfangen wird, die einen Verweis in einen Heapblock enthält, der auf diesem Knoten noch nicht erstellt wurde, oder wenn ein gänzlich neuer Heapblock auf ihm erstellt werden soll. Bei der Lösung für einen dyn. DSM in Kapitel 4 kann ersteres während der Zeigerübersetzung festgestellt werden (Abschnitt 4.2.3), jedoch muß dabei noch ein zusätzlicher Test an anderer Stelle durchgeführt werden, um die zweite Anforderung in allen Fällen erfüllen zu können.

Aufgrund der weggefallenen Zeigerübersetzung muß nun ein neuer geeigneter Zeitpunkt zur Vervollständigung des DSM-Heap gefunden werden. Um feststellen zu können, ob aus einer empfangenen DSM-Seite in einen auf diesem Knoten noch nicht existierenden Heapblock verwiesen wird, müßte die Seite nunmehr in einem zusätzlichem Schritt nach solchen Verweisen durchsucht werden. Dies kann jedoch in Hinblick auf die Effizienz keine adäquate Lösung sein.

Da nun also auf einem Knoten, der gerade eine DSM-Seite empfangen hat, keine Informationen darüber verfügbar sind, ob aus der Seite in Heapblöcke verwiesen wird, die auf diesem Knoten eventuell noch nicht erstellt wurden, muß der DSM-Heap auf diesem Knoten zu diesem Zeitpunkt soweit vervollständigt werden, daß auf ihm mindestens die gleichen Heapblöcke existieren wie auf dem Knoten, von dem er diese Seite geschickt bekommen hat. D.h. es wird nun grundsätzlich bereits unmittelbar nachdem auf einem Knoten eine DSM-Seite empfangen wurde, der DSM-Heap auf diesem Knoten entsprechend den gleichzeitig empfangenen Heapinformationen vervollständigt, d.h. es werden auf diesem Knoten alle Heapblöcke erstellt, die laut der empfangenen Heapinformationen auf dem Knoten, der die Seite gesendet hat, bereits erstellt wurden, aber auf ersterem Knoten noch nicht.

Der Aufbau des DSM-Heap wird somit von einem Knoten, der ein bestimmte DSM-Seite besitzt, zum nächsten Besitzer dieser Seite weitergegeben und so an alle zukünftigen Besitzer dieser Seite vererbt.

Effizienz dieser Methode

Hinsichtlich der Effizienz mag dieses Verfahren der Heapvervollständigung unvorteilhaft erscheinen, da auf einzelnen Knoten möglicherweise Heapblöcke erstellt werden, die dort nie benötigt werden. Daran führt jedoch kein Weg vorbei, da es keine Möglichkeit gibt, Informationen darüber zu erlangen, auf welche Objekte, die auf einer empfangenen Seite stehen und aus denen in einen anderen Heapblock verweisen wird, vom Anwender tatsächlich zugegriffen wird und auf welche nicht. D.h. es ist nicht möglich festzustellen, in welche Heapblöcke (auf einem bestimmten Knoten) der Anwender tatsächlich zugreift und in welche nicht. Also müssen auf allen Knoten alle Heapblöcke erstellt werden, in die auf dem jeweiligen Knoten ein solcher Zugriff stattfinden könnte. Auch bei der Heapvervollständigung während der Zeigerübersetzung werden Heapblöcke auf einzelnen Knoten erstellt, die dort eventuell nie benötigt werden, da der Anwender keineswegs auf alle Zeiger, die auf einer empfangenen Seite stehen, tatsächlich zugreifen muß, also auch nicht zwangsläufig auf allen Knoten in die Heapblöcke, in die diese Zeiger verweisen, zugreifen muß.

In Abhängigkeit von der Gesamtanzahl der Heapblöcke und damit der Anzahl der Heapblöcke, die auf einzelnen Knoten eventuell umsonst erstellt werden, kann dieser Effizienznachteil jedoch hinsichtlich der Geschwindigkeit mehr oder weniger vernachlässigt werden, da eine Heapvervollständigung, sofern nicht allzuviele gänzlich neue Heapblöcke erstellt werden, sofern also der DSM-Heap nicht insgesamt übermäßig stark anwächst, nicht allzuoft pro Knoten durchgeführt werden muß. Denn jeder Heapblock muß auf jedem Knoten natürlich nur einmal erstellt werden. Hinsichtlich der dabei auf einzelnen Knoten stattfindenden Verschwendung von gewissen Mengen an Speicher gilt ähnliches, da dies nur den virtuellen Speicher betrifft und nicht den physischen. Denn bei einem anonymen Einblenden eines Speicherbereichs über `mmap` wird das „Copy-On-Write"-Verfahren angewendet (siehe dazu Abschnitt 3.5), d.h. wenn auf einem Knoten in einzelne Heapblöcke oder einzelne DSM-Seiten nicht zugegriffen wird, dann werden für diesen Heapblock bzw. diese Seiten auf diesem Knoten auch keine physischen Speicherseiten alloziert.

5.3 Die Seitentabelle

Innerhalb der DSM-Threads wird für die Verwaltung der DSM-Seiten eine Seitentabelle verwendet, in der zahlreiche Angaben für jede der DSM-Seiten abgelegt werden. Beim Erstellen eines Heapblocks muß die Seitentabelle auf dem betreffenden Knoten um die Einträge für die neuen DSM-Seiten erweitert werden und dazu entsprechend vergrößert werden.

Beim dyn. DSM aus Kapitel 4 wird dies über ein `realloc` realisiert, das für die Seitentabelle für jeden neuen Heapblock auf einem Knoten ausgeführt wird. Da die maximale Größe der Seitentabelle nicht vor der Laufzeit des DSM-Systems festgelegt werden kann, weil dies nicht hinnehmbare Folgen für die maximale Größe des DSM-Heap zu Folge hätte, kann auf ein `realloc` für die Seitentabelle, trotz seiner negativen Eigenschaften bzgl. der Effizienz, nicht ganz verzichtet werden. Allerdings kann dieser Effizienznachteil abgemildert werden, wenn die Anzahl der Ausführungen von `realloc` gesenkt wird. Dies kann auf einfache Weise realisiert werden, indem die Seitentabelle bei ihrer erstmaligen Erzeugung und bei jedem folgenden `realloc` mit Überkapazitäten in geeigneter Menge erstellt wird, so daß nicht zu jeder Erweiterung ein `realloc` notwendig wird. Ebenso wird beim effizienten dyn. DSM verfahren.

5.3.1 Konsistenz der Seitentabelle

Im Umfang können die Seitentabellen auf den unterschiedlichen Knoten variieren, da auch die Anzahl der Heapblöcke variieren kann, jedoch müssen sie in der Reihenfolge ihrer Einträge auf allen Knoten übereinstimmen. Die einzelnen DSM-Seiten müssen auf allen Knoten an jeweils der gleichen Stelle in der Seitentabelle stehen, d.h. sie müssen überall den gleichen Index haben. Dazu ist es notwendig, daß die DSM-Seiten aus den Heapblöcken, die auf unterschiedlichen Knoten gänzlich neu erstellt werden, auf diesen Knoten nicht gleichzeitig in die Seitentabelle aufgenommen werden, sowie daß diese Seiten im folgenden auch auf den anderen Knoten in der Reihenfolge der erstmaligen Erstellung dieser Heapblöcke in die Seitentabelle aufgenommen werden.

Ersteres wird beim dyn. DSM in Kapitel 4 durch die Verwendung des Heap-Mutex gewährleistet, da die Seitentabelle ausschließlich innerhalb dieser, vor gleichzeitiger Ausführung auf mehreren Knoten geschützten Funktionen um gänzlich neue Heapblöcke erweitert wird. Die Seiten aus diesem Heapblöcken werden im weiteren auf den anderen Knoten im Rahmen einer Vervollständigung des DSM-Heap in die Seitentabelle aufgenommen und zwar jeweils unmittelbar nach der Erstellung eines einzelnen Heapblocks. Die Reihenfolge der Seitentabelleneinträge ergibt sich dabei also aus der Reihenfolge der Erstellung dieser Heapblöcke. Diese Reihenfolge wiederum wird von der Reihenfolge der Heapblöcke in den Heapinformationen bestimmt. Und in die Heapinformationen werden die Heapblöcke in der Reihenfolge eingetragen, in welcher sie als gänzlich neue Heapblöcke erstmalig erstellt wurden.

Bzgl. der Reihenfolge der Eintragungen in die Seitentabelle während einer Vervollständigung des DSM-Heap gelten beim effizienten dyn. DSM die gleichen Bedingungen, allerdings ist der Ausschluß der gleichzeitigen Erweiterung der Seitentabelle um gänzlich neu erstellte DSM-Seiten nicht mehr in allen Fällen durch den Heap-Mutex garantiert. Das hat jedoch nicht unmittelbar mit dem DSM-Heap zu tun.

Im Rahmen des Redesigns des DSM-Heap für den effizienten dyn. DSM wurde das DSM-System um die Möglichkeit erweitert, daß auch von anderen Programmteilen, als der Speicherverwaltung des DSM-Heap während der Laufzeit Erweiterungen in der Seitentabelle vorgenommen werden können. Dies wird genutzt, um Synchronisationsobjekte (z.B. Mutex- oder Barrier-Objekte) dynamisch zu erzeugen. Von Bedeutung ist davon hier lediglich, daß dabei für jedes solcher Objekte ein Seiteneintrag in der Seitentabelle vorgenommen werden muß. Das bedeutet, daß die Seitentabelle nun nicht mehr ausschließlich innerhalb der mit dem Heap-Mutex geschützten Funktionen erweitert wird.

Darum wird ein zusätzlicher Mutex (im folgenden PageTable-Mutex genannt) eingeführt, der immer dann benutzt werden muß, wenn die Seitentabelle um neue Seiteneinträge erweitert werden soll, also wenn die Seiten aus einem gänzlich neuen Heapblock in die Seitentabelle eingetragen werden sollen oder wenn ein neues Synchronisationsobjekt erzeugt wird. Im Rahmen einer Heapvervollständigung ist es nicht nötig, den PageTable-Mutex anzuwenden, da die Seitentabelle dabei aus globaler Sicht, d.h. aus der Sicht auf alle Knoten, nicht erweitert wird, sondern lediglich auf einem einzelnen Knoten vervollständigt wird.

5.3.2 Aktualisierung der Seitentabelle

Um die Konsistenz der Seitentabelle auf allen Knoten aufrecht zu erhalten, ist es notwendig, daß sie zum Zeitpunkt ihrer Erweiterung auf dem entsprechenden Knoten alle Einträge enthält, die bisher auf allen Knoten erstellt wurden. Die Seitentabelle muß also zu gegebenen Zeitpunkt auf den einzelnen Knoten aktualisiert werden, d.h. auf dem jeweiligen Knoten müssen die auf den anderen Knoten gemachten Erweiterungen übernommen werden. Jene Erweiterungen der Seitentabelle, welche auf einem Knoten vorgenommen werden, auf dem ein gänzlich neuer Heapblock erstellt wird, werden zwar während der Heapvervollständigung auf den anderen Knoten ebenfalls vorgenommen, es müssen jedoch auch jene Seiteneinträge von den anderen Knoten übernommen werden, welche auf einem Knoten gemacht werden, auf dem ein neues Synchronisationsobjekt erzeugt wird.

Diese Seiteneinträge werden zwar ebenfalls während der Heapvervollständigung erstellt, allerdings wird diese nicht zwangsläufig zum richtigen Zeitpunkt durchgeführt. Die Heapinformationen enthalten für jeden Heapblock die Anzahl der Seiten, die in ihm stehen, sowie den Seitentabellen-Index von dessen erster Seite. Wenn anhand dieser Informationen während einer Heapvervollständigung auf einem Knoten festgestellt wird, daß die Seiten aus zwei im DSM-Heap aufeinanderfolgenden Heapblöcken in der Seitentabelle nicht unmittelbar aufeinander folgen, dann wird davon ausgegangen, daß die zwischen diesen Heapblöcken liegenden Seiteneinträge auf einem anderen Knoten für andere Objekte als Seiten im DSM-Heap, also für Synchronisationsobjekte, erstellt wurden und werden auf diesem Knoten ebenfalls entsprechend erstellt. Gleiches gilt, wenn dabei über die ebenfalls übertragene Gesamtanzahl von DSM-Seiten festgestellt wird, daß in der Seitentabelle auf anderen Knoten noch Einträge hinter denen für den letzten Heapblock vorhanden sind.

Eine Heapvervollständigung und damit eine Aktualisierung der Seitentabelle wird allerdings nur dann durchgeführt, wenn der entsprechende Knoten während eines Seitenempfangs Informationen über neu erstellte Heapblöcke bzw. andere neue Objekte erhält. Bevor auf einem Knoten die Seitentabelle um gänzlich neue Einträge erweitert wird, muß auf diesem Knoten demzufolge ein Zugriffsfehler auf eine DSM-Seite stattfinden, deren aktueller Besitzer ein Knoten ist, auf dem die Seitentabelle bereits alle bisherigen Erweiterungen enthält. Für Erweiterungen bzgl. der Heapblöcke ist dies über den Zugriff auf die Freelist gewährleistet (siehe Abschnitt 5.2.2). Für Erweiterungen bzgl. anderer Objekte gilt dies jedoch nicht, da nicht unbedingt ein entsprechender Seitenzugriff stattfinden muß, bevor die Seitentabelle für ein solches Objekt erweitert wird. Um auch in solchen Fällen aktuelle Heapinformationen zu erhalten, muß darum zu diesem Zeitpunkt ein solcher Seitenzugriff erzwungen werden.

Der Zugriff auf die Synchronisationsobjekte erfolgt über Variablen, die sich innerhalb der statischen DSM-Daten befinden (zumindest bei „sinnvoller“ Anwendung der DSM-Threads, siehe Abschnitt 2.2.2). Innerhalb der Initialisierungsfunktionen für die Synchronisationsobjekte wird schreibend auf das betreffende Objekt zugegriffen. Das bedeutet, daß immer wenn ein solches Objekt erzeugt wird und die Seitentabelle dafür erweitert wird, der betreffende Knoten der Besitzer der Seite wird, auf der sich das Objekt befindet. Da dabei der Schreibzugriff erst stattfindet, nachdem die Seitentabelle erweitert wurde, werden die Informationen über diese Erweiterung im folgenden zu allen Knoten transferiert, die danach Besitzer dieser Seite werden. Soll nun auf einem anderen Knoten die Seitentabelle ebenfalls erweitert werden, muß auf diesem zuvor lediglich auf diese Seite zugegriffen werden, damit die Informationen über die zuerst beschriebene Erweiterung zu ihm übertragen werden und die Seitentabelle entsprechend vervollständigt wird.

Dies müssen aber noch nicht die Informationen über alle bisher gemachten Erweiterungen sein, da mit dieser Seite nur die Informationen unbedingt übertragen werden, die sich auf Erweiterungen für Objekte beziehen, die sich auf dieser Seite befinden. Synchronisationsobjekte können sich jedoch auf mehreren Seiten im statischen DSM-Datenbereich befinden. Um alle diesbzgl. Informationen zu erhalten, muß auf einem Knoten auf alle Seiten in diesem Bereich zugegriffen werden, bevor die Seitentabelle auf diesem Knoten erweitert wird. Dieser Zugriff muß selbstredend erfolgen, nachdem der über den PageTable-Mutex geschützte Bereich betreten wurde.

Optimierung dieser Lösung

Diese Lösung beinhaltet den Nachteil eines nicht unerheblichen Seitentransfers, der bei jeder Erweiterung der Seitentabelle notwendig wird. Dieser Transfer kann in der Anzahl der zu transferierenden Seiten reduziert werden, wenn einige wenige Seiten gefunden werden können, mit deren Transfer auch der Transfer von Heapinformationen garantiert ist, die vollständig in dem Sinne sind, daß sie die Informationen über alle bisher vorgenommenen Erweiterungen der Seitentabelle beinhalten. Dem Idealfall würde dabei entsprechen, wenn eine einzige solche Seite gefunden werden kann. Dies muß eine Seite sein, auf die auf einem Knoten garantiert zugegriffen wird bevor und nachdem auf diesem Knoten die Seitentabelle erweitert wird, wobei dieser Zugriff jeweils innerhalb des über den PageTable-Mutex geschützten Bereichs stattfinden muß.

Eine solche Seite gibt es jedoch nicht, bzw. es finden keine solche Zugriffe auf irgendeine Seite statt. Es kann jedoch eine beliebige Seite ausgewählt werden und deren Transfer über geeignete Zugriffe erzwungen werden. Wann immer also die Seitentabelle auf einem Knoten erweitert werden soll, wird, nachdem der über den PageTable-Mutex geschützte Bereich betreten wurde, auf eine bestimmte DSM-Seite schreibend zugegriffen, um vollständige Heapinformationen zu „holen", dann die Seitentabelle bei Bedarf vervollständigt und dann erst erweitert. Danach wird, allerdings noch bevor der über den PageTable-Mutex geschützte Bereich wieder verlassen wird, dieser Zugriff wiederholt, damit die Informationen über diese Erweiterung im folgenden an die anderen Knoten „verteilt" werden.

Als Seite, auf welche dabei zugegriffen wird, wird die Seite ausgewählt, auf der sich der PageTable-Mutex befindet. Damit kann überflüssiger Seitentransfer so weit wie möglich reduziert werden, da auf diese Seite sowieso zugegriffen wird, wenn der PageTable-Mutex angefordert wird. Da dies nur ein Lesezugriff ist, wird über diesen Zugriff der vollständige Transfer der Heapinformationen allerdings noch nicht garantiert, so daß das beschriebene Vorgehen notwendig ist.

5.4 Verwaltung des Speichers in den Heapblöcken

Aus der umgekehrten Reihenfolge, in der die DSM-Heapblöcke nun im Speicher stehen, ergeben sich keine Notwendigkeiten zu Änderungen bei der Verwaltung des Speichers in den Heapblöcken, d.h. es kann wiederum die Verwaltungsmethode von Kernighan/Ritchie (siehe Abschnitt 3.3) verwendet werden und es gelten, von der weggefallenen Zeigerübersetzung abgesehen, die in Abschnitt 4.3 gemachten Anmerkungen weiterhin.

Der in Abschnitt 3.3.2 beschriebene Nachteil bei der Verwendung dieser Methode im DSM-Heap, nämlich der eventuell recht umfangreich auftretende Seitentransfer beim Durchsuchen der Freelist, ist hinsichtlich der Effizienz mit dem Anspruch dieser Arbeit jedoch nicht vereinbar. Jede DSM-Seite, auf der sich ein Element der Freelist befindet, muß zu einem Knoten, auf dem die Freelist durchsucht wird, transferiert werden, es sei denn, dieser Knoten ist gerade zufällig der Besitzer dieser Seite. D.h. die Anzahl der Seiten, die dabei transferiert werden müssen und damit der entstehende Aufwand, hängt demzufolge davon ab, auf wie viele Seiten die Elemente der Freelist verteilt sind. Dieser Effizienznachteil fällt also von Anwendung zu Anwendung unterschiedlich stark ins Gewicht, es ist jedoch durchaus nicht ungewöhnlich, daß die Freelist eine hohe Anzahl von Elementen enthält, die auf verschiedenen Seiten stehen.

Dieser Aufwand beim Durchsuchen der Freelist kann reduziert werden, indem die Verteilung ihrer Elemente auf den DSM-Seiten eingeschränkt wird und so erreicht wird, daß weniger Seiten übertragen werden müssen. Dazu müssen deren Elemente von diesen Seiten entfernt werden, die Freelist also ausgelagert werden in dem Sinne, daß ihre Elemente nicht mehr an der gleichen Stelle im DSM-Heap stehen wie der jeweils verwaltete Speicherabschnitt selbst.

5.4.1 Auslagerung der Freelist

Die Elemente der Freelist bestehen aus einem Kopf und dem freien Speicherabschnitt selbst (siehe Abschnitt 3.3). Sie werden in der Art umgestaltet, daß sie den freien Speicherabschnitt nicht mehr beinhalten, sondern lediglich einen Zeiger auf ihn. Damit können sie an beliebigen anderen Stellen im Speicher erstellt werden als nur bei den verwalteten Speicherabschnitten. Auf die Frage, an welchen Stellen dies geschieht, wird unten eingegangen.

Wie in Abbildung 5.4 zu sehen ist, wird quasi eine Kopie der Freelist angelegt, so daß nicht mehr auf die Speicherstellen selbst zugegriffen werden muß, an denen sich die verwalteten Abschnitte befinden, wenn sie durchsucht wird. Ansonsten bleiben der Aufbau und die Funktionsweise der Freelist sowie der Funktionen zu ihrer Verwaltung erhalten, d.h. es gelten die dazu in Abschnitt 3.3 gemachten Ausführungen weiterhin.

Alle notwendigen Informationen zur Verwaltung der freien Speicherabschnitte werden nunmehr extern abgespeichert, d.h. sie stehen an anderer Stelle als die freien Abschnitte selbst. Trotzdem ist es notwendig, die bisherige Struktur der freien und vergebenen Speicherabschnitte beizubehalten, d.h. sie haben weiterhin einen Kopf, in welchem die Größe des jeweiligen Speicherabschnitts und dessen Adresse (mit Kopf) abgespeichert wird. Die Speicherung der Größe ist zwingend erforderlich, da dieser Wert benötigt wird, um einen Speicherabschnitt, den der Anwender wieder freigibt, korrekt in die Freelist aufnehmen zu können.

Auf die Speicherung der Adresse des Speicherabschnitts könnte verzichtet werden, da sie stattdessen auch berechnet werden könnte, wann immer sie benötigt wird. Sie wird hier trotzdem im Kopf abgespeichert, da dieser sowieso in einer Größe angelegt werden muß, die es erlaubt, beide Werte in ihm abzulegen, d.h. für die Speicherung der Adresse wird keinerlei zusätzlicher Speicherplatz belegt. So kann der (wenn auch geringe) Aufwand für deren Berechnung gespart werden.

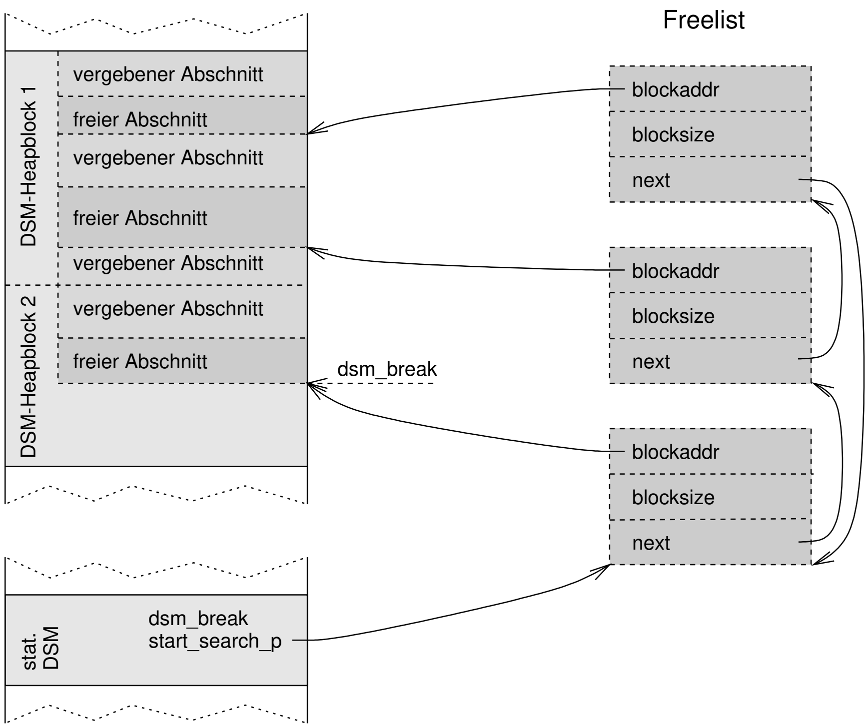

Abbildung 5.4: Auslagerung der Freelist

Die an den Anwender vergebenen Speicherabschnitte müssen im Speicher so ausgerichtet sein, daß sie Objekte eines jeden Datentyps aufnehmen können. D.h. die Anfangsadressen der an den Anwender gelieferten Abschnitte und die Größe der jeweiligen Abschnitte müssen ein Vielfaches eines Wertes sein, der so groß ist wie der bzgl. der Ausrichtung eingeschränkteste Datentyp. Für den DSM-Heap werden für diesen Wert 16 Byte genommen. Aufgrund dieser Bedingungen muß auch der Kopf der Speicherabschnitte 16 Byte groß sein und kann demzufolge ohne weiteres beide genannten Werte aufnehmen.

Eine Alternative zu der Methode, die benötigte Größenangabe in einem Kopf in den vergebenen Speicherabschnitten selbst abzulegen, die demzufolge um jeweils 16 Byte größer angelegt werden müssen als für die Bedienung der Anwenderforderung nötig ist, wäre z.B. die Verwendung einer externen Liste ähnlich der Freelist, in der die Adressen und Größen aller Abschnitte gespeichert werden, die aus dem DSM-Heap an den Anwender vergeben wurden. Dem spricht jedoch der damit verbundene nicht unerhebliche Aufwand entgegen. Es ist keineswegs unwahrscheinlich, daß eine recht hohe Anzahl von Speicherabschnitten an den Anwender vergeben werden, diese Liste also recht lang werden würde. Die Liste müßte bei jeder Vergabe und Rücknahme eines Speicherabschnitts nach der Einfügestelle, da die Liste nach den Adressen der Speicherabschnitte geordnet sein sollte, bzw. den Angaben zu diesem Speicherabschnitt durchsucht werden. Der damit verbundene Aufwand kann hinsichtlich des Ziels dieser Arbeit nicht akzeptiert werden.

Plazierung der Freelist

Die Elemente der Freelist müssen auf DSM-Seiten stehen, da es sich dabei um gemeinsame Daten handelt. Aus denselben Gründen wie Sie für die Heapinformationen aufgeführt wurden (siehe Abschnitt 5.2.1) können sie jedoch weder im statischen Datenbereich des DSM noch im DSM-Heap angelegt werden. Desweiteren ist es sinnvoll, so viele Freelist-Elemente wie möglich auf möglichst wenigen DSM-Seiten unterzubringen, da so der notwendige Seitentransfer beim Durchsuchen der Freelist so gering wie möglich gehalten wird, d.h. es sollten möglichst wenige (am besten gar keine) anderen Daten auf den Seiten stehen, auf denen sich die Freelist-Elemente befinden.

Darum werden für die Freelist zusätzliche, eigenständige DSM-Seiten angelegt, und zwar weder im statischen DSM noch im DSM-Heap, sondern in extra Blöcken (im folgenden Verwaltungsblöcke genannt), in denen sich keinerlei andere DSM-Seiten und keinerlei andere Daten befinden.

Die einzelnen Freelistelemente beinhalten je zwei Zeiger und einen Integerwert (siehe Abbildung 5.4). Wenn dafür (wie unter UNIX nicht unüblich) insgesamt 12 Byte Speicherplatz notwendig sind und eine DSM-Seite 4096 Bytes umfaßt, so passen auf eine DSM-Seite 341 Freelist-Elemente. Das bedeutet, daß für das Durchsuchen von 341 freien Speicherabschnitten lediglich der Transfer einer DSM-Seite notwendig wird. Wird die Freelist nicht ausgelagert, so wird dafür im schlimmsten Fall, nämlich wenn sich jeder dieser Speicherabschnitte auf einer anderen Seite befindet, der Transfer von 341 Seiten notwendig.

5.5 Verwaltungsblöcke

Die DSM-Seiten, auf denen die Freelist-Elemente stehen, werden in Verwaltungsblöcken zusammengefaßt. Für diese Verwaltungsblöcke gelten uneingeschränkt die gleichen Anforderungen bzgl. der Plazierung im virtuellen Speicher wie für die Heapblöcke, und sie werden demzufolge auf die gleiche Art und Weise erstellt (siehe Abschnitt 5.1).

Wann immer ein neues Freelist-Element erzeugt werden soll, wird überprüft, ob in den bisher erstellten Verwaltungsblöcken noch genug Platz dafür frei ist und wenn dies nicht der Fall ist, ein neuer Verwaltungsblock erstellt. Die Verwaltungsblöcke werden dabei mit einer gewissen Standardgröße (einige DSM-Seiten) zwischen den Heapblöcken angelegt (siehe Abbildung 5.5).

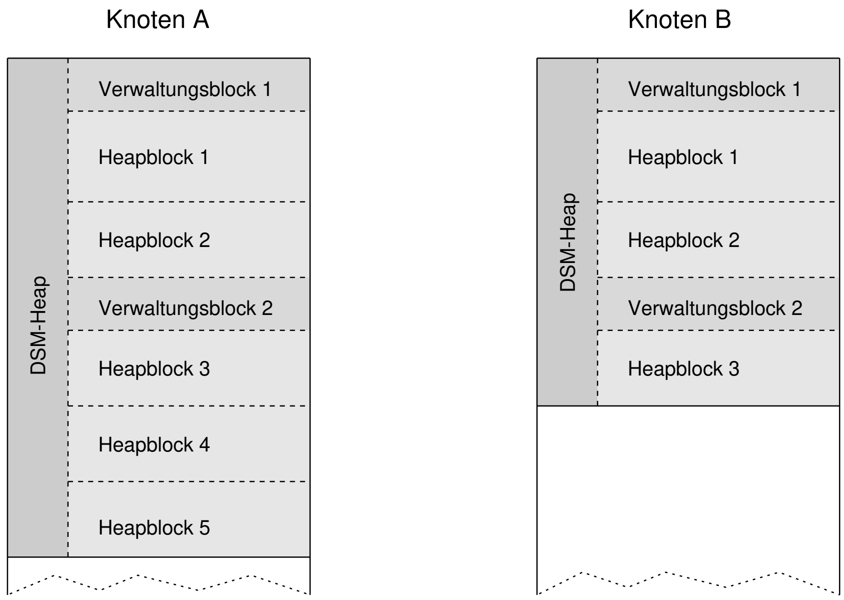

Abbildung 5.5: Die Verwaltungsblöcke im DSM-Heap

Der Unterschied zwischen den beiden Arten von Blöcken ergibt sich einzig und allein aus der Art der Daten, die in ihm abgelegt werden. Für die Verwaltung der Heapblöcke und aller damit zusammenhängender Aspekte wie die Vervollständigung des DSM-Heap, die Heapinformationen, die Seitentabelle usw. (siehe Abschnitte 5.2 und 5.3) ist die Art der Daten, die sich in den Blöcken befinden, jedoch nicht von Relevanz, d.h. bei keinem der diesbzgl. Programmteile des DSM-Systems spielt es eine Rolle, ob es sich bei den behandelten Blöcken um Heap- oder Verwaltungsblöcke handelt. Darum werden für die zusätzliche Behandlung der Verwaltungsblöcke an keinem dieser Programmteile irgendwelche Änderungen notwendig.

Lediglich bei der Verwaltung des Speichers in den Blöcken muß zwischen den Heap- und Verwaltungsblöcken unterschieden werden, denn die Verwaltungsblöcke werden nicht in den für die Heapblöcke verwendeten Speicherverwaltungsmechanismus aufgenommen.

5.5.1 Verwaltung des Speichers in den Verwaltungsblöcken

Der Speicher in den Verwaltungsblöcken muß geeignet verwaltet werden, d.h. es müssen Abschnitte aus ihnen zur Verfügung gestellt und wieder zurückgenommen werden, wenn Freelist-Elemente erzeugt bzw. wieder gelöscht werden sollen. Diese Verwaltung ist mit recht einfachen Mitteln realisierbar, da all diese Speicherabschnitte die gleiche Größe haben, nämlich die eines Freelist-Elements.

Die Freelist-Elemente werden jeweils unmittelbar hintereinander in den Verwaltungsblöcken erstellt und diese so in Abschnitte von der Größe der Freelist-Elemente unterteilt. Wird ein Element wieder aus der Freelist gelöscht, so wird der entsprechende Verwaltungsblock-Abschnitt in eine zusätzliche Liste eingetragen und im folgenden für neue Freelist-Elemente wiederverwendet und in dem Fall wieder aus der Liste entfernt. Bevor demzufolge tatsächlich ein neuer Verwaltungsblock-Abschnitt erstellt wird, wird diese Liste durchsucht, ob noch ein freier Abschnitt in einem Verwaltungsblock vorhanden ist. Ist diese Liste leer und der Speicher in dem zuletzt erstellten Verwaltungsblock vollständig aufgebraucht, wird ein neuer Verwaltungsblock angelegt. In Abbildung 5.6 ist dieser Mechanismus dargestellt.

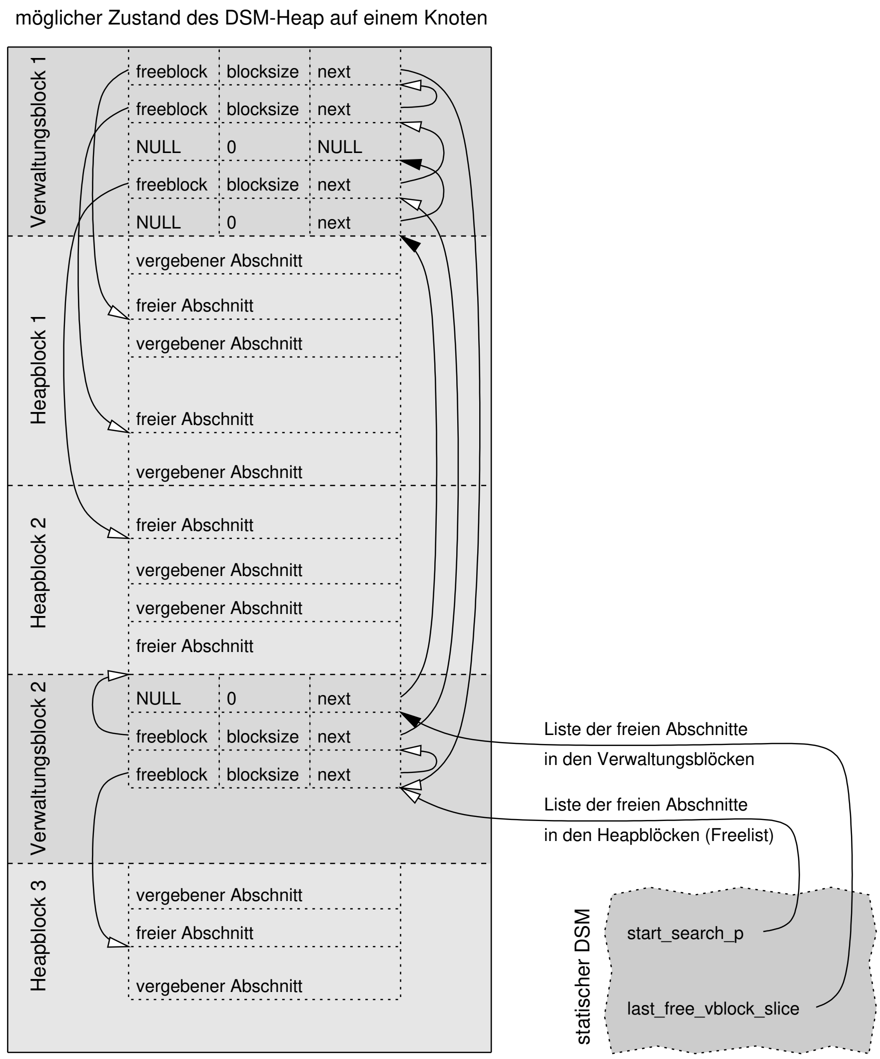

Abbildung 5.6: Die Listen der Speicherverwaltung des DSM

5.6 Vor- und Nachteile des effizienten dyn. DSM

Aus dem Umstand, daß die DSM-Zeiger bei dieser Lösung nicht mehr übersetzt werden müssen, ergibt sich in bezug auf die Effizienz neben den Vorteilen, daß der Rechenaufwand für die Übersetzung sowie der damit zusammenhängende Verwaltungsaufwand, z.B. für die Zeigertabelle, wegfallen, u.a. noch der Vorteil, daß die Nachrichten, in denen die DSM-Seiten übertragen werden, nicht mehr durch die Einträge der Zeigertabelle in ihrer Größe zunehmen. Aber auch in anderer Hinsicht ergeben sich aus dem Wegfall der Zeigerübersetzung Vorteile. Die in Abschnitt 4.5 aufgeführten Einschränkungen bei der Benutzung von DSM-Zeigern durch den Anwender sowie das diesbzgl. Risiko von Anwenderfehlern fallen vollständig weg, da die Ursache dafür vollständig wegfällt.

Der Wegfall der Zeigerübersetzung kommt der Effizienz auch bei der Auslagerung der Freelist zugute, da für die Verwaltung der freien Speicherabschnitte im DSM-Heap nun mehr als doppelt soviele Zeiger verwendet werden müssen als bei der ursprünglichen Version der Speicherverwaltungsmethode nach Kernighan/Ritchie und all diese Zeiger anderenfalls, d.h. falls die Übersetzung der Zeiger noch notwendig sein würde, bei jedem Transfer der entsprechenden Seiten übersetzt werden müßten.

Aufgrund dieser Tatsache könnten die Operationen zur Bearbeitung der Freelist nun auch über geeignete Methoden weiter optimiert werden, da dabei eine eventuelle Verwendung von weiteren Zeigern keine negativen Auswirkungen mehr auf die Effizienz hätte. Auf die Vorstellung solcher Methoden, z.B. zur schnelleren Auffindung einer bestimmten Stelle in einer Liste, wird hier jedoch verzichtet, da von diesen bereits zahlreiche bekannt sind und sie mit der eigentlichen Problematik einer effizienten Speicherverwaltung in einem verteilten System eher weniger zu tun haben. Auch muß gesagt werden, daß die Verwendung solcher Methoden für die Freelist nur geringen Einfluß auf die Gesamtperformance des DSM-Systems hätte, da in einem verteilten System diesbzgl. anderen Aspekten wie z.B. dem Nachrichtentransfer weitaus größere Bedeutung zukommt als der Optimierung von Berechnungen die nur lokal, d.h. jeweils nur auf einem einzigen Knoten, ausgeführt werden.

Ein gewisser Nachteil dieser Lösung sind die in Abschnitt 5.1.2 aufgeführten Risiken, die aus der Anwendung von `mmap` unter fester Vorgabe der Adresse, an der die Speicherbereiche für die Heapblöcke eingeblendet werden sollen, resultieren. In bezug auf die Probleme, die entstehen können, wenn der Anwender der DSM-Threads `mmap` in der gleichen Weise benutzt, muß einschränkend zu den dazu in Abschnitt 5.1.2 gemachten Aussagen angemerkt werden, daß es zwar nicht unbedingt notwendig ist, dem Anwender dies rigoros zu verbieten. Dann muß er aber über die Plazierung des DSM-Heap im virtuellen Speicher informiert werden, und ihm muß die Möglichkeit gegeben werden, während der Laufzeit feststellen zu können, welchen Speicherbereich der DSM-Heap aktuell einnimmt. Dies kann auf einfache Weise realisiert werden, indem die Anwenderschnittstelle zum DSM-Heap (siehe Anhang A) um eine Funktion erweitert wird, welche die Untergrenze des zuletzt erstellten Heap- bzw. Verwaltungsblocks liefert. Damit kann der Anwender jedoch lediglich verhindern, daß er selbst einen Speicherbereich einblendet, in dem sich bereits der DSM-Heap befindet. Der umgekehrte Fall, daß der DSM-Heap in einen vom Anwender bereits eingeblendeten Speicherbereich hineinwächst, kann auf diese Weise nicht verhindert werden, d.h. mit diesem Risiko muß der Anwender leben, wenn er auf die Verwendung von `mmap` in dieser Weise nicht verzichten kann, und die Adresse des Bereichs, den er so einblenden will, entsprechend auswählen. Dies ist jedoch ein tragbarer Nachteil, da eine derartige Verwendung von `mmap` eher ungewöhnlich und im allgemeinen ohnehin nicht empfehlenswert ist.

5.7 Zusammenfassender Vergleich der Lösungen

In Abbildung 5.7 werden die Hauptmerkmale des semi-dynamischen DSM aus Kapitel 3, des DSM mit Zeigerübersetzung aus Kapitel 4 sowie des effizienten DSM aus diesem Kapitel einander gegenübergestellt.

Version Merkmale	semi-dynamischer DSM	dynamischer DSM mit Zeigerübersetzung	dynamischer DSM ohne Zeigerübersetzung und mit ausgelagerter Freelist
Funktionalität	feste Größe des DSM-Heap -> nicht echt dynamisch DSM-Zeiger sind beliebig nutzbar	DSM-Heap zur Laufzeit beliebig vergrößerbar -> echt dynamisch Einschränkungen bzgl. der Nutzung von DSM-Zeigern wegen deren Registrierung	DSM-Heap zur Laufzeit beliebig vergrößerbar -> echt dynamisch DSM-Zeiger sind beliebig nutzbar
Effizienz	geringer Aufwand bei der Erstellung /Verwaltung des DSM-Heap Umfang des Seitentransfers beim Durchsuchen der Freelist ist abhängig von der Verteilung der freien Speicherabschnitte im DSM-Heap	etwas höherer Aufwand bei der Erstellung/Verwaltung der DSM-Heapblöcke Umfang des Seitentransfers beim Durchsuchen der Freelist ist abhängig von der Verteilung der freien Speicherabschnitte im DSM-Heap Nachteil: Aufwand zur notwendigen Zeigerübersetzung	etwas höherer Aufwand bei der Erstellung/Verwaltung der DSM-Heapblöcke reduzierter Seitentransfer beim Durchsuchen der Freelist, da dessen Umfang unabhängig von der Verteilung der Speicherabschnitte ist
besondere Vorteile	effizienteste Version wenn nur sehr wenig dyn. DSM benötigt wird (1-2 Seiten)	Zeigerübersetzung kommt einer eventuellen Erweiterung der DSM-Theads für heterogene verteilte Systeme entgegen	im allgemeinen Anwendungsfall die effizienteste Version am breitesten einsetzbar, da keinerlei Einschränkungen bei der Anwendung (abgesehen von mmap -Risiken)

Abbildung 5.7: Vergleich der verschiedenen Versionen des DSM-Heap

Kapitel 6

Performancetests

Anhand von Performancetests sollen nun abschließend die Auswirkungen der in Kapitel 5 vorgestellten Maßnahmen zur Effizienzsteigerung bei der Verwaltung des dynamischen DSM auf die Ausführungsgeschwindigkeit von DSM-Anwendungsprogrammen überprüft werden. Dafür werden zwei Programme aus der SPLASH-2-Benchmark-Suite [35, 36] genutzt. Es ist jedoch aufgrund der Art und Weise wie der DSM-Heap innerhalb dieser Programme genutzt wird zu erwarten, daß die Geschwindigkeitsunterschiede bei der Ausführung dieser Programme mit den unterschiedlichen Versionen des DSM-Heap nur wenig erheblich sind. Darum wird zusätzlich ein Programm ausgeführt, in welchem der DSM-Heap so genutzt wird, daß sich deutliche Geschwindigkeitsunterschiede zeigen müßten, sofern die Änderungen zur Verbesserung der Effizienz des DSM-Heap die gewünschte Wirkung haben.

Durchgeführt wurden die Performancetests auf einem PC-Cluster bestehend aus Rechnern mit AMD Athlon 550 Mhz Prozessoren und 128 MB Hauptspeicher, die über Fast Ethernet und Myrinet verbundenen sind.

6.1 FFT und LU

Von den SPLASH-2-Benchmarks werden die Programme *FFT* und *LU* benutzt [37]. In FFT („*Fast Fourier Transform*“) wird eine schnelle Fourier Transformation ausgeführt, wobei eine komplexe 1D-Version des „radix $\sqrt{n}$ sixstep FFT“ Algorithmus [38] zur Anwendung kommt. In LU wird eine LU-Zerlegung einer dichten $n \times n$ Matrix in ein $N \times N$ Array mit $B \times B$ Blöcken mit $n = NB$ durchgeführt. Als Problemgrößen wurden hier für FFT 4096 Datenpunkte (n) sowie für LU eine Matrixgröße von 512×512 (n) und eine Blockgröße von 32×32 (B) gewählt. Beide Programme wurden auf 4 Rechnern ausgeführt. In Abbildung 6.1 sind die Ausführungsgeschwindigkeiten beider Programme unter Benutzung des DSM-Heap aus Kapitel 4 sowie des effizienten DSM-Heap aus Kapitel 5 dargestellt.

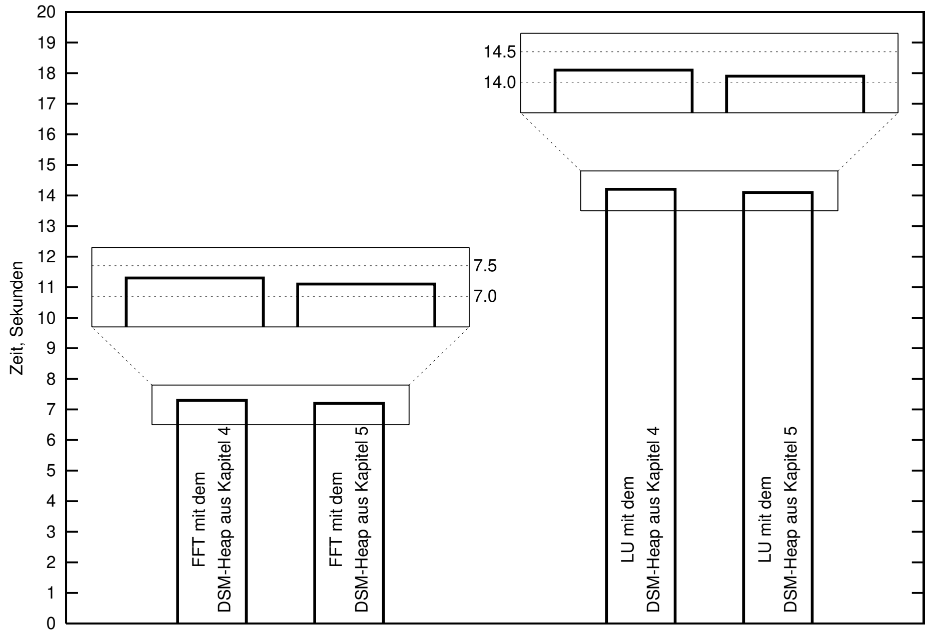

Abbildung 6.1: FFT und LU mit den verschiedenen Versionen des dyn. DSM

Sowohl bei FFT als auch bei LU ergibt sich nur eine sehr geringfügige Erhöhung der Ausführungsgeschwindigkeit bei der Verwendung der beiden verschiedenen Versionen des dynamischen DSM. Dies hat bei beiden Programmen die gleichen Gründe, welche in der Verwendungsweise des DSM-Heap in der jeweiligen Applikation liegen.

Innerhalb von FFT und LU werden zwar recht große Datenmengen im DSM-Heap abgelegt, diese Daten bestehen jedoch bei beiden Programmen hauptsächlich aus einigen wenigen, großen Arrays. Darum wird in FFT und LU nur einige wenige Male Speicher aus dem DSM-Heap angefordert. In FFT wird `dsm_malloc` (siehe Anhang A) lediglich 7 mal und in LU 13 mal aufgerufen, `dsm_free` jeweils keinmal. Dementsprechend wenige DSM-Zeiger werden in beiden Programmen benutzt (in FFT 16 Stück, in LU 28 Stück) und dementsprechend wenige Elemente enthält jeweils die Freelist (zu keinem Zeitpunkt mehr als 2 Stück). Das bedeutet, daß beim DSM-Heap mit Zeigerübersetzung (Kapitel 4) jeweils nur wenige Zeigerübersetzungen stattfinden (bei FFT ca. 1250, bei LU ca. 2100), und das bedeutet weiterhin, daß ein Durchsuchen der Freelist, was ohnehin nur 7 bzw. 13 mal stattfindet, den Transfer von maximal 2 Seiten nach sich zieht. Dieser minimale Mehraufwand schlägt sich in den Ausführungszeiten nicht nieder.

Die Haupteffizienznachteile der in Kapitel 4 gezeigten Lösung für einen dynamischen DSM kommen also weder bei FFT noch bei LU wesentlich zum

Tragen. Daß die Ausführung durch die Verwendung des effizienteren dyn. DSM aus Kapitel 5 trotzdem etwas beschleunigt wird, wenn auch nur vernachlässigbar wenig, liegt hauptsächlich daran, daß die Aktionen, die beim Transferieren einer jeden Seite ausgeführt werden, durch den Wegfall der Zeigerübersetzung und die Optimierung des Transfers der Heapinformationen vereinfacht wurden und im Umfang abgenommen haben.

6.2 Ein spezielles Testprogramm

Es sollen nun die Ausführungsgeschwindigkeiten eines Programms unter Verwendung der unterschiedlichen Versionen des DSM-Heap verglichen werden, in welchem der DSM-Heap in dem Sinne sehr intensiv genutzt wird, daß sehr viele kleine Abschnitte aus ihm angefordert werden.

Die Hauptdatenstruktur im Testprogramm ist eine Liste, die von jedem gestarteten Thread um genau 84 Elemente erweitert wird, wobei für jedes Element `dsm_malloc` aufgerufen wird. Ein Listenelement enthält 7 DSM-Zeiger (welche bei der Verwendung des DSM-Heap aus Kapitel 4 in der Zeigertabelle registriert werden) sowie einen Integerwert. Zusätzlich wird von jedem Thread genau `n+1` Elementen eines Arrays von Integerzeigern (welches im stat. DSM-Datenbereich angelegt wurde) über `dsm_malloc` jeweils ein 4 Byte großes Objekt aus dem DSM-Heap zugewiesen, wobei `n` jeweils die Anzahl der Threads ist, von denen dies bisher getan wurde. Danach werden letztgenannte Speicherbereiche über `dsm_free` wieder freigegeben. Diese Abschnitte werden in die Freelist aufgenommen und werden, da sie für die Elemente der Liste zu klein sind, erst dann wiedervergeben, wenn der nachfolgende Thread die Speicheranforderungen für die Arrayelemente vornimmt. Von diesen `n+1` Speicheranforderungen werden also jeweils `n` aus der Freelist bedient und pro Thread dafür nur genau einmal Speicher im DSM-Heap „völlig neu“ belegt.

So wird erreicht, daß die Freelist pro Thread um genau ein Element erweitert wird, und daß sich alle Elemente der Freelist auf unterschiedlichen Seiten befinden, denn im Testprogramm wird von jedem gestarteten Thread genau eine Seite (4096 Byte) des DSM-Heap verbraucht. Die Elemente der Liste sind 32 Byte groß. Zuzüglich der 16 Byte für den Kopf eines jeden Speicherabschnitts im DSM-Heap (siehe Abschnitt 5.4.1) sind demzufolge für jedes Listenelement 48 Byte des DSM-Heap nötig, wobei auf jedem Thread für ein einzelnes Element 32 zusätzliche Byte angefordert werden. Insgesamt sind pro Thread 4064 Byte ($84 * 48$ Byte + 32 Byte) des DSM-Heap für die Liste nötig. Pro Thread werden für den jeweils einzigen „völlig neu“ erstellten Speicherabschnitt bei den Speicheranforderungen für die Arrayelemente 32 Byte aus dem DSM-Heap verbraucht (16 Byte statt der benötigten 4 Byte wegen der Ausrichtung + 16 Byte für den Kopf). Insgesamt macht dies 4096 (4064 + 32) Byte des DSM-Heap pro Thread.

Pro Thread wird `dsm_malloc` 84 mal für die Listenelemente, zuzüglich `n+1` mal für die Arrayelemente (wobei `n` wie oben die Anzahl der bisher bis zu dieser Stelle ausgeführten Threads ist) und `dsm_free` `n+1` mal aufgerufen. Bei der Verwendung des DSM-Heap mit Zeigerübersetzung werden pro Thread 588 ($84 * 7$) Zeiger der Liste, zuzüglich `n+1` Zeiger des Arrays, zuzüglich je einem Zeiger der Freelist in der Zeigertabelle registriert, wobei die Zeiger des Arrays jeweils wieder freigegeben werden.

Zuerst werden von den einzelnen Threads die Speicherabschnitte für die Arrayelemente angefordert und dabei eine neue Seite im DSM-Heap belegt, dann diese Abschnitte wieder freigegeben, wonach sich auf der neuen Seite (und allen zuvor im DSM-Heap belegten Seiten) genau ein Freelist-Element befindet und dann die Elemente der Liste erstellt. Diese Aktionen sind über einen Mutex vor gleichzeitiger Ausführung durch mehrere Threads geschützt. Nachdem dieser Mutex im Anschluß wieder freigegeben wurde, wird nocheinmal lesend auf alle bisher erstellten DSM-Heap-Seiten und dann schreibend auf alle von dem jeweiligen Thread selbst erstellten Elemente der Liste (die sämtlich auf der zuletzt erstellten Seite liegen) zugegriffen.

Ausgeführt wird das Testprogramm auf 1, 2, 4 und 8 Rechnern und zwar jeweils mit einer Anzahl von 8, 16, 24, 32, 40 und 48 Threads. Getestet wird eine Version des DSM-Heap mit Zeigerübersetzung und ohne ausgelagerte Freelist (Kapitel 4), eine Version ohne Zeigerübersetzung und ohne ausgelagerte Freelist (Kapitel 5 bis Abschnitt 5.4.1) sowie eine Version ohne Zeigerübersetzung und mit ausgelagerter Freelist (Kapitel 5 einschließlich der Abschnitte 5.4.1 und 5.5). In Abbildung 6.2 sind die Ergebnisse dieser Tests dargestellt.

Gegenüber den Programmen FFT und LU sind bei dem speziellen Testprogramm recht deutliche Unterschiede bei den Ausführungszeiten mit den unterschiedlichen Versionen des DSM-Heap zu erkennen. So ergibt sich ein Geschwindigkeitsgewinn vom DSM-Heap ohne ausgelagerte Freelist aus Kapitel 5 zum DSM-Heap aus Kapitel 4 von durchschnittlich 4%, vom DSM-Heap mit ausgelagerter Freelist aus Kapitel 5 zum DSM-Heap ohne ausgelagerte Freelist aus Kapitel 5 von 18.8% und vom DSM-Heap mit ausgelagerter Freelist aus Kapitel 5 zum DSM-Heap aus Kapitel 4 von 23.5%. Dabei tritt bei höheren Anzahlen von ausgeführten Threads ein höherer Gewinn an Ausführungsgeschwindigkeit auf. Die entsprechenden Werte betragen bei einer Anzahl von 48 Threads 7.7%, 30.9% und 35.4%.

Daß die Ausführungszeit des Testprogramms mit der Anzahl der Threads unabhängig von der verwendeten Version des DSM-Heap im Gegensatz zur Anzahl der Threads nicht linear sondern schneller zunimmt, liegt hier ausschließlich an dem Aufbau des Testprogramms. Im weiteren wird zu sehen sein, daß die Ausführungsgeschwindigkeit des Testprogramms wesentlich von der Anzahl der Seitenzugriffsfehler bestimmt wird. Diese Anzahl steigt bei dem hier verwendeten Testprogramm schneller als die Anzahl der Threads.

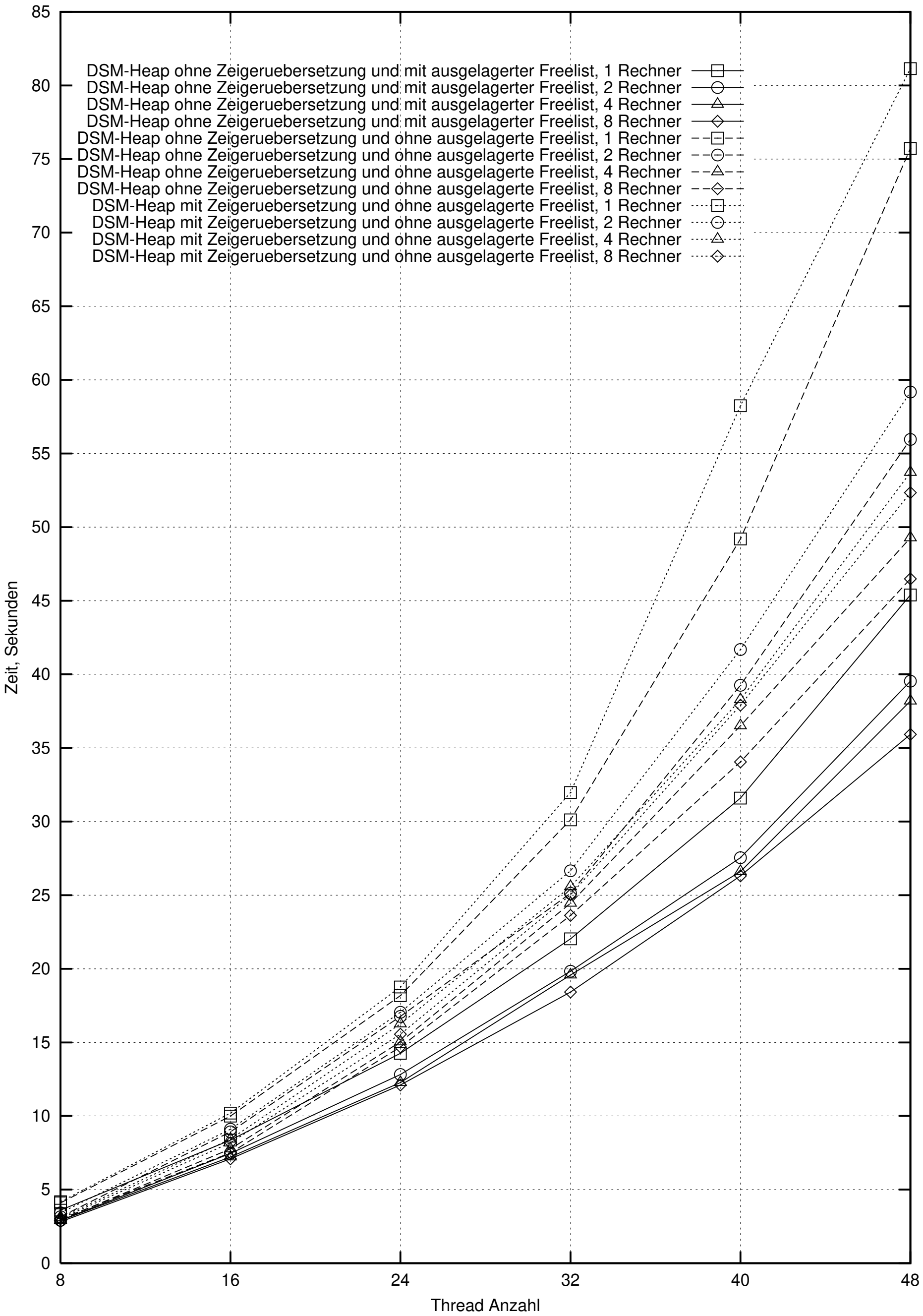

Abbildung 6.2: Ausführungszeiten des speziellen Testprogramms

Verantwortlich für den nichtlinearen Anstieg dieser Zugriffsfehler ist die Tatsache, daß pro Thread an mehreren Stellen im Testprogramm auf sämtliche Seiten im DSM-Heap zugegriffen wird, die bisher erstellt worden sind.

Die Anzahl der bereits erstellten Seiten hängt dabei von der Anzahl der Threads ab, die bisher den Mutex-Bereich betreten haben, da darin von jedem Thread genau eine Seite erstellt wird. Von Thread 5 wird also an jeder der Stellen auf fünf DSM-Seiten zugegriffen. Betritt der fünfte Thread den Mutex-Bereich, wird erst auf die vier bereits erstellten Seiten zugegriffen, dann die fünfte Seite erstellt (während der Speicheranforderungen für fünf Arrayelemente), und dann noch zweimal auf die fünf Seiten zugegriffen (während der Wiederfreigabe des Speichers für die fünf Arrayelemente und der Erweiterung der Liste). Nach dem Verlassen des Mutex-Bereichs wird auf bestimmte Elemente der Liste zugegriffen (auf je ein Element auf den ersten vier Seiten und alle auf der fünften Seite) und so nochmals auf die fünf Seiten zugegriffen. Thread 5 greift also viermal auf fünf Seiten zu, Thread 6 viermal auf sechs Seiten usw. Bei acht gestarteten Threads beträgt die Anzahl dieser Zugriffe demzufolge $4 * 1 + 2 + \ldots + 8 = 144$. Der nichtlineare Anstieg dieser Anzahl ist in Abbildung 6.3 zu erkennen, in der deren Anzahl für eine Thread-Anzahl von 8 bis 48 dargestellt ist.

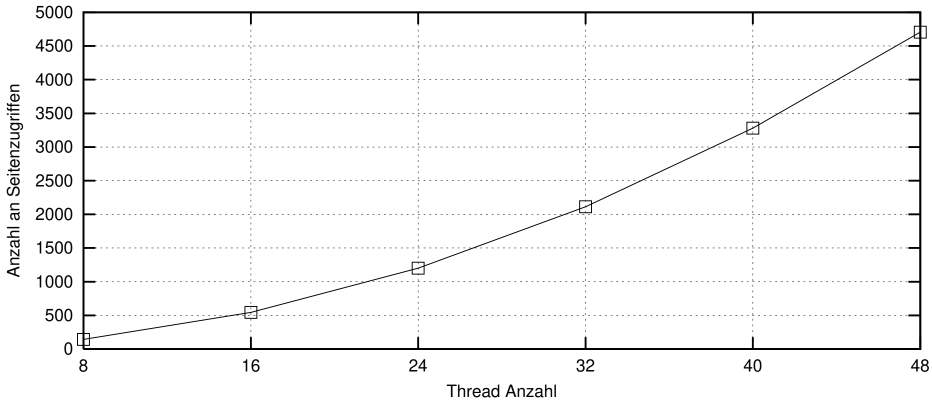

Abbildung 6.3: Anstieg der Seitenzugriffe beim speziellen Testprogramm

All dies sind Zugriffe auf Seiten, deren Besitzer der entsprechende Knoten zum Zugriffszeitpunkt möglicherweise gerade nicht ist, da auch außerhalb des Mutex-Bereichs auf die entsprechenden Seiten zugegriffen wird, d.h. Zugriffe auf diese Seiten auf mehreren Knoten gleichzeitig stattfinden können. Aus jedem dieser Zugriffe kann darum ein Seitenzugriffsfehler resultieren. Auch wenn dies nicht immer der Fall ist, verursachen diese Seitenzugriffe bzw. potentiellen Seitenzugriffsfehler den nichtlinearen Anstieg der Gesamtanzahl der Seitenzugriffsfehler und damit der Ausführungsgeschwindigkeit. Pro Thread finden zwar noch weitere Zugriffe auf Objekte im stat. und dyn. DSM statt, deren Anzahl ist jedoch auf allen Threads gleich, und sie tragen darum nicht zu dem nichtlinearen Anstieg der Zugriffsfehler bei. Bei den getesteten Versionen des DSM-Heap ohne ausgelagerte Freelist verläuft der Anstieg der Anzahl der Seitenzugriffsfehler allerdings noch steiler als bei dem DSM-Heap mit ausgelagerter Freelist. Dies begründet sich damit, daß die Anzahl der Aufrufe von `dsm_malloc` (1 + 2 + ... + Threadanzahl + Threadanzahl $*$ 84) und `dsm_free` (1 + 2 + ... + Threadanzahl) im Testprogramm schneller ansteigt als die Anzahl der Threads (was allerdings auch bei dem DSM-Heap mit ausgelagerter Freelist einen zusätzlichen Anteil zum nichtlinearen Anstieg der Ausführungsgeschwindigkeit beiträgt), und daß bei nicht ausgelagerter Freelist bei jedem dieser Aufrufe auf sämtliche bisher erstellten DSM-Heap-Seiten (wiederum 1 + 2 + ... + Threadanzahl) zugegriffen wird.

Auffällig ist, daß die Auslagerung der Freelist einen sehr viel größeren Gewinn an Effizienz mit sich bringt als der Wegfall der Zeigerübersetzung. Dies mag umso erstaunlicher erscheinen, wenn man die recht hohe Anzahl von Zeigerübersetzungen betrachtet, die jeweils auf allen Knoten insgesamt durchgeführt werden. In Abbildung 6.4 ist diese Anzahl dargestellt, wobei die Kurven für die unterschiedliche Anzahl von Rechnern zu einer zusammengefaßt wurden, da sich die jeweiligen Beträge dabei nur in sehr geringem Maße unterscheiden.

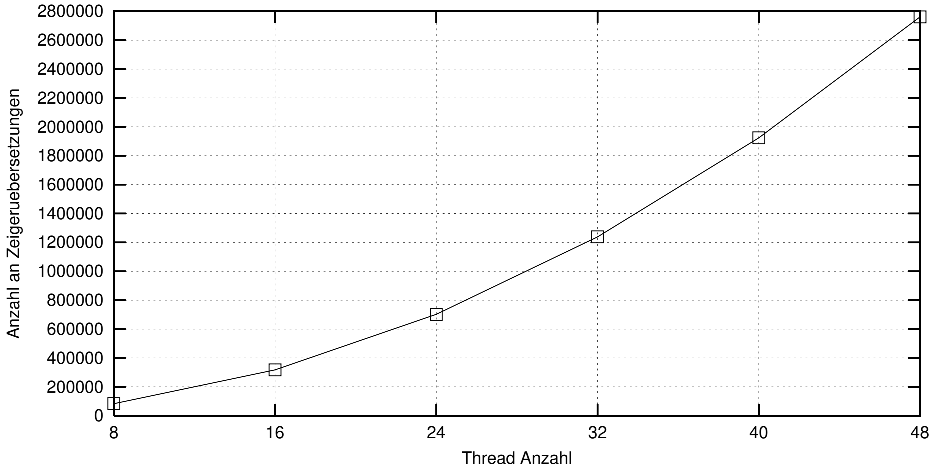

Abbildung 6.4: Zeigerübersetzungen beim speziellen Testprogramm

Dies bestätigt jedoch die Aussage, daß dem Netzwerkverkehr in einem verteilten System in bezug auf die Performance herausragende Bedeutung zukommt. Auch wenn die Zeigerübersetzungen in großer Anzahl stattfinden, haben sie, da sie jeweils nur einige einfache Rechenschritte erfordern, welche lokal auf dem jeweiligen Knoten ausgeführt werden, wesentlich geringeren Einfluß auf die Gesamteffizienz des DSM-Systems als es die Anzahl an Seitentransfers hat. So zeigt sich in Abbildung 6.5, in der die Anzahl der Seitenzugriffsfehler und der Seitentransfers für das Testprogramm unter

Verwendung der unterschiedlichen Versionen des DSM-Heap dargestellt sind (für die Kurven für die unterschiedliche Anzahl von Rechnern gilt gleiches wie für Abbildung 6.4), eine recht ähnliche Differenz zwischen den Kurvenanstiegen wie für die Ausführungszeiten des Testprogramms in Abbildung 6.2. Bei 48 Threads ergibt sich eine Verminderung der Seitenzugriffsfehler bzw. der Seitentransfers vom DSM-Heap ohne ausgelagerte Freelist aus Kapitel 5 zum DSM-Heap aus Kapitel 4 von 7.3% bzw. 4.8%, vom DSM-Heap mit ausgelagerter Freelist aus Kapitel 5 zum DSM-Heap ohne ausgelagerte Freelist aus Kapitel 5 von 34.5% bzw. 22.4% und vom DSM-Heap mit ausgelagerter Freelist aus Kapitel 5 zum DSM-Heap aus Kapitel 4 von 39.3% bzw. 26.1%.

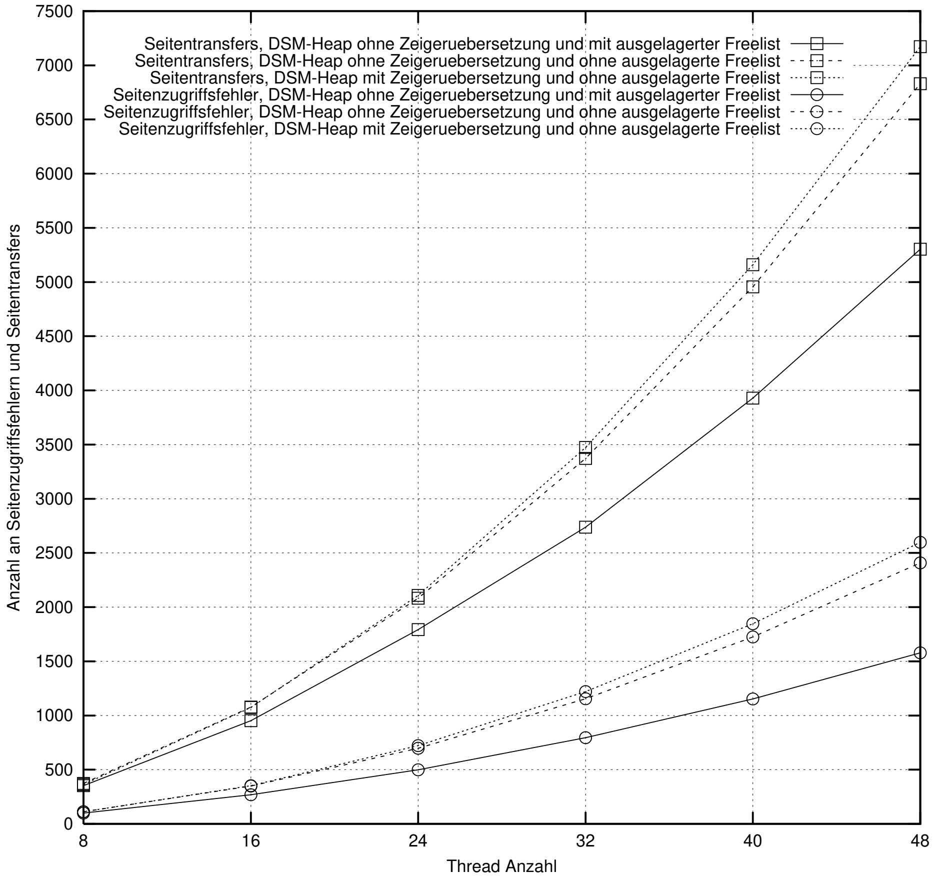

Abbildung 6.5: Seitenzugriffsfehler und -transfers beim spez. Testprogramm

Die leicht reduzierte Anzahl von Seitenzugriffsfehlern und Seitentransfers beim DSM-Heap ohne ausgelagerte Freelist aus Kapitel 5 gegenüber dem DSM-Heap aus Kapitel 4 ergibt sich daraus, daß bei letzterem bei jedem `dsm_malloc` und jedem `dsm_free` ein zusätzlicher Schreibzugriff auf die Seite vorgenommen wird, auf der sich der Zeiger befindet, für den die jeweilige

Funktion aufgerufen wurde. Dies ist notwendig um die Konsistenz der Zeigertabelle sicherzustellen. Über diese Vorgehensweise wird ein Transfer dieser Seite und damit auch der aktuellen Informationen in der Zeigertabelle, die diese Seite betreffen, erzwungen, sofern der betreffende Knoten nicht zufällig ohnehin zu diesem Zeitpunkt der Besitzer einer aktuellen Kopie dieser Seite ist, was jedoch zu dem Zeitpunkt keineswegs immer der Fall sein muß.

Die stark reduzierte Anzahl von Seitenzugriffsfehlern und Seitentransfers beim DSM-Heap mit ausgelagerter Freelist aus Kapitel 5 gegenüber dem DSM-Heap ohne ausgelagerte Freelist aus Kapitel 5 ergibt sich hingegen ausschließlich daraus, daß sich bei ersterem bei dem verwendeten Testprogramm alle Elemente der Freelist auf einer einzigen Seite im DSM-Heap befinden, nämlich auf der ersten Seite des ersten und in dem Fall einzigen Verwaltungsblocks. Demzufolge muß in dem Fall bei einem Durchsuchen der Freelist lediglich auf diese Seite zugegriffen werden, wohingegen bei dem verwendeten Testprogramm bei nicht ausgelagerter Freelist bei einem Durchsuchen derselben auf alle jeweils bis zu diesem Zeitpunkt erstellten DSM-Heap-Seiten zugegriffen werden muß.

Es zeigt sich hierbei also recht deutlich, daß eine Reduzierung der Seitenzugriffsfehler und damit der Seitentransfers weitaus größeren Einfluß auf die Performance des DSM-Systems hat als es eine Reduzierung der Anzahl der Zeigerübersetzungen hätte bzw. es der komplette Wegfall der Zeigerübersetzungen hat und in einem verteilten System ersterem immer besondere Beachtung geschenkt werden sollte.

Zusammenfassung

In dieser Diplomarbeit wurde eine Methode zur dynamischen Verwaltung von verteiltem, gemeinsam genutztem Speicher für DSM-Threads entwickelt und implementiert, wobei der Effizienz dieser Verwaltung besondere Aufmerksamkeit gewidmet wurde. Ausgehend von einer bereits funktionsfähigen, allerdings mit Einschränkungen an den Anwender in bezug auf die Benutzung von Zeigern verbundenen Lösung für einen dynamischen DSM (Kapitel 4) wurden deren Nachteile bzgl. der Effizienz beseitigt.

In bezug auf die Funktionalität des dynamischen verteilten Speichers, durften dabei keine Kompromisse gemacht werden. D.h. er muß „echt“ dynamisch in dem Sinne sein, daß der Anwender während der Laufzeit beliebig oft und, unter den physischen Bedingungen, die bzgl. der Menge an verfügbarem Speicher bei dem jeweils verwendeten verteilten System gegebenen sind, beliebig viel verteilten Speicher vom DSM-System anfordern kann, und daß er die erhaltenen Speicherabschnitte an das DSM-System zurückgeben und im folgenden wiederverwenden kann. Der Anwender soll den dynamischen verteilten Speicher also in gleicher Weise benutzen können wie er es von „normalem“ dynamischen Speicher gewohnt ist. Diese Anforderungen an den dynamischen DSM sind besonders wichtig, wenn das DSM-System in Zusammenhang mit Programmiermethoden zum Einsatz kommen soll, bei denen zur Laufzeit viele Speicheranforderungen etc. auftreten (OOP/C++).

Bei der Vorbereitung dieser Arbeit wurde von der Annahme ausgegangen, daß der Hauptperformancenachteil des dynamischen DSM, auf dem dabei aufgebaut wurde, darin bestand, daß sämtliche Zeiger, die sich auf einer zu transferierenden DSM-Seite befinden, an die auf den verschiedenen Knoten möglicherweise unterschiedlichen Adressen, auf die diese Zeiger verweisen, angepaßt werden mußten. Dieser Nachteil konnte durch die gezielte Plazierung der DSM-Heapblöcke im virtuellen Speicher der einzelnen Knoten, wodurch diese Blöcke und damit alle Objekte im DSM-Heap auf allen Knoten die gleichen Adressen haben, beseitigt werden, da damit die Notwendigkeit zur Übersetzung der DSM-Zeiger vollständig wegfällt.

Über diese Vorgehensweise konnte gleichzeitig der oben erwähnte funktionale Nachteil des dynamischen DSM mit Zeigerübersetzung beseitigt werden. Denn durch die ebenfalls weggefallene Notwendigkeit zur Registrierung aller

DSM-Zeiger in einer Zeigertabelle können auch die Einschränkungen komplett aufgehoben werden, die dem Anwender beim DSM-Heap aus Kapitel 4 in bezug auf die Benutzung von Zeigern in den DSM-Heap auferlegt werden müssen. Der Anwender kann nun (d.h. bei Verwendung des DSM-Heap aus Kapitel 5) also auch Zeiger, denen er ihre Werte nicht über die Funktionen der Anwenderschnittstelle zum DSM-Heap zugewiesen hat, als DSM-Zeiger benutzen. Konnten z.B. vorher lediglich einfachverkette Listen verwendet werden (`next`-Zeiger: Aufruf von `dsm_malloc`, `prev`-Zeiger: dann nicht möglich, da für diese ebenfalls `dsm_malloc` aufgerufen werden müßte), jedenfalls wenn die Funktionen zur Registrierung und Freigabe von Zeigern in der Zeigertabelle nicht im Anwendungsprogramm verwendet werden sollen wie es beim speziellen Testprogramm in Kapitel 6 getan wurde, so können nun auch beliebig mehrfachverkettete Listen verwendet werden. Mit dem dynamischen DSM aus Kapitel 5 kann der Anwender DSM-Zeiger beliebig verwenden und manipulieren, d.h. sie können nun wie „gewöhnliche“ Zeiger benutzt werden.

Während der Performancetests stellte sich heraus, daß der mit der Zeigerübersetzung verbundene Effizienznachteil weniger Einfluß auf die Gesamtperformance des DSM-Systems hat als im Vorfeld angenommen wurde. Selbst bei einer großen Anzahl von zu übersetzenden Zeigern, d.h. wenn ein Großteil des Platzes auf einer zu transferierenden Seite von Zeigern belegt wird, spielt die Übersetzung all dieser Zeiger im Vergleich zu dem restlichen Aufwand, der bei einem Transfer dieser Seite betrieben werden muß, in bezug auf die Ausführungsgeschwindigkeit eine sehr untergeordnete Rolle.

Als ebenfalls erheblicher Performancenachteil wurde beim DSM-Heap mit Zeigerübersetzung aus Kapitel 4 wie auch schon beim semi-dynamischen DSM aus Kapitel 3 der Seitentransfer vermutet, der auftritt, wenn beim Durchsuchen der Freelist auf einem Knoten auf Elemente derselben zugegriffen wird, die sich auf DSM-Seiten befinden, von denen dieser Knoten zu diesem Zeitpunkt nicht zufällig gerade eine aktuelle Kopie besitzt. Dieser Seitentransfer kann in Abhängigkeit von dem eingesetzten Anwendungsprogramm recht beträchtliche Ausmaße annehmen. Diese Vermutung wurde in vollem Umfang bestätigt.

Durch ein Auslagern der Freelist, indem deren Elemente anstatt auf den DSM-Seiten selbst, auf denen sich die verwalteten freien Speicherabschnitte befinden, auf eigens dafür erstellten DSM-Heap-Seiten angelegt werden, konnte die Verteilung der Freelistelemente im DSM-Heap, d.h. die Anzahl an DSM-Seiten auf denen sie sich befinden, in erheblichem Maße eingeschränkt werden und so der bei einem Durchsuchen der Freelist auftretende Seitentransfer in ebensolchem Umfang reduziert werden.

Während der Performancetests zeigte sich dann, daß diese Maßnahme absolut den gewünschten Effekt hat. Bei dem eingesetzten speziellen Testprogramm konnten durch sie die Seitenzugriffsfehler und Seitentransfers bedeutend reduziert werden und in weit größerem Maße als durch den Wegfall der Zeigerübersetzung führte dies zu einer Erhöhung der Ausführungsgeschwindigkeit. Einschränkend muß dabei allerdings hinzugefügt werden, daß die starke Ausprägung dieses Effekts hierbei von dem Aufbau des speziellen Testprogramms begünstigt wurde. Bei dessen Konstruktion wurden die Hauptmaßnahmen zur Effizienzsteigerung der Verwaltung des DSM-Heap besonders berücksichtigt, indem extra viele DSM-Zeiger eingesetzt wurden und dafür gesorgt wurde, daß sich auf jeder Seite des DSM-Heap ein freier, zu verwaltender Speicherabschnitt befand.

Wie an den Ausführungszeiten von FFT und LU zu sehen ist, aufgrund deren Aufbaus sich diesbzgl. keine nennenswerten Unterschiede bei der Verwendung der verschiedenen Versionen des dynamischen DSM ergeben haben, hängt der Grad der durch die Maßnahmen zur Effizienzsteigerung erzielten Verbesserung der Ausführungsgeschwindigkeit immer von dem konkret eingesetzten Anwendungsprogramm ab, d.h. in welcher Art und Weise der DSM-Heap in dem jeweiligen Programm genutzt wird. Positiv hervorzuheben ist dabei jedoch, daß diese Maßnahmen in keinem Fall zu nennenswerten Verschlechterungen der Performance führen. Die Einschränkung „nennenswert“ muß hierbei gemacht werden, da in einer sehr eingeschränkten Gruppe von Anwendungsprogrammen sehr geringfügige Verschlechterungen der Ausführungsgeschwindigkeit auftreten können. Dies betrifft Programme in denen nur einige wenige, kleine Abschnitte aus dem DSM-Heap angefordert werden und deshalb insgesamt nur einige wenige DSM-Heap-Seiten genutzt werden und bei denen sich freie Speicherabschnitte im DSM-Heap auf nicht mehr als einer DSM-Seite befinden. Der Aufwand der Maßnahmen zur Effizienzsteigerung wie das Anlegen eines Verwaltungsblocks für die Freelist und der zusätzliche Transfer der Seite mit deren Elementen würde in diesem Fall größer sein als der durch sie zu erzielende Performancegewinn. In dem Fall würde der semi-dynamische DSM eine etwas bessere Performance haben, die sich jedoch nur in äußerst geringem Maße bemerkbar machen würde.

Zusammenfassend kann gesagt werden, daß sich in dieser Diplomarbeit bestätigt hat, daß in einem verteilten System, in dem die Knoten über ein Netzwerk verbunden sind, der Transfer von Nachrichten zwischen den Knoten immer einen Flaschenhals darstellt, der sich negativ auf die Performance des Systems auswirken kann, und daß sich dabei über eine möglichst umfangreiche Reduzierung des Netzwerkverkehrs wie dies in bezug auf den Transfer von DSM-Seiten mit der Auslagerung der Freelist getan wurde, eine weit erheblichere (in Abhängigkeit von der eingesetzten Applikation) Steigerung der Effizienz erzielen läßt als dies mit Maßnahmen erreicht werden kann, die ihre Wirkung hauptsächlich lokal auf den jeweiligen Knoten zeigen wie dies bei dem Wegfall der Übersetzung der DSM-Zeiger der Fall ist. Letzteres bringt zwar hier nur einen geringen Effizienzvorteil mit sich, ist jedoch trotzdem keinesfalls nur von geringer Bedeutung, da mit dem Wegfall der Einschränkungen bei der Benutzung von DSM-Zeigern eine wesentliche Verbesserung und eine Verallgemeinerung des Programmiermodells für den dynamischen DSM erreicht werden konnte.

Anhang A

Schnittstellenbeschreibung

Die Anwenderschnittstelle zum DSM-Heap orientiert sich (von den Funktionen zur Festlegung der Standardgrößen für die Heapblöcke abgesehen) an den „gebräuchlichen“ Schnittstellen für dynamische Speicherverwaltungen:

`void dsm_heap_size(size_t size)`
Setzt die Größe des ersten anzulegenden DSM-Heapblocks. Dabei wird `size` auf ein Vielfaches von `DSM_PAGESIZE` aufgerundet. Wird `dsm_heap_size` nicht aufgerufen, wird für die Größe des ersten Heapblocks eine Voreinstellung übernommen. `dsm_heap_size` muß vor `dsm_init` aufgerufen werden. Der Parameter `size` muß größer als oder gleich `0` sein. In letzterem Fall unterbleiben einige Initialisierungen für den DSM-Heap und dieser kann nicht genutzt werden.

`void dsm_heap_incr_size(size_t size)`
Wie `dsm_heap_size`, legt jedoch die Größe der DSM-Heapblöcke fest, die inkrementell erstellt werden, sobald der Platz im ersten bzw. in allen zuvor erstellten Heapblöcken nicht mehr ausreicht, um eine Forderung des Anwenders nach einem Abschnitt aus dem DSM erfüllen zu können. `size` muß größer als `0` sein.

`void *dsm_malloc(size_t size)`
Liefert einen Zeiger auf einen DSM-Abschnitt der Größe `size` bzw. `NULL`, falls der Anforderung nicht nachgekommen werden kann. Der Speicherabschnitt ist uninitialisiert (dies gilt ebenso für `dsm_realloc`, `dsm_valloc` und `dsm_memalign`) und ist derart ausgerichtet, daß jeder Datentyp in ihm gepeichert werden kann (dies gilt ebenso für `dsm_realloc` und `dsm_calloc`).

`void dsm_free(void *ptr)`
Gibt den DSM-Abschnitt, auf den `ptr` zeigt, wieder frei. Dabei muß mit `ptr` der Rückgabewert eines zuvor ausgeführten `dsm_malloc`, `dsm_realloc`, `dsm_calloc`, `dsm_memalign` oder `dsm_valloc` übergeben werden.

`void *dsm_realloc(void *ptr, size_t size)`
Ändert die Größe des DSM-Abschnitts, auf den `ptr` zeigt. Liefert einen Zeiger auf einen neuen DSM-Abschnitt der Größe `size` und gibt den alten DSM-Abschnitt, auf den `ptr` zeigt, via `dsm_free` frei. Zuvor wird der Inhalt des alten Abschnitts bis zur kleineren Größe beider Abschnitte in den neuen Abschnitt kopiert. Falls der Anforderung nicht nachgekommen werden kann, wird `NULL` zurückgeben und der Speicherabschnitt, auf den `ptr` zeigt, unverändert gelassen.

`void *dsm_calloc(size_t count, size_t size)`
Liefert einen Zeiger auf einen DSM-Abschnitt für einen Vektor mit `count` Elementen, von denen jedes die Größe `size` hat. Der komplette Speicherabschnitt ist mit `0` initialisiert. Liefert `NULL`, falls der Anforderung nicht nachgekommen werden kann.

`void *dsm_memalign(size_t alignment, size_t size)`
Liefert einen Zeiger auf einen DSM-Abschnitt der Größe `size`, dessen Anfangsadresse ein Vielfaches von `alignment` ist, bzw. `NULL`, falls der Anforderung nicht nachgekommen werden kann.

`void *dsm_valloc(size_t size)`
Liefert einen Zeiger auf einen DSM-Abschnitt der Größe `size`, dessen Anfangsadresse ein Vielfaches von `DSM_PAGESIZE` ist, bzw. `NULL`, falls der Anforderung nicht nachgekommen werden kann.

Literaturverzeichnis

[1] Frank Mueller: »Distributed Shared-Memory Threads: DSM-Threads«, *Workshop on Run-Time Systems for Parallel Programming*, pages 31-40, April 1997.

[2] Frank Mueller: »On the Design and Implementation of DSM-Threads«, *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315-324, June 1997.

[3] Technical Committee on Operating Systems and Application Environments of the IEEE: *Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[4] N. Hahn: »Dynamische Speicherverwaltung in einem verteilten System mit gemeinsam genutztem Speicher«, Studienarbeit, Institut für Informatik, Humbolt-Universität zu Berlin, März 1999.
www.informatik.hu-berlin.de/~mueller

[5] Andrew S. Tannenbaum: *Verteilte Betriebssysteme*, Prentice Hall, München, 1995.

[6] K. Li: »Shared Virtual Memory on Loosely Coupled Multiprocessors«, Ph. D. Thesis, Yale Univ., 1986.

[7] K. Li, P. Hudak: »Memory Coherence in Shared Virtual Memory Systems«, *ACM Trans. on Computer Systems*, Vol. 7, pages 321-359, November 1989.

[8] J. Silcock: »Distributed Shared Memory: A Survey«, Technical Report TR C95/22, School of Computing and Mathematics, Deakin University, June 1995.

[9] J. Protic, M. Tomasevic, V. Milutinovic: »Distributed Shared Memory: Concepts and Systems«, *IEEE Parallel and Distributed Technology*, pages 63-79, 1996.

[10] B. Nitzberg, V. Lo: »Distributed Shared Memory: A Survey of Issues and Algorithms«, *IEEE Computer*, Vol. 24(8), pages 52-60, August 1991.

[11] B. Fleisch, G. Popek: »Mirage: A Coherent Distributed Shared Memeory Design«, *Proc. 12th Symp. on Opearting Systems Principles*, ACM, pages 211-223, 1989.

[12] J.K. Bennet, J.B. Carter, W. Zwaenepoel: »Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence«, *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, pages 168-176, 1990.

[13] J.B. Carter, J.K. Bennet, W. Zwaenepoel: »Implementation and Performance of Munin«, *Proc. 13th Symp. on Operating Systems Principles*, ACM, pages 152-164, 1991.

[14] B.N. Bershad, M.J. Zekauskas: »Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors«, CMU Report CMU-CS-91-170, September 1991.

[15] B.N. Bershad, M.J. Zekauskas, W.A. Swandon: »The Midway Distributed Shared Memory System«, *Proc. IEEE COMPCON Conf.*, IEEE, pages 528-537, 1993.

[16] N. Carriero, D. Gelernter, J. Leichter: »Distributed Data Structures in Linda«, *Proc. ACM Symposium on Principles of Programming Languages*, ACM, 1986.

[17] D. Gelernter: »Generative Communication in Linda«, *ACM Trans. on Programming Languages and Systems*, Vol. 7, pages 80-112, January 1985.

[18] H.E. Bal:*Programming Distributed Systems*, Hemel Hempstead, England: Prentice Hall Int'l, 1991.

[19] H.E. Bal, M.F. Kaashoek, A.S. Tannenbaum: »Orca: A Language for Parallel Programming of Distributed Systems«, *IEEE Trans. on Software Engeneering*, Vol. 18, pages 190-205, March 1992.

[20] Frank Mueller: »A library implementation of POSIX threads under UNIX«, *Proceedings of the USENIX Conference*, pages 29-41, January 1993.

[21] K. Gharachorloo, D. Lenosik, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy: »Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors«, *International Symposium on Computer Architecture*, pages 15-26, 1990.

[22] A. Itzkovitz, A. Schuster, L.Wolfovich: »Thread Migration and its Applications in Distributed Shared Memory Systems«, *The Journal of Systems and Software*, 1997.

[23] B. Weissman, B. Gomes, J.W. Quittek, M. Holtkamp: »Efficient fine-grain thread migration with Active Threads«, *Proceedings of IPPS/ SPDP 1998*, Orlando, Florida, March 1998.

[24] R. Namyst, J.F. Mehaut: »PM^2: Parallel Multithreaded Machine«, *Proceedings of ParCo '95*, Gent, September 1995.

[25] I. Foster, C. Kesselman, R. Olson, S. Tuecke: »Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems«, Technical Report Version 1.3, Argonne national Labs, December 1993.

[26] E. Mascarenhas, V. Rego: »Ariadne: Architecture of a Portable Threads System Supporting Thread Migration«, *Software - Practice and Experience*, Vol. 26(3), pages 327-356, March 1996.

[27] E. Jul, H. Levy, N. Hutchinson, A. Black: »Fine-Grained Mobility in the Emerald System«, *ACM Transactions on Computer Systems*, Vol. 6(1), pages 109-133, February 1988.

[28] D. Cronk, M. Haines, P. Mehrotra: »Thread Migration in the Presence of Pointers«, *Technical Report, Institute for Computer Applications in Science and Engineering*, Number TR-96-73, December 1996.

[29] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, R.J. Littlefield: »The Amber System: Parallel Programming on a Network of Multiprocessors», *ACM Symposium on Operating System Priciples*, Dec. 1989.

[30] J. Casa, R. Konuru, S.W. Otto, R. Prouty, J. Walpole:»Adaptive Load Migration Systems for PVM«, *Proceedings of Supercomputing '94*, pages 390-399, Washington D.C., Nov. 1994.

[31] B.W. Kernighan, D.M. Ritchie: *Programmieren in C*, 2. Ausgabe, ANCI C, Carl Hanser Verlag/Prentice Hall, München/London, 1990.

[32] O.S.F. Carvalho, G. Roucairol: »On mutual exclusion in computer networks« *Comm. ACM*, Vol 26(2), pages 146-148, February 1983.

[33] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner: *Linux-Kernel-Programmierung*, Algorithmen und Strukturen der Version 2.0, 4. Auflage, Addison-Wesley, Bonn, 1997.

[34] G.Antoniu, L.Bougé, R. Namyst: »An efficient and transparent thread migrtion scheme in the PM2 runtime system«, *Proc. 3rd Workshop an Runtime Systems for Parallel Programming (RTSPP '99)*, Lect. Notes in Computer Science, Vol. 1586, San Juan, Puerto Rico, April 1999.

[35] J.P. Singh, W.D. Weber, A. Gupta: »SPLASH: Stanford Parallel Applications for Shared-Memory«, *Computer Architecture News*, Vol. 20(1), pages 5-44, March 1992.

[36] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta: »The SPLASH-2 Programs: Characterization and Methodological Considerations«, *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24-36, Santa Margherita Ligure, Italy, June 1995.

[37] S.C. Woo, J.P. Singh, J.L. Hennessy: »The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors«, *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASP-LOS-VI, pages 219-229, October 1994.

[38] D.H. Bailey: »FFT's in External or Hierarchical Memory«, *Journal of Supercomputing*, Vol 4(1), pages 23-35, March 1990.

Eigenständigkeitserklärung

Ich erkläre hiermit, die vorliegende Diplomarbeit vollständig selbstständig und ausschließlich unter Benutzung der angegebenen Quellen angefertigt zu haben.

Berlin, 27. März 2001 Niklas Hahn

Einverständniserklärung

Ich erkläre hiermit, daß ich mit der öffentlichen Auslage der vorliegenden Diplomarbeit durch das Institut für Informatik der Humboldt-Universität zu Berlin einverstanden bin.

Berlin, 27. März 2001 Niklas Hahn