Humboldt-Universität zu Berlin
Institut für Informatik

Dynamische Speicherverwaltung in einem verteilten System mit gemeinsamen Speicher

Studienarbeit

Niklas Hahn

Betreuung: Frank Müller, Ph.D.

Berlin, 21. März 2000

Inhaltsverzeichnis

Einleitung

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf und der Implementierung eines Modells zur dynamischen Speicherverwaltung für verteilte Systeme mit gemeinsamem Speicher.

Die Grundlage dieser Arbeit bilden die *DSM-Threads* [1, 2], ein Laufzeitsystem das auf den Pthreads [3] aufbaut und die Ausführung von verteilten Threads mit einem virtuellen, verteilten, gemeinsamen Speicher (DSM - distributed shared memory) realisiert. Dabei handelt es sich bei dem gemeinsam genutzten Speicher in der bisherigen Implementation der DSM-Threads ausschließlich um statischen Speicher. Ziel dieser Arbeit soll es nun sein, den gemeinsamen Speicher um eine dynamische Komponente zu erweitern, d.h., dem Benutzer des verteilten Systems die Möglichkeit zu geben, während der Laufzeit gemeinsamen Speicher vom System anfordern und zurückgeben zu können.

Eine wichtige Einschränkung bezüglich der Zielsetzung dieser Arbeit muß genannt werden, und zwar, daß die dynamische Speicherverwaltung nur für homogene, verteilte Systeme implementiert werden soll.

Kapitel 1 bietet eine kurze Einführung zu den DSM-Threads und skizziert die Schwierigkeiten einer dynamischen Verwaltung gemeinsamen Speichers. In Kapitel 2 wird der grundlegende Aufbau des hier implementierten dynamischen, gemeinsamen Speichers vorgestellt, bevor in Kapitel 3 auf dessen Einteilung in einzelne Speicherblöcke näher eingegangen wird. Die Verwaltung des Speichers innerhalb dieser Speicherblöcke wird in Kapitel 4 besprochen. Schließlich wird in Kapitel 5 eine Möglichkeit des Umgangs mit Zeigern, die auf unterschiedlichen Knoten gültig sein sollen, dargelegt.

Kapitel 1

Grundlagen

1.1 Distributed Shared Memory

Die Nutzung eines virtuellen, verteilten, gemeinsamen Speichers ist eine Technik zur Programmierung von Mehrfach-Prozessor-Systemen, in denen es keinen physikalischen, gemeinsam genutzten Speicher gibt [5]. Im Gegensatz zu der Annahme, daß Programme auf diesen Maschinen in verschiedenen Adreßräumen laufen, entwickelten Li [6] und Li/Hudak [7] die Idee eines einzigen, virtuellen Adreßraumes für ein solches System.

Dabei befindet sich jede Speicherseite auf genau einer Maschine.[1] Wird auf einer Maschine auf eine Seite zugegriffen, die sich nicht, bzw. von der sich keine gültige Kopie, auf dieser Maschine befindet, wird ein Hardware-Seitenzugriffsfehler ausgelöst. Daraufhin wird eine Nachricht an die Maschine gesendet, welche die Seite besitzt und die die Seite dann an die anfragende Maschine sendet. Nach dem Empfang der Seite kann der Zugriff auf sie erfolgreich wiederholt werden.

1.2 Die DSM-Threads

Die DSM-Threads [1, 2] bilden ein verteiltes Laufzeitsystem, welches verteilte Threads mit einem virtuellen, verteilten, gemeinsam genutzten Speicher unterstützt. Das System basiert auf den Pthreads [3] und dezentraler Kommunikation zwischen den einzelnen Knoten des verteilten Systems.

Es folgt eine kurze Beschreibung der grundlegenden Funktionsweise. Auf jedem Knoten, auf dem verteilte Threads ausgeführt werden sollen, wird ein Prozeß gestartet, der DSM-Systemthreads, ein Pthread-System [4] und Anwenderthreads enthält. Das DSM-Laufzeitsystem besteht aus zwei Serverthreads, dem Kommunikationsmanager und dem Seitenmanager, und aus einer gewissen Anzahl von Workerthreads. Ein Workerthread wird vom DSM-System beispielsweise aktiviert, um als Benutzerthread zu fungieren, wenn der Anwen-

[1] Dies gilt für Systeme in denen immer nur auf einem einzigen Prozessor zu einem bestimmten Zeitpunkt auf eine bestimmte Speicherseite lesend oder schreibend zugegriffen werden kann (Seitenzugriffsprotokoll „SingleReaderSingleWriter“). Soll solch ein Zugriff auf mehreren Prozessoren gleichzeitig erfolgen können, so muß es mehrere Lese- oder Schreibkopien einer Seite geben (Seitenzugriffsprotokolle „MultiReaderSingleWriter“ und „MultiReaderMultiWriter“).

der `dsm_thread_create` aufgerufen hat, um einen Thread auf einem entfernten Knoten zu starten.

Anfangs sind auf dem Startknoten alle Seiten des DSM-Speichers validiert und auf allen entfernten Knoten invalidiert. Ist eine DSM-Speicherseite auf einem Knoten validiert, bedeutet dies, daß dieser Knoten der Besitzer dieser Seite ist und auf ihm auf sie zugegriffen werden kann. Invalidiert bedeutet umgekehrt, daß die Seite vor Zugriff geschützt ist. Aus einem Zugriff auf eine invalidierte DSM-Seite resultiert das Signal `SIGSEGV`. Die Validierung bzw. Invalidierung der DSM-Seiten wird über den UNIX-Systemruf `mprotect` mit den Attributen `PROT_READ`, `PROT_WRITE` oder `PROT_NONE` vorgenommen

Der Kommunikationsmanager, er hat die höchste Priorität in diesem Prozeß, öffnet einen Kommunikationskanal und wartet auf eintreffende Nachrichten. Wenn eine Nachricht eintrifft, wird sie, abhängig von ihrem Typ, vom Kommunikationsmanager selbst behandelt oder einem Workerthread übergeben. Im Falle eines Zugriffsfehlers auf eine DSM-Seite, d.h. eines Zugriffsfehlers auf ein gemeinsam genutztes Objekt (mit Objekt wird in dieser Arbeit ein „Stück Speicher“ bezeichnet), also auf Daten, die auf mehreren Knoten zur Verfügung stehen sollen, bittet der Signalhandler den Seitenmanager (höchste Priorität - 1) um diese Seite und legt sich schlafen. Der Seitenmanager aktiviert dann einen Workerthread, der die Aufgabe hat, die benötigten Zugriffsrechte zu beschaffen, z.B. indem er eine Anfrage nach der betreffenden Seite an den wahrscheinlichen Seitenbesitzer schickt. Empfängt der Kommunikationsmanager eines Knotens eine derartige Anfrage und dieser Knoten ist nicht der Seitenbesitzer, so schickt er die Anfrage an den scheinbaren Besitzer weiter. Ist dieser Knoten aber selbst der Seitenbesitzer, dann aktiviert er einen Workerthread, der die Seite invalidiert und an den anfragenden Knoten schickt. Nachdem dessen Kommunikationsmanager die Seite empfangen hat, beauftragt dieser wiederum einen Workerthread damit, die Seite zu validieren und den Userthread aufzuwecken. Der Userthread kann dann den Zugriff mit Erfolg wiederholen.

1.3 Verwaltung von gemeinsamem Speicher

Aus der gemeinsamen Nutzung eines Speichers durch mehrere Knoten resultieren für dessen Verwaltung, insbesondere wenn der Speicher dynamisch verwaltet werden soll, einige nicht triviale Probleme, die hier kurz skizziert werden sollen.

Wie aus Abschnitt 1.2 ersichtlich, handelt es sich bei dem gemeinsamen Speicher der DSM-Threads um seitenbasierten Speicher. Bei solchem wird, im Falle eines Zugriffsfehlers auf ein gemeinsam genutztes Objekt, die Seite, auf der sich das Objekt befindet, nach dem oben angedeuteten Mechanismus verschickt. Daraus folgt die Notwendigkeit, daß die gemeinsamen Objekte auf allen beteiligten Knoten jeweils auf derselben Seite im DSM-Speicher stehen (die Seiten des gemeinsamen Speichers sind im DSM-System durchnumeriert), d.h., Objekt a muß auf allen Knoten auf DSM-Seite i stehen. Dazu ist es notwendig, daß die DSM-Daten in einem zusammenhängenden Bereich des Speichers stehen. Hinsichtlich der Schwierigkeit, dieses sicherzustellen, muß zwischen statischen, automatischen und dynamischen Objekten unterschieden werden.

In der bisherigen Implementation der DSM-Threads wird nicht der gesamte Adreßraum als gemeinsamer Speicher verwendet, sondern nur der Teil, welcher die statischen Objekte enthält, und davon auch nur jener Teil, der von mehreren Threads genutzt werden soll. Objekte mit statischer Lebensdauer, d.h. die Objekte, die während der gesamten Laufzeit vorhanden sind (in C globale Variablen und `static`-Variablen), werden vor der eigentlichen Programmausführung vom Laufzeitsystem im Datensegment des jeweiligen Prozesses initialisiert. Diese Objekte stehen also automatisch in einem zusammenhängenden Bereich, wenn sie alle unmittelbar nacheinander initialisiert werden. Darum gibt es an den Anwender die Auflage, sämtliche von ihm zur gemeinsamen Nutzung vorgesehenen Daten, zwischen zwei Makros `DSM_DATA_BEGIN` und `DSM_DATA_END`, global zu initialisieren.

Abgesehen von der Tatsache, daß es unsinnig wäre, wenn auch automatische Objekte, d.h. die Objekte, deren Lebensdauer von der Ausführung des Blocks abhängt, in dem sie definiert wurden (in C die lokalen Variablen abzüglich der `static`-Variablen), als gemeinsame Objekte zugelassen werden würden, wäre für diese Objekte die Bedingung, in einem zusammenhängenden Speicherbereich zu stehen, nur mit sehr erheblichem Aufwand zu realisieren. Denn diese Objekte werden in nicht vorschreibbarer Reihenfolge unter anderen, nicht gemeinsamen Objekten, auf dem Stack des jeweiligen Threads abgelegt.

Aus der Seitenbasierung eines gemeinsamen Speichers ergibt sich direkt ein weiteres Problem. Da hierbei immer ganze Seiten verschickt werden, muß Sorge dafür getragen werden, daß sich auf diesen Seiten ausschließlich gemeinsame Objekte befinden, damit keine „normalen“ Objekte verschickt bzw. überschrieben werden.

1.3.1 Dynamische Verwaltung

Wie in der Einleitung angedeutet wurde, sollen die DSM-Threads mit dieser Arbeit dahingehend erweitert werden, daß auch Objekte mit dynamischer Lebensdauer, d.h. Objekte, für die während der Laufzeit Speicher vom System angefordert werden kann, gemeinsam genutzt werden können.

Die einzige Möglichkeit während der Laufzeit zusätzlichen Speicher vom Betriebssystem zur Verfügung gestellt zu bekommen, besteht für einen Prozeß darin, die Grenze zwischen seinem Datensegment und dem unbelegten virtuellen Speicher zu verschieben. In diesem Speicherbereich, dem Heap, können neben den gemeinsamen dynamischen Objekten natürlich noch beliebige „normale“ dynamische Objekte angelegt werden. Darüber hinaus kann die Reihenfolge, in der für die einzelnen Objekte der Speicher angefordert wird, nicht reglementiert werden. Der Heap der einzelnen Prozesse kann also auf den unterschiedlichen Knoten, im Gegensatz zum Datensegment, eine recht unterschiedliche Struktur haben (ein möglicher Heap-Zustand ist in Abbildung 1.1 zu sehen), die sich während der Laufzeit ständig ändern kann. Daraus ergeben sich für die Verwaltung der dynamischen DSM-Objekte einige spezielle Schwierigkeiten.

Die Bedingung, daß alle DSM-Daten in einem zusammenhängenden Speicherbereich stehen sollen, kann offensichtlich nicht mehr ohne weiteres erfüllt werden.

Weiterhin muß damit umgegangen werden, daß die dynamischen Objekte, im Gegensatz zu den statischen, erst während der Laufzeit entstehen. Damit Objekte gemeinsam genutzt werden können, müssen sie auf allen beteiligten Knoten vorhanden sein. Wenn auf Knoten A ein dynamisches Objekt im Speicher angelegt wird (Objekt z in Abbildung 1.1), muß dies auf Knoten B bzw. auf allen Knoten, die auf dieses Objekt zugreifen wollen, ebenfalls geschehen.

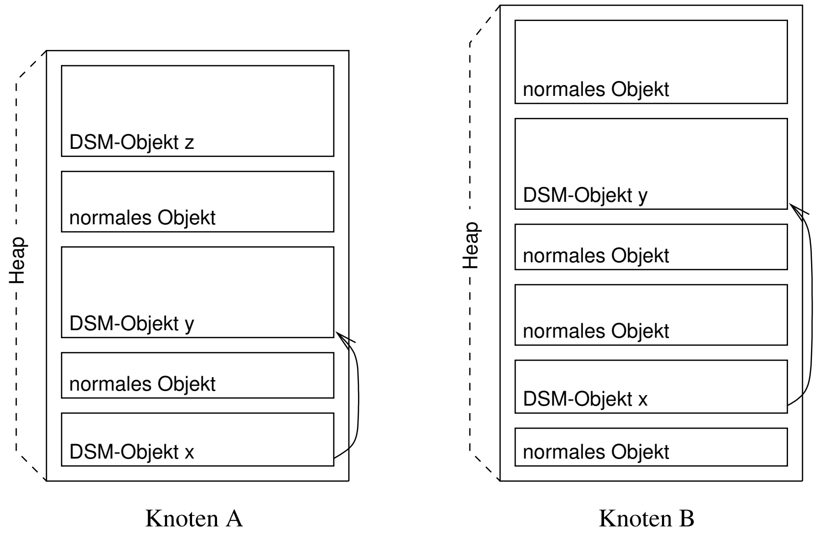


Abbildung 1.1: Unterschiedliche Heap-Struktur auf zwei Knoten

Ein weiteres Problem entsteht mit der Verwendung von Zeigern. Eine dynamische Speicherverwaltung ergibt wenig Sinn, wenn nicht gleichzeitig die Möglichkeit der Benutzung von Zeigern auf die dynamischen Objekte implementiert wird. Zu den unterschiedlichen Zeigern, d.h., aus welchem Speicherbereich sie in welchen verweisen dürfen, später mehr. Hier sei nur kurz das Problem der unterschiedlichen Objektadressen dargestellt. Aus den eben genannten Gründen können die einzelnen DSM-Seiten im Heap, im Gegensatz zu den DSM-Seiten im Datensegment (zumindest in einem homogenen System), auf den unterschiedlichen Knoten unterschiedliche Anfangsadressen haben, womit sich dann auch unterschiedliche Adressen für die Objekte auf diesen Seiten ergeben. Abbildung 1.1 zeigt auf Knoten A einen Zeiger, der in Objekt x steht und auf ein Objekt y verweist. Wird nun auf Knoten B über eine Zeiger-auf-Zeiger-Variable, die die Adresse des Zeigers in x enthält, auf den Inhalt dieses Zeigers, also auf y, zugegriffen, werden die Seiten mit den Objekten x und y von A nach B geschickt. Dann enthält diese Variable auf B die Adresse, die sie schon auf A enthielt. Diese Adresse muß aber auf B nicht die Adresse von x sein, da die Seite mit Objekt x auf B eine andere Anfangsadresse haben kann als auf A. Aus demselben Grund kann es sein, daß Objekt x auf B nicht mehr, wie auf A, die korrekte Adresse von Objekt y enthält. Aufgrund der unterschiedlichen Objektadressen können die Werte der Zeiger auf den verschiedenen Knoten unterschiedlich sein.

Kapitel 2

Der DSM-Heap

Mit der Bezeichnung „DSM-Heap“ sollen hier jene Stellen im Heap zusammengefaßt werden, welche die dynamischen DSM-Objekte enthalten. Für die Lösung der in Kapitel 1.3.1 genannten Probleme und anderer ist der Aufbau des DSM-Heap von grundlegender Bedeutung.

Diese Probleme sind sehr schwer zu lösen, wenn die gemeinsamen und die nicht gemeinsamen dynamischen Objekte in beliebiger Reihenfolge im Heap stehen können. Hinzu kommt, daß hier eine Lösung für einen seitenbasierten gemeinsamen Speicher gefunden werden soll. Das bedeutet, wenn die Anordnung der unterschiedlichen Objekte im Speicher in keiner Weise reglementiert werden würde, so würde im schlimmsten Fall (bei alternierender Anlegung gemeinsamer und nicht gemeinsamer Objekte) für jedes einzelne gemeinsame Objekt im Heap eine ganze Speicherseite benötigt werden. Wegen der nötigen Ausrichtung auf Seitengrenze würde in den meisten Fällen noch mehr Speicher als eine Seite benötigt werden.

2.1 Der Aufbau des DSM-Heap

Die Anordnung der verschiedenen Objekte im Heap kann reglementiert werden, indem die gemeinsamen Objekte in einem gesonderten Bereich des Heap angelegt werden.

Die einfachste Möglichkeit, dieses zu realisieren, besteht darin, während der Initialisierungsphase des DSM-Systems einen Block im Heap anzulegen (z.B. via `malloc`) und in diesem Block während der Laufzeit alle gemeinsamen dynamischen Objekte abzulegen. Dieses einfache Modell bringt den Vorteil mit sich, daß auch das Problem der falschen Zeigerwerte (siehe Kapitel 1.3.1) gleich mitgelöst wird. Denn wenn der DSM-Heapblock im Heap angelegt wird, bevor dort andere dynamische Objekte angelegt werden können, steht dieser auf allen Knoten am Beginn des Heap und hat demzufolge auf allen Knoten dieselbe Anfangsadresse. Damit haben auch alle gemeinsamen Objekte im Heap dieselben Adressen, sofern all diese Objekte auf allen Knoten und überall in derselben Reihenfolge angelegt werden, und folglich stellt sich das Problem unterschiedlicher Zeigerwerte nicht. In einer ersten Lösung wurde dieses Modell tatsächlich implementiert.

Doch der Zielsetzung dieser Arbeit würde dieses Modell nicht vollständig genügen, da es einen schwerwiegenden Nachteil beinhaltet. Wenn der DSM-Heapblock, und damit der gesamte verfügbare, dynamische, gemeinsame Speicher, bereits in der Initialisierungsphase des DSM-Systems angelegt werden soll, muß auch dessen Größe zu diesem Zeitpunkt festgelegt werden. Zwar kann dem Anwender angeboten werden, die Größe des DSM-Heap nach seinen Bedürfnissen festzulegen und in manchen Anwendungsfällen mag das den Ansprüchen des Anwenders genügen, doch im allgemeinen Fall bedeutet eine feste Größe für einen dynamischen Speicher zweifellos eine nicht hinnehmbare Einschränkung. Im Grunde kann von dynamischem Speicher in diesem Fall gar nicht mehr die Rede sein.

Einen dynamischen Speicher mit unbegrenzter Größe kann man jedoch auch innerhalb dieses Ansatzes mittels einer recht einfach zu implementierenden Erweiterung verwirklichen. Wenn die Möglichkeit eingebaut wird, den Heapblock beliebig oft während der Laufzeit zu vergrößern, kann der DSM-Heap unbeschränkt wachsen (im Rahmen der gegebenen Hardware). Wann immer der Speicherplatz im Heapblock verbraucht ist, kann über `realloc` eine Vergrößerung desselben realisiert werden (sofern der Heapblock ursprünglich mittels `malloc` angelegt wurde). Da dann aber nicht mehr sichergestellt werden kann, daß der Heapblock auf allen Knoten an der gleichen Adresse beginnt, stellt sich, wenn auch in eingeschränkter Form, wieder das Problem unterschiedlicher Objektadressen. Damit fällt ein Hauptvorteil dieses einfachen Strukturmodells wieder weg. Deshalb und weil ein `realloc` immer mit großem Aufwand verbunden ist, wird diese Lösung hier nicht verfolgt.

Das gerade beschriebene einfache Modell wird dahingehend modifiziert, daß zwar weiterhin mit einem gesonderten Bereich im Heap für die dynamischen DSM-Daten gearbeitet wird, dieser Bereich sich aber nunmehr aus mehreren einzelnen Speicherblöcken zusammensetzen kann. In Abbildung 2.1 ist dieser Ansatz skizziert.

Dabei werden die DSM-Heapblöcke immer erst dann angelegt, wenn Bedarf nach mehr dynamischen DSM-Speicher auftritt. Der erste Block wird erstellt sobald, vom Anwender das erste Mal Speicher für ein dynamisches DSM-Objekt angefordert wird. Dann werden alle weiteren derartigen Forderungen aus diesem Block bedient, solange genug Platz vorhanden ist. Benötigt ein Objekt mehr Speicher als in dem ersten Block noch frei ist, wird ein neuer angelegt usw. Wann immer also der Anwender eine gewisse Menge dynamischen DSM-Speichers benötigt, werden die bisher angelegten DSM-Heapblöcke durchsucht, ob in einem von ihnen noch ein Bereich dieser Größe frei ist (siehe dazu Kapitel 4). Ist das nicht der Fall, wird ein neuer Block angelegt. Für die Größe des ersten Blocks und der inkrementell anzulegenden Blöcke gibt es Standardwerte, die bei Bedarf vom Anwender eingestellt werden können. Diese Standardwerte können jedoch bei der Erstellung eines Blocks überschritten werden, wenn die Größe eines dynamischen DSM-Objekts die Standardblockgröße überschreitet.

Damit wird der entscheidende Vorteil dieses Ansatzes gegenüber dem einfachen Modell ersichtlich: da weder die Größe eines DSM-Heapblocks noch deren Anzahl begrenzt ist, ist nun auch nicht mehr die Gesamtgröße des DSM-Heap begrenzt. Dieser Vorteil (bzw. die Erfüllung dieser grundlegende Anforderung an einen dynamischen Speicher) muß jedoch mit einem deutlich höheren Verwaltungsaufwand für den DSM-Heap erkauft werden.

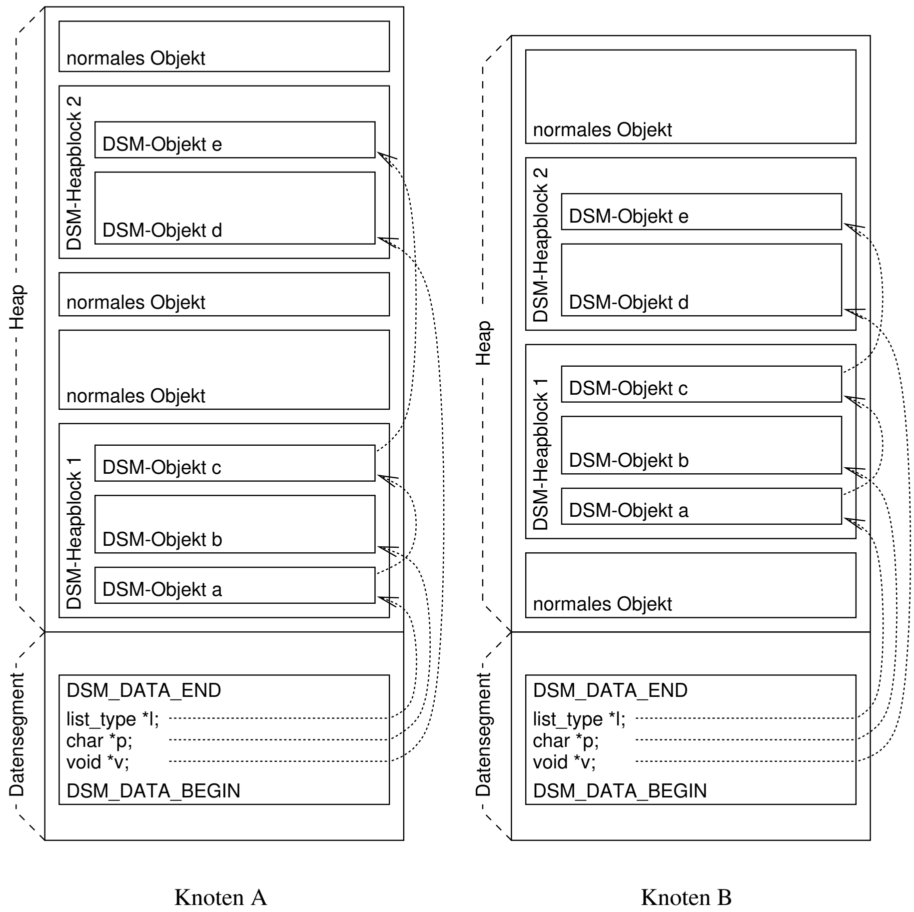


Abbildung 2.1: Struktur des DSM-Heap

Es muß dafür Sorge getragen werden, daß Anzahl, Größe und Reihenfolge der einzelnen Heapblöcke auf allen Knoten übereinstimmen. In Kapitel 3 wird darauf näher eingegangen.

Bei den DSM-Threads werden zahlreiche Informationen über alle Speicherseiten in einer Seitentabelle festgehalten. Die Details über die Seitentabelle sollen hier nicht erörtert werden. Von Relevanz ist nur, daß in der bisherigen Implementation der DSM-Threads die Seitentabelle in der Initialisierungsphase des DSM-Systems mit einer festen Größe angelegt wird, nachdem die Anzahl aller DSM-Seiten berechnet wurde. Da nun aber während der Laufzeit neue Seiten hinzukommen können, nämlich genau dann, wenn ein neuer Heapblock erstellt wird, muß auch die Möglichkeit der dynamischen Erweiterung der Seitentabelle implementiert werden. Siehe auch dazu Kapitel 3.

Der Speicher innerhalb der Heapblöcke muß geeignet verwaltet und über eine Schnittstelle dem Anwender zur Verfügung gestellt werden. Die Verteilung des Speichers über mehrere Blöcke macht dies nicht leichter. In Kapitel 4 wird erläutert, wie diese Aufgabe gelöst werden kann.

Die in Kapitel 1.3.1 aufgeführte Problematik der unterschiedlichen Objektadressen kommt bei dieser Struktur des DSM-Heap wieder voll zum tragen (siehe Abbildung 2.1). Eine Lösung dieses Problems wird in Kapitel 5 dargestellt.

2.2 Zugriff auf den DSM-Heap

Dem Anwender wird Speicherplatz aus dem DSM-Heap in der Form von Zeigern auf einzelne Speicherabschnitte übergeben. Das bedeutet, der Anwender kann nur Zeigervariablen verwenden, um auf gemeinsame dynamische Objekte zuzugreifen. Dabei kann er beliebig viele und beliebig große Objekte im DSM-Heap anlegen und kann Speicher wieder ans DSM-System zurückgeben. Wenn man von der Möglichkeit der Veränderung der Standardgrößen der Heapblöcke absieht, bekommt der Anwender von der Aufteilung des DSM-Heap in einzelne Blöcke nichts mit, denn diese Aufteilung führt zu keinerlei Einschränkungen oder Bedingungen hinsichtlich der Anwendung.

Es ist zwar nicht verboten und führt zu keinen Fehlern, wenn der Anwender Speicher aus dem DSM-Heap anderen als DSM-Variablen zuweist, doch das macht natürlich keinen Sinn. Der Sinn eines gemeinsamen dynamischen Speichers besteht darin, allen Knoten den Zugriff auf ein und dasselbe dynamische Objekt zu ermöglichen. Dies ist nur über gemeinsame Variablen möglich. Demzufolge müssen die Variablen, denen dynamische DSM-Objekte zugewiesen werden sollen, im statischen DSM-Datenbereich stehen (wie in Abbildung 2.1 zu sehen), d.h., der Anwender muß sie, wie die anderen gemeinsamen Variablen, zwischen den Makros `DSM_DATA_BEGIN` und `DSM_DATA_END` initialisieren.

Kapitel 3

Verwaltung der DSM-Heapblöcke

Die Verwaltung der Heapblöcke bezieht sich auf deren Erstellung auf den einzelnen Knoten, und zwar zum richtigen Zeitpunkt, in der richtigen Größe und in der richtigen Reihenfolge.

In Kapitel 2.1 wurde bereits dargelegt, daß ein grundsätzlich neuer Heapblock, d.h. ein Heapblock, der noch auf keinem Knoten existiert, immer dann erstellt wird, wenn der Platz in den bisher erstellten Blöcken nicht mehr ausreicht, eine Anforderung des Anwenders nach dynamischen DSM-Speicher zu befriedigen. Es reicht aber nicht aus, auf den einzelnen Knoten nur diese völlig neuen Heapblöcke bei Bedarf zu erstellen, sondern es müssen auf ihnen auch jene Blöcke erstellt werden, die bereits auf anderen Knoten angelegt wurden. Daraus folgt, daß zwischen den Knoten Informationen über die Struktur des DSM-Heap ausgetauscht werden müssen.

3.1 Erstellung eines Heapblocks

Zur Erstellung eines Heapblocks muß als erstes dessen Größe berechnet werden. Da mit einem seitenbasierten, gemeinsamen Speicher gearbeitet wird, müssen auch die einzelnen Heapblöcke in Seiten aufgeteilt werden. Also ist deren Größe immer ein Vielfaches der Seitengröße. Die Größe hängt natürlich von der Menge des Speichers ab, der vom Anwender für ein Objekt benötigt wird. Überschreitet diese Menge die Standardblockgröße, wird zur Ermittlung der Blockgröße die Größe des Objekts auf ein Vielfaches der Seitengröße aufgerundet. Andernfalls wird immer der Standardwert benutzt. Es gibt einen Standardwert für den ersten Heapblock und einen für alle weiteren anzulegenden Blöcke. Beide Werte können vom Anwender eingestellt werden. Die Unterscheidung zwischen dem ersten und den weiteren, inkrementellen Heapblöcken wird gemacht, um dem Anwender die Möglichkeit zu geben, alle gemeinsamen dynamischen Objekte in einem Heapblock unterzubringen, falls er in der Lage ist, deren Gesamtgröße vor dem Start der Anwendung abschätzen zu können. Aus Effizienzgründen ist es günstig, so wenig Heapblöcke wie möglich zu erstellen. Aus demselben Grund kann er die Standardgröße für die inkrementell anzulegenden Blöcke verändern.

Ist die benötigte Größe für den Heapblock berechnet, wird er mittels `malloc` im Heap angelegt. Wegen der nötigen Ausrichtung des Blocks auf eine Seitengrenze im Speicher muß dabei eine zusätzliche Menge Speicher mit der Größe einer DSM-Seite für ihn alloziert werden. Die Seitenausrichtung des Blocks erfolgt dann unter Anwendung des im DSM-System schon für die Ausrichtung des statischen DSM-Datenbereichs genutzten Algorithmus.

Wenn es sich bei dem Knoten, auf dem der Block gerade erstellt wird, um einen anderen als den Startknoten handelt, wird der Block nun mit `mprotect` vor jeglichem Zugriff geschützt. Man könnte meinen, es sei unsinnig, einen Block auf dem Knoten, auf dem er erstellt wird, vor Zugriff zu schützen, da ja wohl sowieso gleich auf ihn zugegriffen wird. Dem ist aber nicht so. Erstens, muß nicht unbedingt sofort nach der Erstellung eines Blocks, auf dem Knoten auf dem er erstellt wurde, auf ihn zugegriffen werden (im Falle der Erstellung eines Blockes, um die DSM-Heapstruktur auf einem Knoten zu aktualisieren). Und zweitens, hängen einige Informationen in der Seitentabelle der einzelnen Knoten davon ab, ob es sich um den Startknoten oder einen entfernten Knoten handelt. Dabei wird davon ausgegangen, daß die Seiten auf allen entfernten Knoten zu Beginn invalidiert sind, also vor Zugriff geschützt sind. Wären nun die Seiten in einem gerade erstellten Heapblock auf einem entfernten Knoten nicht erst einmal invalidiert, ergäben sich daraus Widersprüche zu den Einträgen in der Seitentabelle, was zu zahlreichen Fehlern im Ablauf führen würde. Zum Beispiel wäre keine korrekte Validierung/Invalidierung der Seiten mehr möglich, das System würde nicht mehr korrekt terminieren, usw.

Nun wird der neue Block in die Seitentabelle des Knotens eingetragen. Da in der Seitentabelle aber kein Platz für Erweiterungen vorgesehen ist, muß sie mittels `realloc` mit der entsprechenden Größe im Speicher neu angelegt werden. Der hohe Aufwand eines `realloc` für die Seitentabelle ist ein Grund für die bessere Effizienz weniger großer Blöcke gegenüber vieler kleiner.

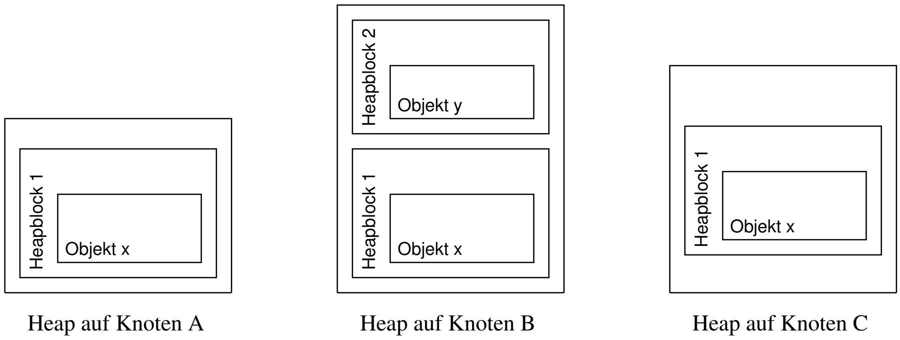
Als letztes werden die Informationen über die Struktur des DSM-Heap (siehe Abschnitt 3.3.1 und Kapitel 4.2) entsprechend ergänzt.

3.2 Anordnung der Heapblöcke

Wie bereits in Kapitel 2.1 erwähnt, ist es notwendig, daß Anzahl, Größe und Reihenfolge der einzelnen Heapblöcke auf allen Knoten übereinstimmen. Die Forderung nach der gleichen Anzahl kann dabei etwas abgeschwächt werden, und zwar hinsichtlich des Zeitpunktes ihrer Erfüllung.

Wann immer auf einem Knoten auf ein auf irgendeinem Knoten bereits erstelltes dynamisches DSM-Objekt zugegriffen werden soll, muß, wie leicht einzusehen ist, der Heapblock, in dem sich dieses Objekt befindet, auf diesem Knoten existent sein. Wenn dieser Block auf diesem Knoten bei Zugriff auf die Variable mit dem Objekt noch nicht existiert, muß er folgerichtig erst einmal angelegt werden, bevor auf das Objekt tatsächlich zugegriffen werden kann (1. und 2. in Abbildung 3.1). Dabei ist es aber nicht nötig, alle Blöcke, die bisher auf allen Knoten angelegt worden sind, auch auf diesem Knoten anzulegen (3. in Abbildung 3.1). Es reicht aus, auf diesem Knoten alle Heapblöcke anzulegen, die im Heap vor dem Heapblock liegen, welcher, infolge des Zugriff in ihn, eigentlich erstellt werden muß. Diese Forderung ist notwendig, damit die Reihenfolge der

Heapblöcke auf allen Knoten dieselbe ist. Diese Reihenfolge wiederum ist notwendig, weil die einzelnen Seiten aus den Heapblöcken auf allen Knoten jeweils an derselben Stelle in die Seitentabelle eingetragen werden müssen. Wird z.B. auf Knoten A ein Heapblock i angelegt und dabei die zweite Seite aus diesem Block als Seite 7 in die Seitentabelle eingetragen, so muß auch auf allen anderen Knoten die zweite Seite aus Heapblock i Seite 7 in der Seitentabelle sein. Daß dabei auch die Größe der einzelnen Heapblöcke auf allen Knoten gleich sein muß, ist leicht einzusehen.



1. Auf Knoten A wird Objekt x erstellt und dazu Heapblock 1 angelegt.
2. Auf Knoten B wird erstmals auf Objekt x zugegriffen und darum vorher Heapblock 1 angelegt. Dann wird auf Knoten B Objekt y erstellt und dazu Heapblock 2 angelegt.
3. Auf Knoten C wird erstmals auf Objekt x zugegriffen und darum vorher Heapblock 1 angelegt. Heapblock 2 muß zu diesem Zeitpunkt nicht auf Knoten C angelegt werden. Wird jedoch auf Knoten C zuerst auf Objekt y zugegriffen, so müssen die Heapblöcke 1 und 2 angelegt werden.

Abbildung 3.1: Unterschiedliche Heapblock-Anzahl auf verschiedenen Knoten

Aus demselben Grund ist es notwendig, daß, bevor auf einem Knoten ein völlig neuer Heapblock angelegt wird, auf diesem Knoten vorher alle bisher auf anderen Knoten angelegten Blöcke angelegt werden.

Die Erfüllung dieser Forderungen ist noch für einen anderen Aspekt von größter Bedeutung, und zwar für die Verwaltung des Speichers im DSM-Heap (Kapitel 4). Zur Begründung dessen sei hier als Vorgriff nur erwähnt, daß es dabei eine Liste der freien Speicherabschnitte innerhalb aller DSM-Heapblöcke gibt. Soll ein neues Objekt im DSM-Heap angelegt oder ein Objekt wieder freigegeben werden, wird jeweils diese Liste, im folgenden Freelist genannt, durchlaufen. Die Freelist würde, durch unterschiedliche Reihenfolge, Größe oder Anzahl der Heapblöcke auf den unterschiedlichen Knoten, zerstört werden.

3.3 Aktualisierung des DSM-Heap

Es gibt verschiedene Möglichkeiten, die eben aufgeführten Anforderungen an die Struktur des DSM-Heap zu erfüllen. Variieren können diese Möglichkeiten bezüglich der Art und Weise, wie die Informationen über die Struktur des DSM-Heap von einem Knoten an die anderen Knoten übermittelt werden, und bezüglich des Zeitpunktes der Aktualisierung des DSM-Heap auf einem Knoten, d.h., wann die Struktur des DSM-Heap auf einem Knoten vervollständigt wird.

3.3.1 Die Informationen über den DSM-Heap

Zunächst einmal muß geklärt werden, welche Informationen über die Struktur des DSM-Heap zwischen den Knoten übertragen werden müssen, um eine korrekte Aktualisierung zu ermöglichen. Gäbe es eine feste Größe für alle Heapblöcke, würde es ausreichen, die Anzahl der Heapblöcke auf dem jeweiligen Knoten zu transferieren. Doch wie in Abschnitt 3.1 dargelegt wurde, kann die Größe eines Heapblocks die Standardgröße überschreiten. Es müssen also auch Informationen über jeden einzelnen dieser größeren Blöcke übertragen werden.

Auf jedem Knoten werden die Anzahl der Seiten im DSM-Heap, die Anzahl der Heapblöcke und die Anzahl der größeren Heapblöcke auf diesem Knoten gespeichert. Außerdem gibt es eine Blocktabelle mit einem Eintrag für jeden Heapblock auf diesem Knoten, der die Anfangs- und Endadresse des Blocks und den Index seiner ersten Seite in der Seitentabelle enthält (siehe Abbildung 3.2). Diese Informationen werden im folgenden als „lokal“ bezeichnet, weil sie jeweils ausschließlich die aktuelle Struktur des DSM-Heap auf genau diesem Knoten beschreiben.

Zusätzlich zu diesen lokalen Informationen, gibt es auf jedem Knoten einige „globale“ Informationen über die DSM-Heap-Struktur. Diese werden global genannt, weil sie Informationen über alle bisher auf irgendwelchen Knoten erstellten Heapblöcke beinhalten und zwischen den Knoten ausgetauscht werden. Sie sind nicht zu jedem Zeitpunkt auf allen Knoten aktuell, sondern nur auf dem Knoten, der als letzter einen völlig neuen Heapblock erstellt hat und die globalen Heapinformationen darum erweitert hat, und auf allen Knoten, die sie danach geschickt bekommen haben (dazu mehr im nächsten Abschnitt). Sie umfassen genau die Informationen, die benötigt werden, um den DSM-Heap auf einem beliebigen Knoten vollständig aktualisieren zu können, nämlich die Anzahl der Heapblöcke, die Anzahl der größeren Heapblöcke und eine Tabelle mit einem Eintrag für jeden größeren Block, der die Anzahl der Seiten in diesem Block und seinen Index in der lokalen Blocktabelle enthält (Abbildung 3.2).

```
local:  int NumHeapPages
        int NumHeapBlocks                          -> | block_begin           |
        int NumGreaterHeapBlocks                      | block_end             |
        HBT_t HeapBlockTable[blockindex] ----------   | first_pagetable_index |

global: int NumBlocks
        int NumGreaterBlocks                       -> | NumPages              |
        GHBT_t GreaterHeapBlockTable[blockindex] --   | HeapBlockTable_index  |
```

Abbildung 3.2: Lokale und globale Informationen über den DSM-Heap

Aufgrund der schnelleren Bearbeitung wird hier die Speicherung von Informationen in Tabellenform gegenüber der Listenform vorgezogen, obwohl für eine Tabelle in der kein Platz mehr ist, ein aufwendiges `realloc` ausgeführt werden muß. Denn wie man unten sehen wird, werden die einzelnen Tabellen hier recht intensiv und vor allen Dingen sehr oft bearbeitet. Weiterhin kann die initiale wie die inkrementelle Größe der Tabellen entsprechend groß gewählt werden, so daß kein oder kein oftmaliges `realloc` notwendig wird.

Es muß auf die Konsistenz der globalen Heapinformationen geachtet werden. Sie dürfen nicht auf mehreren Knoten gleichzeitig verändert werden und sie müssen, sobald sie auf einem Knoten ergänzt werden sollen, weil auf ihm ein völlig neuer Block angelegt wurde, vollständig sein, d.h., sie müssen auf diesem Knoten bereits die Informationen über sämtliche bereits auf irgendwelchen Knoten angelegten Heapblöcke enthalten. Die Sicherstellung dieser Konsistenz wird jedoch bereits von einem anderen Bereich der Implementation quasi mitgeliefert, und zwar von dem Mechanismus der Konsistenzerhaltung der Freelist. Mehr dazu in Kapitel 4.2.1.

Die globalen Heapinformationen werden zu einem geeigneten Zeitpunkt zwischen den Knoten transferiert und dann zu einem geeigneten Zeitpunkt auf den einzelnen Knoten mit den lokalen Informationen verglichen und der Heap entsprechend aktualisiert.

3.3.2 Transfer der Heapinformationen

Wann und auf welche Art und Weise werden die globalen Heapinformationen am besten verschickt?

Eine Möglichkeit wäre, die Informationen über ein Broadcast von dem Knoten, auf dem gerade ein neuer Heapblock erstellt wurde und die Informationen dementsprechend ergänzt wurden, an alle anderen Knoten zu verschicken. Doch damit wären zwei Nachteile verbunden, die beide einen unnötig hohen Nachrichtentransfer ergeben würden. Erstens würden die Informationen dann auch an Knoten verschickt, auf denen sie vielleicht gar nicht benötigt werden, weil auf diesen Knoten nicht in diesen und in keinen danach erstellten Heapblock zugegriffen wird und weil auf diesen Knoten kein völlig neuer Heapblock mehr erstellt werden muß. Und zweitens wäre es hier nötig, daß von allen Knoten eine Empfangsbestätigung zurückgesendet wird. Das würde, neben einer Verdoppelung des Nachrichtentransfers, zu einer Verzögerung bei der Weiterarbeit auf dem Knoten führen, von dem der Broadcast ausgegangen ist, weil auf ihm auf den Empfang aller Bestätigungen gewartet werden müßte (Abbildung 3.3, links).

Eine andere Möglichkeit besteht darin, die globalen Heapinformationen nicht automatisch zu verschicken, sondern erst bei Bedarf. Dabei müßte ein Knoten, auf welchem die Informationen benötigt werden, eine Nachricht mit der Bitte um Übermittlung der Informationen an den Knoten schicken, auf welchem als letztes ein neuer Heapblock erstellt wurde (Abbildung 3.3, Mitte). Der Nachteil dieses Ansatzes ist, daß ein recht aufwendiges Protokoll benutzt werden müßte, um festzustellen, von welchem Knoten die Heapinformationen jeweils „geholt" werden müssen. Außerdem entsteht auch bei diesem Ansatz ein zusätzlicher Nachrichtentransfer.

Hier wird ein dritter Ansatz verfolgt. Die globalen Heapinformationen werden nicht nach jeder Änderung an ihnen oder nach Aufforderung verschickt, sondern immer, wenn eine Seite transferiert wird. Wann immer eine Seite von einem Knoten verschickt wird, werden die globalen Informationen über die Struktur des DSM-Heap in einen Puffer kopiert, dann in die Nachricht, welche die Seite enthält, eingefügt und so mit der Seite mitgeschickt (Abbildung 3.3, rechts). Der Empfänger-Knoten entpackt die Informationen wieder und, wenn sie aktueller sind als seine Informationen, wenn die empfangenen Informationen also aussagen, daß auf dem Sender-Knoten mehr Heapblöcke existieren als auf dem Empfänger-Knoten, dann werden die alten globalen Heapinformationen auf diesem Knoten mit den neuen überschrieben.

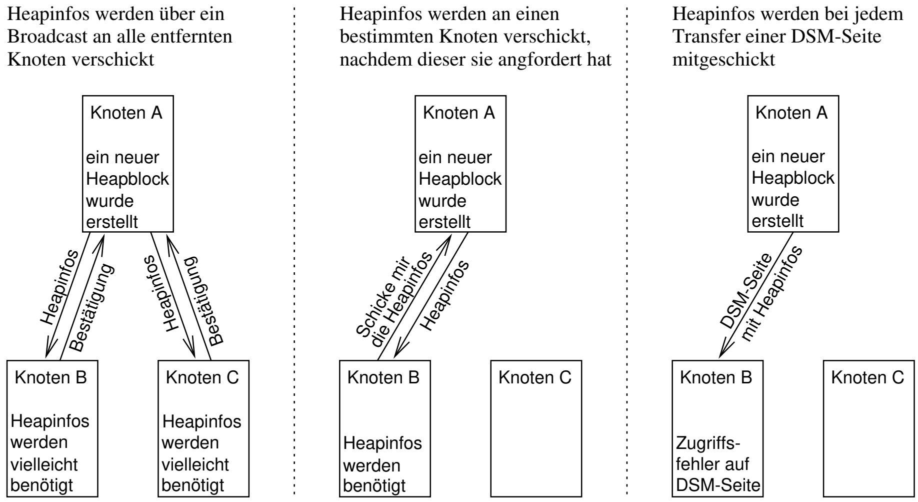


Abbildung 3.3: Möglichkeiten des Transfers der Heapinformationen

Der entscheidende Vorteil dieses Ansatzes gegenüber den anderen skizzierten besteht darin, daß diesmal keinerlei zusätzlicher Nachrichtentransfer benötigt wird. In einem verteilten System, in dem der Nachrichtentransfer über ein Netzwerk läuft, hat dieser Vorteil besondere Bedeutung, da der Netzwerkverkehr immer einen Flaschenhals darstellt. Es macht also Sinn, den Nachrichtentransfer soweit wie möglich zu reduzieren. Zwar werden bei diesem Ansatz alle Nachrichten, mit denen Seiten verschickt werden, etwas größer, aber da die Heapinformationen nicht viel Platz benötigen (sie umfassen immer zwei `long`-Werte für die Anzahl der Blöcke und der größeren Blöcke und je zwei `long`-Werte für die Anzahl der Seiten und den Blocktabellen-Index eines jeden größeren Blocks, also selbst bei 100 größeren Blöcken umfassen die Informationen gerade einmal 808 Byte), fällt dieser Nachteil nicht sehr ins Gewicht.

Es stellt sich die Frage, ob die Heapinformationen, die von einem Knoten zu einem anderen Knoten übertragen werden, immer genug Informationen enthalten, um die zu diesem Zeitpunkt nötige Aktualisierung vollständig durchführen zu können. Um dies zu beantworten muß zwischen zwei Fällen, in denen die Heapinformationen eines Knotens benötigt werden, unterschieden werden.

Im ersten Fall wird auf einem Knoten auf ein von einem anderen Knoten bereits erstelltes dynamisches DSM-Objekt zugegriffen, obwohl der Heapblock in dem sich das Objekt befindet auf diesem Knoten noch nicht existiert (Abbildung 3.4, linke Seite). Dann benötigt dieser Knoten die Informationen über genau diesen Block und alle davor erstellten Blöcke. Da er die Informationen von dem Knoten bekommt, welcher gerade der Besitzer der Seite war, auf der sich das Objekt befindet, und auf diesem Knoten der Block, in dem sich diese Seite befindet, natürlich existieren muß, ist in diesen Informationen unbedingt mindestens dieser Block und alle davor erstellten Blöcke enthalten. Informationen über irgendwelche anderen später erstellten Blöcke auf anderen Knoten, sind in dem Fall uninteressant, da diese Blöcke nicht benötigt werden, um auf dieses Objekt zugreifen zu können.

Im anderen Fall, wenn ein neues Objekt erstellt werden soll, verhält es sich ähnlich (Abbildung 3.4, rechte Seite). Dabei wird die Freelist nach einem genügend großem freiem Speicherabschnitt durchsucht. Da nie ein neues Objekt erstellt wird, ohne die Freelist zu bearbeiten, muß der Knoten auf dem ein neues Objekt erstellt wird, grundsätzlich erst einmal der Besitzer der Seiten werden, auf denen sich die Elemente der Freelist befinden. Um Besitzer dieser Seiten zu werden, muß er diese von deren letzten Besitzer geschickt bekommen. Damit erhält er auch die Informationen über alle bisher erstellten Heapblöcke. Denn ein neuer Heapblock wird grundsätzlich erst erstellt, nachdem die Freelist erfolglos durchsucht worden ist und wird dann sofort in die Heapinformationen eingetragen. Das heißt der Knoten, der gerade die Freelist bearbeitet, verfügt immer auch über die Informationen über alle bisher erstellten Heapblöcke.

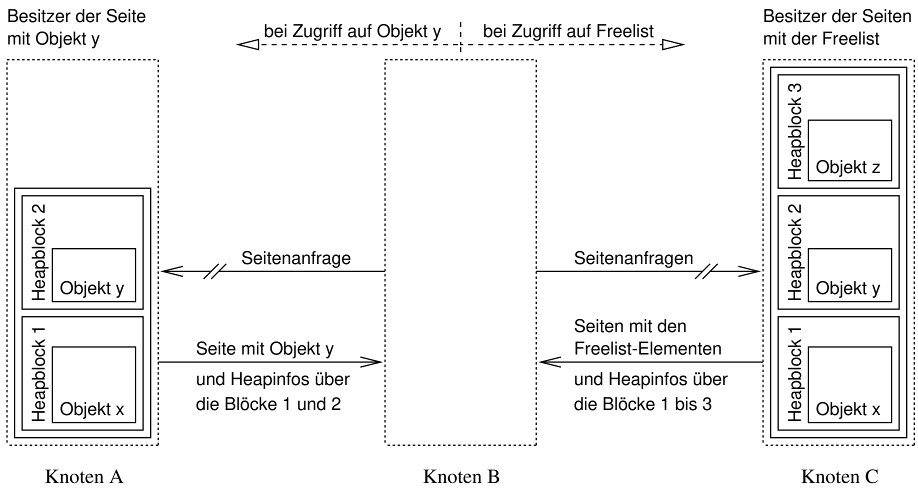


Abbildung 3.4: Vollständigkeit der übertragenen Heapinformationen

3.3.3 Zeitpunkt der Aktualisierung

Das erste Mal wird auf einem Knoten in einen bei ihm noch nicht existierenden Heapblock während der Zeigerübersetzung zugegriffen. Die Details der Zeigerübersetzung werden in Kapitel 5 behandelt. Für die Heapaktualisierung ist nur von Bedeutung, daß dabei die Werte sämtlicher Zeiger, also die Adressen auf die die Zeiger verweisen, auf einer gerade empfangenen Seite übersetzt werden. Denn solch ein Zeiger kann selbstverständlich von der gerade empfangenen Seite auf eine ganz andere Seite im DSM-Heap verweisen. Und genauso gut kann es sein, daß diese Seite in einem Heapblock liegt, der auf dem Knoten, der die Seite mit dem Zeiger gerade empfangen hat, noch nicht erstellt wurde. Bei jedem Seitenempfang müssen alle Zeiger übersetzt werden, da man nicht wissen kann, welche von ihnen vom Anwender darauffolgend genutzt werden und welche nicht. Der Wert eines Zeigers kann aber nicht übersetzt werden, wenn diese Adresse im DSM-Heap auf diesem Knoten nicht existiert. Also muß in diesem Fall eine Heapvervollständigung durchgeführt werden, und zwar bis zu dem Block, der die Seite enthält, auf die der Zeiger verweist.

Abgesehen von einer Ausnahme ist eine weitere Aktualisierung des Heap zu einem anderen Zeitpunkt als während eines Zugriffs auf einen nicht existenten Heapblock über einen Zeiger nicht notwendig. Der Zugriff auf die Heapblöcke erfolgt ausschließlich über Zeiger. Das geschieht entweder, wenn vom Anwender auf ein bereits erstelltes Objekt im DSM-Heap zugegriffen wird oder wenn auf die Elemente der Freelist in den Heapblöcken zugegriffen wird, weil vom Anwender ein DSM-Objekt wieder freigegeben bzw. ein neues Objekt im DSM-Heap erstellt werden soll (siehe Abbildung 3.4, eine Aktualisierung des Heap auf Knoten B erfolgt nach dem Empfang einer Seite von Knoten A oder C). Wird auf einem Knoten auf einen Heapblock zugegriffen, der auf diesem Knoten noch nicht existiert, dann erfolgt dieser Zugriff in jedem Fall über einen dieser Zeiger. Dieser Knoten kann nicht der Besitzer der Seite sein, auf der sich dieser Zeiger befindet, denn sonst könnte dieser nicht in einen nicht existenten Heapblock verweisen. Also findet ein Transfer dieser Seite zu diesem Knoten statt. Damit wird der Wert des Zeigers übersetzt und der fehlende Heapblock erstellt. Das bedeutet, daß immer, wenn auf einem Knoten auf einen auf diesem Knoten nicht existenten Heapblock zugegriffen werden soll, dieser Heapblock, während der Übersetzung des Zeigers, der auf diesen Heapblock verweist, erstellt wird.

Es kann allerdings eine Situation entstehen, in der auf einem Knoten alle Heapblöcke, die bisher auf anderen Knoten erstellt wurden, erstellt werden müssen, obwohl dieser Knoten in keinen dieser Blöcke zugreifen muß. Zur Erläuterung dessen folgt ein Beispiel (Abbildung 3.5). Auf Knoten A wird der Heapblock 2 neu erstellt, weil der Anwender für eine Variable p Speicher aus dem DSM-Heap angefordert hat und in Heapblock 1 nicht mehr genug Speicher frei ist. In den Heapblock 2 zeigt nun ausschließlich der auf Objekt y verweisende Zeiger in Variable p. Ein Element der Freelist befindet sich noch nicht in Heapblock 2, weil noch kein freier Speicherabschnitt in ihm vorhanden ist. Also gibt es in der Freelist noch keinen Zeiger, der in Heapblock 2 verweist. Nun soll auf Knoten B Objekt z erstellt werden, weil für eine Variable q ebenfalls DSM-Speicher angefordert wird. Steht dabei in Heapblock 1 nicht mehr genügend Speicher dafür zur Verfügung, muß auf Knoten B aus Heapblock 2 oder, falls auch in diesem nicht mehr genug Speicher frei ist, aus einem völlig neuen Heapblock 3 ein neuer Speicherabschnitt zur Verfügung gestellt werden. Dazu ist es notwendig, daß Heapblock 2 auf Knoten B existiert. Wurde aber auf Knoten B nach der Erstellung von Heapblock 2 auf Knoten A nicht auf Variable p zugegriffen und liegt Variable q im statischen DSM-Datenbereich auf einer anderen Seite als Variable p,[1] dann wurde Heapblock 2 auf Knoten B bisher nicht während einer Zeigerübersetzung erstellt. Denn auf Knoten B wurde bis zu diesem Zeitpunkt noch keine Seite, die einen Zeiger in Heapblock 2 enthält, empfangen.

[1] Würden die Variablen p und q im statischen DSM-Datenbereich auf der gleichen Seite stehen, so wäre Heapblock 2 während einer Zeigerübersetzung auf Knoten B erstellt worden. Denn da Knoten B über Variable q auf diese Seite zugreift, wäre diese Seite von Knoten A, der, weil er auf Variable p zugegriffen hatte, der aktuelle Besitzer dieser Seite war, zu Knoten B transferiert worden. Heapblock 2 wäre dann während der Übersetzung der Zeiger auf dieser Seite auf Knoten B erstellt worden, weil auch der in Heapblock 2 verweisende Zeiger aus Variable p auf Knoten B übersetzt worden wäre.

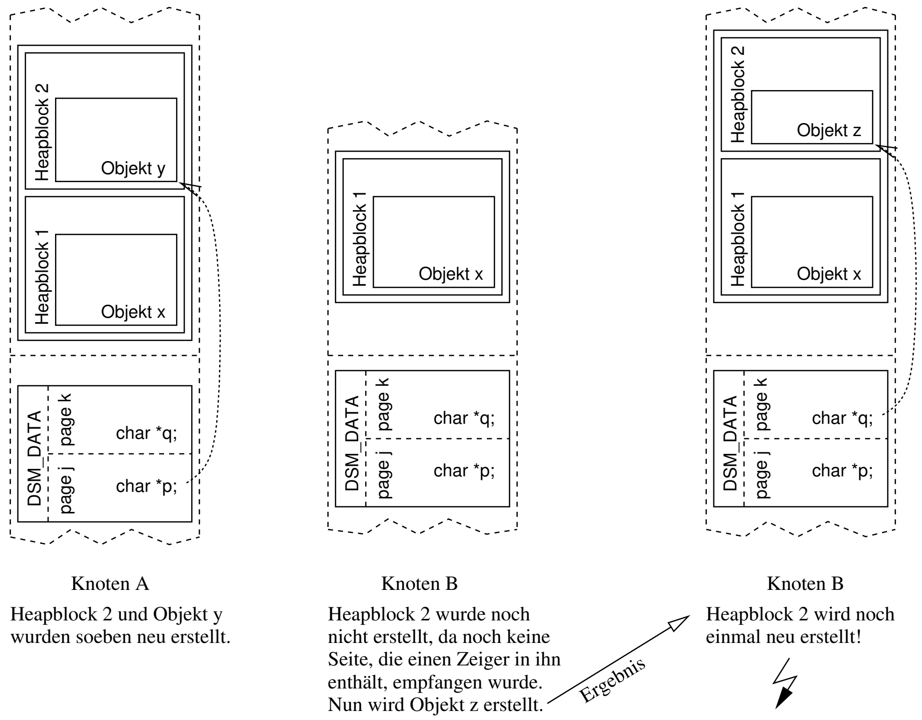


Abbildung 3.5: Ausbleibende Aktualisierung des DSM-Heap

Da in Heapblock 1 nicht mehr genug Platz ist, führt das dazu, daß auf Knoten B von der Speicherverwaltung des DSM-Heap der Heapblock 2 noch einmal als völlig neuer Block angelegt wird (Abbildung 3.5, rechts), woraus zahlreiche Fehler resultieren. Um dies zu vermeiden, werden immer, bevor von der Speicherverwaltung des DSM-Heap ein neuer Speicherabschnitt zur Verfügung gestellt wird, die globalen mit den lokalen Heapinformationen verglichen und der Heap entsprechend vervollständigt (siehe Funktion `dsm_sbrk` in Kapitel 4.2). Die aktuellen Heapinformationen stehen zu diesem Zeitpunkt auf dem betreffenden Knoten bereits zur Verfügung, da dieser auf die Freelist zugegriffen und damit aktuelle Heapinformationen geschickt bekommen hat (siehe Abschnitt 3.3.2).

Kapitel 4

Speicherverwaltung im DSM-Heap

Da an die Verwaltung des Speichers innerhalb der DSM-Heapblöcke keine speziellen Anforderungen oder Bedingungen gestellt werden müssen, muß dafür keine neue Verwaltungsmethode entwickelt werden.

Zu beachten ist dabei jedoch, daß die Struktur dieses Speichers auf allen Knoten gleich sein muß. Um dies sicherzustellen, müssen die Daten, welche die Speicherstruktur beschreiben, zwischen den Knoten ausgetauscht werden. Da ein Mitschicken dieser Daten bei einem Seitentransfer, wie es mit den globalen Heapinformationen praktiziert wird, aufgrund ihres Umfangs nicht praktikabel ist, muß dabei mit gemeinsamen, also mit DSM-Daten, gearbeitet werden. Und da deren Umfang nicht vorher absehbar ist, müssen dies dynamische DSM-Daten sein. Das hat zwei Konsequenzen. Zum einen sollten diese Daten auf möglichst wenige Speicherseiten verteilt sein, um zu vermeiden, daß jede Änderung an der Struktur des DSM-Heap, wenn ein neues dynamisches DSM-Objekt angelegt oder eines wieder freigegeben wird, einen allzu umfangreichen Seitentransfer nach sich zieht. Und zum anderen sollten diese Daten möglichst wenige Zeiger beinhalten, damit bei einem Transfer der Seiten, die jene Daten enthalten, nicht unnötig viele Zeigerübersetzungen stattfinden müssen.

Aus diesem Grund wird für den DSM-Heap die von Kernighan und Ritchie in ihrem Buch „Programmieren in C“ [8] vorgestellte Speicherverwaltungsmethode angewendet. Innerhalb dieser Methode wird nur eine vorwärtsverkettete Liste für die Verwaltung der freien Speicherabschnitte verwendet. Jeder verwaltete freie Speicherabschnitt enthält einen Kopf, in welchem die Größe des Abschnitts festgehalten wird und von dem auf den nächsten freien Abschnitt verwiesen wird. Im Gegensatz zu vielen anderen Speicherverwaltungsmethoden wird hier insgesamt nur ein Zeiger pro freiem Speicherabschnitt benötigt. Das bedeutet, daß hierbei zwar im Falle einer Anforderung oder Freigabe von Speicher auf einem Knoten, sämtliche DSM-Seiten, auf denen sich freie Speicherabschnitte befinden, auf diesem Knoten verfügbar sein müssen, aber bei einem Transfer dieser Seiten braucht nur eine minimal erhöhte Anzahl an Zeigern übersetzt werden.

Die erste der beiden oben genannten Konsequenzen aus der Notwendigkeit der Benutzung von dynamischen DSM-Daten zur Beschreibung der Struktur des DSM-Heap kann demzufolge mit der Verwaltungsmethode von Kernighan und Ritchie nicht gezielt umgesetzt werden. Denn die Seiten mit den Elementen der Freelist müssen, wann immer auf einem Knoten Speicher vom Anwender angefordert oder freigegeben wird, zu diesem Knoten transferiert werden, sofern dieser sie noch nicht besitzt. Dies kann nur umgangen werden, indem diese Daten, also die Informationen darüber, wo sich freie Speicherabschnitte befinden und wie groß diese sind, an anderer Stelle abgelegt werden, als innerhalb der freien Abschnitte selbst. Das soll jedoch erst in einer zukünftigen Arbeit verwirklicht werden, da Effizienz kein vorrangiges Ziel innerhalb dieser Arbeit war, auch wenn dieser Entwicklungsaspekt hier nicht völlig unbeachtet bleiben soll.

4.1 Speicherverwaltung nach Kernighan/Ritchie

Zu den Einzelheiten der Speicherverwaltungsmethode von Kernighan/Ritchie sei auf ihr Buch „Programmieren in C“ [8] verwiesen, hier soll nur deren grundlegende Funktionsweise kurz beschrieben werden.

Die Speicherverwaltung umfaßt drei Funktionen: `kr_malloc` vergibt Speicher an den Anwender, `kr_morecore` beschafft mehr Speicher vom Betriebssystem und `kr_free` nimmt Speicher vom Anwender zurück. Da für den verwalteten Speicher kein zusammenhängender Bereich im Heap garantiert werden kann, denn während der Laufzeit muß nicht ausschließlich über diese Speicherverwaltung Speicher vom System angefordert werden, besteht dieser aus einzelnen, von einander getrennten Speicherabschnitten. Die freien Speicherabschnitte werden über eine vorwärtsverkettete Liste, die Freelist, verwaltet. Jedes Element der Freelist enthält neben dem freien Abschnitt selbst eine Angabe über die Größe des Abschnitts und einen Zeiger auf den nächsten freien Abschnitt. Die Freelist ist nach den Adressen der freien Speicherabschnitte geordnet, und von dem letzten Element wird zurück auf das erste verwiesen (siehe Abbildung 4.1).

Wird vom Anwender via `kr_malloc` Speicher vom System angefordert, wird die Freelist durchlaufen und die Angaben über die Größe der freien Abschnitte mit der angeforderten Größe verglichen. Wird ein ausreichend großer Abschnitt gefunden, so wird dieser, sofern er genau die richtige Größe hat, aus der Freelist entfernt, oder es wird, wenn er größer als benötigt ist, der hintere Teil dieses Abschnitts dem Anwender übergeben und die Größenangabe für diesen Abschnitt entsprechend verringert. Wird kein ausreichend großer Abschnitt gefunden, so wird über den Aufruf von `kr_morecore` ein neuer Abschnitt erstellt, aus dem der Anwender dann bedient wird.

`kr_morecore` benutzt den UNIX-Systemruf `sbrk`, um mehr Speicher vom System zu erhalten. `sbrk(n)` gibt, sofern noch freier Speicher vom System geliefert werden kann, einen Zeiger auf einen `n` Bytes großen Speicherabschnitt zurück. Dazu verschiebt `sbrk` die Grenze zwischen dem Datensegment eines Prozesses und dem unbelegten virtuellen Speicher (siehe Kapitel 1.3.1). Da dies recht aufwendig ist, wird von `kr_morecore` immer eine Mindestgröße für `n` genommen, d.h., es wird immer gleich ein größerer Speicherabschnitt vom System angefordert, als für die Befriedigung der aktuellen Anfrage des Anwenders nötig wäre, und dem Anwender ein Teilstück daraus zur Verfügung gestellt. Benötigt dieser einen mehr als `n` Bytes großen Abschnitt, wird `n` entsprechend erhöht. Den neuen Speicherabschnitt übergibt `kr_morecore` letztlich der Funktion `kr_free`, damit er in die Freelist aufgenommen wird.

Die Funktion `kr_free` fügt einen Speicherabschnitt in die Freelist ein. Dies wird entweder nötig, wenn der Anwender einen zuvor mit `kr_malloc` angeforderten Speicherabschnitt wieder freigeben möchte, oder wenn mit `kr_morecore` ein völlig neuer Speicherabschnitt erstellt wurde. Da die Freelist nach den Adressen der Abschnitte geordnet ist, durchläuft `kr_free` zuerst die Freelist bis zu der Stelle, an der der neue Abschnitt eingefügt werden muß. Die richtige Stelle liegt entweder zwischen zwei Elementen der Freelist oder an ihrem Ende. Ist diese Stelle gefunden, wird der neue Abschnitt in die Freelist aufgenommen, indem die entsprechenden „next“-Zeiger angepaßt werden. Grenzt der neue Abschnitt direkt an einen bereits vorhandenen Abschnitt, so werden diese Abschnitte zu einem zusammengefaßt.

Diese einfache Speicherverwaltungsmethode von Kernighan/Ritchie arbeitet nicht sehr effizient, und in dieser Hinsicht gibt es durchaus bessere Methoden. Bezüglich der Verwaltung des Speichers im DSM-Heap spielt dieser Nachteil allerdings nur eine untergeordnete Rolle, weil die Vorteile, die sich aus den oben genannten Gründen für das DSM-System aus der einfachen Verwaltung mit wenigen Zeigern ergeben, hierbei schwerer wiegen.

Auch ein anderer allgemeiner Nachteil dieser Methode, und zwar, daß nie Speicher an das Betriebssystem zurückgegeben wird, beispielsweise durch den Aufruf von `sbrk(-n)`, ist bei ihrer Anwendung im DSM-System von keiner großen Bedeutung. Denn, wie im nächsten Abschnitt zu sehen sein wird, wird `sbrk` in der an den DSM-Heap angepaßten Speicherverwaltung nicht mehr benutzt. Also kann hier auch kein Speicher über `sbrk` ans Betriebssystem zurückgegeben werden. Die einzige Möglichkeit dies innerhalb des DSM-Systems tun zu können, besteht darin, einen der über das System-`malloc` erstellten Heapblöcke über das System-`free` wieder freizugeben. Doch eine derartige Speicherrückgabe wäre innerhalb des DSM-Systems mit besonders großem Aufwand verbunden. Neben dem hohen Aufwand für die Rücknahme des Speichers durch das Betriebssystem ergäbe sich hier zusätzlich ein recht hoher Aufwand für die Verkleinerung der Seitentabelle, die notwendig wird, weil die DSM-Seiten, auf denen sich der zurückgegebene Speicherabschnitt befand, wieder aus der Seitentabelle entfernt werden müßten. Darum würde sich innerhalb des DSM-Systems nur die Rückgabe von recht großen Speicherabschnitten lohnen, und das auch nur, wenn es unbedingt nötig wäre, weil z.B. der virtuelle Speicher vollständig belegt ist. Die Benutzung von allzu großen, gemeinsamen Datenmengen ist jedoch wegen der hohen Kosten für den Netzwerktransfer wenig sinnvoll. Es ist darum nicht empfehlenswert und wenig wahrscheinlich, daß der Anwender Mengen an Daten, die die heutzutage zur Verfügung stehenden Mengen an virtuellem Speicher überschreiten, als DSM-Daten benutzt. Deshalb kann im DSM-System auf eine Möglichkeit der Rückgabe von Speicher an das Betriebssystem verzichtet werden.

4.2 Anpassung an den DSM-Heap

Die Speicherverwaltungsmethode von Kernighan und Ritchie kann im wesentlichen unverändert für die Verwaltung des DSM-Heap genutzt werden. Zu ersetzten ist der UNIX-Systemruf `sbrk`. Der Anwender soll aus dem DSM-Heap bedient werden, wenn er gemeinsamen Speicher anfordert. Das bedeutet, daß der Speicher, den der Anwender zur Verfügung gestellt bekommt, bereits während der Erstellung der DSM-Heapblöcke über das System-`malloc` vom Betriebssystem angefordert wurde. Also soll `kr_morecore` nicht zusätzlichen Speicher vom Betriebssystem beschaffen, sondern einen neuen Speicherabschnitt aus einem DSM-Heapblock bereitstellen, wenn in der Freelist kein genügend großer Abschnitt mehr vorhanden ist, um die aktuelle Anfrage des Anwenders zu befriedigen. In Abbildung 4.1 ist das Layout des DSM-Heap in Verbindung mit der Freelist skizziert.

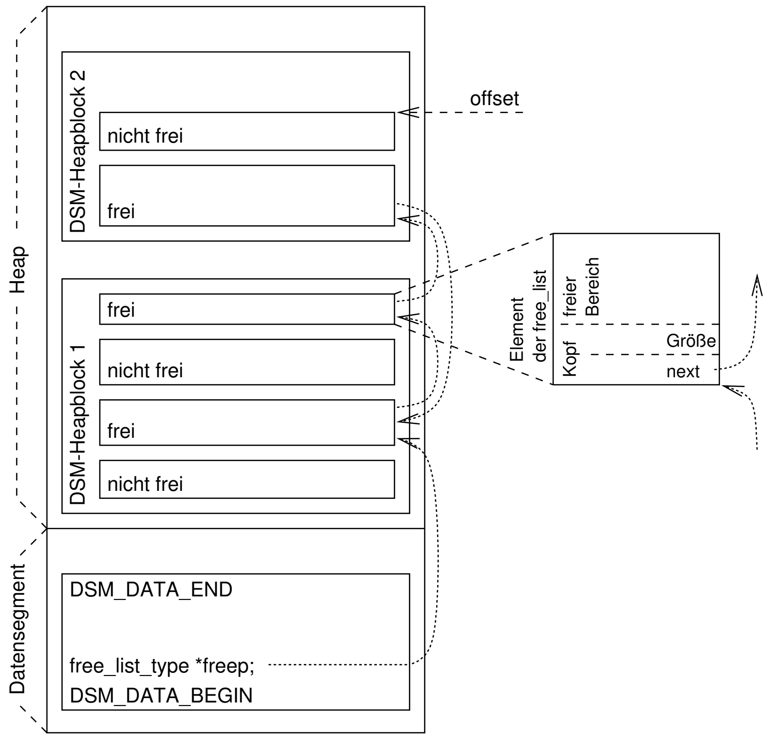


Abbildung 4.1: Die Liste der freien Speicherabschnitte im DSM-Heap

Statt des Systemrufs `sbrk` wird darum von `kr_morecore` die neue Funktion `dsm_sbrk` aufgerufen. `dsm_sbrk(n)` liefert einen Zeiger auf einen `n` Bytes großen Speicherabschnitt aus dem DSM-Heap. Aus den in Kapitel 3.3.3 dargelegten Gründen wird in `dsm_sbrk` zuerst der DSM-Heap auf dem betreffenden Knoten vervollständigt, falls dies nötig ist. Dann wird überprüft ob in dem zuletzt erstellten DSM-Heapblock noch genügend freier Speicher vorhanden ist, um aus ihm einen `n` Bytes großen Speicherabschnitt zur Verfügung zu stellen. Dafür gibt es eine statische DSM-Variable `offset`, in der festgehalten wird, wieviel Speicher aus dem letzten Heapblock bereits vergeben wurde (siehe Abbildung 4.1). `offset` beschreibt den Abstand zwischen dem Beginn des letzten Heapblocks und dem Ende des letzten an `kr_morecore` vergebenen Speicherabschnitts. Ist noch genügend Platz in diesem Heapblock vorhanden, so wird ein Zeiger auf die Adresse `blockbegin+offset` an `kr_morecore` zurückgegeben und `offset` um `n` erhöht. Ist nicht mehr genügend Platz in diesem Heapblock vorhanden, so wird ein völlig neuer Heapblock erstellt und die globalen Heapinformationen entsprechend ergänzt (siehe Kapitel 3.1), dann der Wert von `offset` auf `n` gesetzt und ein Zeiger auf den Anfang des neuen Blocks an `kr_morecore` zurückgegeben.

Der Zugriff auf die Freelist erfolgt über einen Zeiger auf ihr erstes Element, dem ersten freien Speicherabschnitt im DSM-Heap. Da auf diesen Kopf, wie auf die Freelist selbst, von allen Knoten zugegriffen wird, die Speicher aus dem DSM-Heap anfordern oder welchen wieder freigeben, muß es sich dabei um ein gemeinsames Objekt handeln und demzufolge im DSM-Speicher angelegt werden. Der Kopf muß bereits existieren, bevor das erste Mal Speicher aus dem DSM-Heap zur Verfügung gestellt werden kann, denn sonst kann nicht auf die Freelist zugegriffen werden. Also kann er nicht im DSM-Heap erstellt werden. Darum wird dieser im statischen DSM-Datenbereich initialisiert (siehe Abbildung 4.1). Bei der ersten Anforderung von Speicher aus dem DSM-Heap durch den Anwender, wird der Wert des „next"-Zeigers des Kopfes auf die Anfangsadresse des ersten Heapblocks gesetzt.

Eine weitere notwendige Anpassung der Speicherverwaltungsfunktionen von Kernighan/Ritchie an die Gegebenheiten des DSM-Systems betrifft die Zeiger der Freelist. Wie alle anderen Zeiger auf einer DSM-Seite müssen auch diese Zeiger während eines Transfers dieser Seit, übersetzt und dazu in einer Tabelle registriert werden (siehe Kapitel 5). Darum müssen die Funktionen `kr_malloc`, `kr_free` und `kr_morecore` um einige Funktionsaufrufe zur Registrierung der Zeiger in der Tabelle und der Wiederfreigabe aus dieser ergänzt werden.

4.2.1 Konsistenz der Freelist

Da die Struktur des Speichers innerhalb des DSM-Heap auf allen Knoten die gleiche sein muß, d.h., die einzelnen dynamischen DSM-Objekte müssen überall an der gleichen Stelle im DSM-Heap stehen, muß die Freelist auf allen Knoten identisch sein.

Zum Zeitpunkt des Zugriffs auf die Freelist auf einem Knoten, muß diese auf diesem Knoten alle Änderungen beinhalten, die bisher von irgendwelchen Knoten an ihr vorgenommen worden sind. Dies ist sichergestellt, wenn jeder Knoten, der auf die Freelist zugreifen will, diese immer von dem Knoten übernimmt, der sie als letzter verändert hat, oder von einem Knoten, der sie nach der Veränderung von diesem Knoten übernommen hat. Dies wiederum ist bereits sichergestellt, weil sich die Elemente der Freelist auf DSM-Seiten befinden und es zwar mehrere Lesekopien einer DSM-Seite geben kann aber nie mehr als eine Schreibkopie (innerhalb der DSM-Threads können bisher ausschließlich die Seitenzugriffsprotokolle „SingleReaderSingleWriter" oder „MultiReaderSingleWriter" verwendet werden). Ein Knoten, der auf die Seiten mit den Elementen der Freelist zugreift und sie nicht schon besitzt, bekommt diese also immer von einem Knoten, auf dem sie alle bisher gemachten Änderungen beinhaltet.

Es muß allerdings noch dafür gesorgt werden, daß nie mehrere Knoten gleichzeitig die einzelnen Elemente der Freelist bearbeiten, um Inkonsistenzen in ihr zu vermeiden. Bei jedem Zugriff auf die Freelist wird sie an irgendeiner Stelle verändert, denn es werden von jedem `kr_malloc` oder `kr_free`, den einzigen Stellen im Programm an denen auf die Freelist zugegriffen wird, Veränderungen an ihr vorgenommen, um einen freien Speicherabschnitt aus ihr zu entfernen bzw. einen aufzunehmen. Also muß die Freelist vor jeglichem gleichzeitigen Zugriff durch mehrere Knoten geschützt werden. Dies geschieht, indem sämtliche Aufrufe von `kr_malloc` und `kr_free` mit einem Mutex, im folgenden Heap-Mutex genannt, versehen werden und so vor gleichzeitigem Aufruf durch mehrere Knoten geschützt werden. Dabei muß es sich um denselben Mutex für beide Funktionen handeln. Da das Einsetzen des Heap-Mutex nicht dem Anwender überlassen werden soll, erhält dieser keinen direkten Zugriff auf `kr_malloc` oder `kr_free`, sondern bekommt die Funktionen `dsm_malloc` und `dsm_free` zur Verfügung gestellt, innerhalb derer `kr_malloc` und `kr_free` mit dem Heap-Mutex umgeben sind und aufgerufen werden, um DSM-Speicher anfordern oder wieder freigeben zu können.

Diese Vorgehensweise zur Konsistenzerhaltung der Freelist liefert direkt die Sicherstellung des Ausschlusses der gleichzeitigen Bearbeitung durch mehrere Knoten für sämtliche zur Verwaltung des dynamischen DSM-Speichers notwendige Datenstrukturen. Denn all diese Daten werden grundsätzlich nur infolge von Aufrufen von `dsm_malloc` oder `dsm_free` bearbeitet.

So ist mit der Verwendung des Mutex für alle Aufrufe von `kr_malloc` und `kr_free` garantiert, daß die globalen Heapinformationen nicht von mehreren Knoten gleichzeitig verändert werden können (siehe Konsistenz der globalen Heapinformationen in Kapitel 3.3.1). Diese Daten werden ausschließlich innerhalb von `dsm_sbrk` verändert bzw. ergänzt, denn nur an dieser Stelle wird ein völlig neuer DSM-Heapblock erstellt. Aufgrund dessen, daß `dsm_sbrk` wiederum ausschließlich von `kr_morecore` und `kr_morecore` ausschließlich von `kr_malloc` aufgerufen wird, kann die Veränderung dieser Daten nur auf dem Knoten erfolgen der gerade der Besitzer des Heap-Mutex ist. Damit und mit den im letzten Absatz von Kapitel 3.3.2 gemachten Aussagen über Vollständigkeit der Heapinformationen auf dem Knoten, auf dem ein neuer Heapblock erstellt werden soll, ist auch die geforderte Vollständigkeit der Heapinformationen zum Zeitpunkt ihrer Ergänzung sichergestellt. Die Konsistenz dieser Daten ist somit gesichert. Ähnliches gilt auch für die Zeigertabelle (siehe Kapitel 5.2.3).

Kapitel 5

Gemeinsame Zeiger

Unter „gemeinsamen Zeigern“ werden hier die Zeiger verstanden, welche auf ein dynamisches gemeinsames Objekt, also ein Objekt im DSM-Heap, verweisen und von mehreren Knoten genutzt werden. In Kapitel 1.3.1 wurde die Problematik der möglicherweise unterschiedlichen Werte dieser Zeiger erwähnt. Die DSM-Heapblöcke und damit die Objekte im DSM-Heap können auf den verschiedenen Knoten unterschiedliche Adressen haben. Darum können die Werte der einzelnen Zeiger, d.h., die Adressen, auf die die Zeiger verweisen, auf den verschiedenen Knoten unterschiedlich sein. Daraus folgt, daß die Werte der Zeiger, die sich auf einer zu transferierenden DSM-Seite befinden, nicht einfach durch den Empfängerknoten von dem Senderknoten übernommen werden können, sondern an die möglicherweise unterschiedliche Lage der Objekte auf diesen Knoten angepaßt werden müssen. Hier wird diese Anpassung vorgenommen, indem die Werte von sämtlichen Zeigern auf einer zu transferierenden DSM-Seite auf der Senderseite in eine „global“ gültige Normalform, d.h., in auf allen Knoten gültige Werte, übersetzt werden, und auf der Empfängerseite wieder in eine „lokal“ gültige Form, d.h., in Adressen die der Lage der betreffenden Objekte auf diesem bestimmten Knoten entsprechen, zurück übersetzt werden. Die Details dazu folgen unten.

5.1 Welche Zeiger sind betroffen?

Es müssen die Werte von allen Zeigern übersetzt werden, die auf eine Adresse im DSM-Heap verweisen und die von mehreren Knoten genutzt werden sollen, die also zwischen den Knoten transferiert werden sollen und darum auf allen Knoten gültig sein müssen. Theoretisch können alle Arten von Zeigern, d.h. Zeiger in statischen, automatischen und dynamischen Objekten, in den DSM-Heap verweisen, von denen jedoch nicht alle bei der Übersetzung berücksichtigt werden müssen.

Wie in Kapitel 2.2 gesagt wurde, müssen die Variablen, denen dynamischer DSM-Speicher zugewiesen werden soll, im statischen DSM-Datenbereich initialisiert werden. Die Werte aller Zeiger in den DSM-Heap, die in diesen statischen DSM-Objekten stehen, müssen übersetzt werden, da diese Zeiger auf allen Knoten gültig sein müssen. In den Objekten, auf die diese Zeiger verweisen, also in den dynamischen DSM-Objekten, können natürlich auch Zeiger in den DSM-Heap stehen, deren Werte ebenfalls auf allen Knoten gültig sein müssen, also übersetzt werden müssen.

Abgesehen davon, daß es keinen Sinn macht, wenn der Anwender Speicher aus dem DSM-Heap anderen als DSM-Variablen zuweist, ist die ebenfalls in Kapitel 2.2 gemachte Aussage, dieses führe zu keinen Fehlern, nicht ganz richtig. Durch die zur Übersetzung der Zeigerwerte nötige Registrierung der Zeiger in einer Tabelle (siehe nächsten Abschnitt) kommt es zu Fehlern, wenn der Anwender über `dsm_malloc` versucht, die Adresse eines DSM-Speicherabschnitts einem Zeiger, dessen Adresse sich außerhalb des DSM-Datenbereiches befindet, zuzuweisen. Von `dsm_malloc` wird die Adresse dieses Zeigers in der Zeigertabelle registriert. Dazu muß die DSM-Seite, auf der sich der Zeiger befindet, ermittelt werden. Das funktioniert natürlich nicht, wenn sich dieser Zeiger gar nicht auf einer DSM-Seite befindet, wenn er sich also in einem statischen oder dynamischen „Nicht-DSM"-Objekt oder in einem automatischen Objekt befindet.

Gleichwohl kann der Anwender auch mittels solcher „Nicht-DSM"-Zeiger auf den DSM-Heap zugreifen, um die von `dsm_malloc` bereitgestellten Speicherbereiche in irgendeiner Weise zu bearbeiten. Dabei ist es von keiner Bedeutung, daß die Werte dieser Zeiger nicht übersetzt werden können, denn dies ist nicht nötig, da sie nicht gemeinsam genutzt werden und nicht zwischen den Knoten transferiert werden.

5.2 Die Zeigertabelle

Um die Werte der betreffenden Zeiger übersetzen zu können, muß auf diese zugegriffen werden. Dazu muß das Programm Informationen darüber besitzen, an welchen Stellen sich im DSM-Speicher Zeiger befinden, also welche Adressen die Zeiger haben. Diese Informationen können bereitgestellt werden, indem die Adressen der Zeiger in einer Tabelle registriert werden [9].

5.2.1 Zeigerregistrierung

Die Adressen aller gemeinsamen Zeiger müssen vor deren erster Benutzung in der Zeigertabelle eingetragen werden. Wird ein Speicherbereich aus dem DSM-Heap, in den Zeiger verweisen, durch den Anwender wieder freigegeben, so müssen diese Zeiger wieder aus der Zeigertabelle entfernt werden, da sonst an Stellen im Speicher Adreßübersetzungen vorgenommen werden, an denen dies nicht geschehen darf. Es gibt verschiedene Möglichkeiten, wann und von wem die Registrierungs- und Freigabefunktionen für die Zeiger aufgerufen werden können.

Eine Möglichkeit besteht darin, den Anwender damit zu beauftragen, für alle betreffenden Zeiger Registrierungs- und Freigaberufe in sein Programm einzufügen. Auch wenn dies keine allzu schwierige Aufgabe ist, soll dies hier dem Anwender nicht zugemutet werden, denn es muß dabei durchaus mit einiger Sorgfalt vorgegangen werden, damit alle Zeiger zum richtigen Zeitpunkt registriert und wieder freigegeben werden und kein Zeiger mehrfach registriert wird. Zudem spricht gegen diese Möglichkeit, daß ein Hauptanliegen bei der Implementierung des DSM-Systems darin besteht, es seinen potentiellen Benutzern möglichst einfach zu machen, von einer nicht verteilten zu einer verteilten

Programmierung zu wechseln und bereits erstellte Programme mit möglichst geringem Aufwand an die DSM-Threads angepaßt werden können.

Den Compiler die Registrierung und Freigabe der Zeiger vornehmen zu lassen, stellt eine weitere Möglichkeit dar [9]. Dabei muß der Compiler so modifiziert werden, daß er mittels seiner Symboltabelle und sogenannter „use-def"-Ketten erkennt, wo aktive Zeiger definiert sind und diese automatisch in der Zeigertabelle einträgt bzw. aus ihr freigibt. Doch da eine Compilermodifizierung hier, z.B. aus Portabilitätsgründen, nicht in Frage kommt, wurde die Verwendung dieser Möglichkeit nicht ernsthaft erwogen.

Soll weder der Anwender mit der Zeigerregistrierung belastet werden, noch der Compiler entsprechend verändert werden, muß das DSM-System diese Aufgabe selbst erledigen.

Zum Ablauf in einer heterogenen Umgebung wird von den DSM-Threads das „External Data Representation"-Protokoll XDR [10] zur Codierung von Daten, die zwischen zwei Knoten mit unterschiedlicher Datenrepräsentation transferiert werden sollen, eingesetzt. Innerhalb der Initialisierung des DSM-Systems wird dazu eine XDR-Übersetzungstabelle generiert. Dabei wird für jede innerhalb der Makros `DSM_DATA_BEGIN` und `DSM_DATA_END` initialisierte Variable ein Eintrag in der Übersetzungstabelle erstellt. Die Einträge beinhalten unter anderem die Adresse der Variable und in Abhängigkeit vom Typ der Variablen den Name einer Initialisierungsfunktion für diese Variable. Diese Funktion wird während der Initialisierungsphase des DSM-Systems mit der Adresse der Variablen als Parameter aufgerufen.

Um die Zeiger innerhalb der DSM-Daten vom DSM-System automatisch registrieren zu lassen, kann als Initialisierungsfunktion für die Zeigervariablen die Zeigerregistrierungsfunktion genutzt werden. Dann wird während der Generierung der XDR-Übersetzungstabelle für jeden dieser Zeiger der Registrierungsruf in die Tabelle eingetragen, und alle zwischen `DSM_DATA_BEGIN` und `DSM_DATA_END` initialisierten Zeiger werden automatisch während der Initialisierung des DSM-Systems in der Zeigertabelle registriert. Der entscheidende Nachteil dieses Ansatzes wird damit jedoch schon deutlich: die automatische Registrierung bleibt auf die Zeiger im statischen DSM-Speicher beschränkt. Werden vom Anwender beispielsweise Listen benutzt, deren erstes Element er im `DSM_DATA`-Bereich initialisiert und deren Folgeelemente er im DSM-Heap anlegt, so werden nur die entsprechenden Zeiger des ersten Elements der Liste in der Zeigertabelle registriert. Dieselbe Problematik stellt sich auch für die Freelist (siehe Kapitel 4.2). Die Anzahl und die Adressen aller Elemente solcher Listen läßt sich nie schon während der Initialisierungsphase, d.h. vor der eigentlichen Laufzeit, bestimmen. Also können auch nicht alle Zeiger solcher Listen vor der eigentlichen Laufzeit in der Zeigertabelle registriert werden. Für die Zeiger der Folgeelemente muß dies auf eine andere Art und Weise geschehen. Wenn aber sowieso noch eine zusätzliche Methode zur Registrierung der Zeiger, die nicht über die XDR-Übersetzungstabelle registriert werden können, benötigt wird, können auch gleich alle Zeiger mittels dieser anderen Methode registriert werden.

Ein anderer Weg und dies ist schließlich der hier beschrittene, besteht darin, die betreffenden Zeiger erst während der Laufzeit, unmittelbar vor deren erster Benutzung, zu registrieren. Ein Zeiger wird frühstens zu dem Zeitpunkt das erste Mal benutzt, an dem das Objekt, auf das er verweisen soll, im Speicher angelegt wird. Das bedeutet, daß spätestens sobald via `dsm_malloc` Speicher aus dem DSM-Heap angefordert wird, der Zeiger, dem die Adresse auf den Speicherabschnitt übergeben werden soll, in der Zeigertabelle registriert werden muß. Darum wird für den Zeiger, für den der Anwender Speicher anfordert, die Zeigerregistrierungsfunktion innerhalb von `dsm_malloc` aufgerufen. Da zur Registrierung eines Zeigers dessen Adresse bekannt sein muß, wird `dsm_malloc` um einen Parameter erweitert, in dem die Adresse des Zeigers übergeben werden muß. Dies ist eine hinnehmbare Anforderung an den Anwender, da dies für ihn ein rein mechanischer Vorgang ist und mit keinerlei intellektueller Anstrengung verbunden ist. Die Zeiger, denen über `dsm_malloc` ein Speicherabschnitt zugewiesen wurde, müssen aus der Zeigertabelle entfernt werden, sobald dieser Speicherabschnitt vom Anwender wieder freigeben wird. Dazu wird `dsm_free` ebenfalls um einen Parameter zur Übergabe der Adresse des Zeigers, dem der freizugebende Speicherabschnitt zugewiesen wurde, erweitert und innerhalb von `dsm_free` die Funktion zur Freigabe eines Zeigers aus der Zeigertabelle für diesen Zeiger aufgerufen.

In dieser Form ist jedoch auch dieser Ansatz unzulänglich. Es ist unzureichend, nur die in den DSM-Heap verweisenden Zeiger zu registrieren, denen via `dsm_malloc` ein Speicherabschnitt aus dem DSM-Heap zugewiesen wird. Auch jene gemeinsamen Zeiger müßten registriert werden, für die `dsm_malloc` nicht aufgerufen wird, weil ihnen Adressen von Objekten in im DSM-Heap bereits erstellten Speicherabschnitten zugewiesen werden, wie beispielsweise die „prev"-Zeiger in Listen.

Um auch diese Zeiger während eines `dsm_malloc` registrieren zu können, müßte die Struktur, die die vom Anwender angeforderten Speicherbereiche erhalten, analysiert werden, um zu ermitteln, an welchen Adressen innerhalb dieser Speicherabschnitte Zeiger stehen. Dies könnte über die Analyse des an `dsm_malloc` übergebenen `size`-Parameters erfolgen. Anhand der Struktur eines komplexen, zeigerenthaltenden Variablen-Typs wie z.B. eines Listentyps, der dabei als Argument des `sizeof`-Operators übergeben wird, und des eventuell angebenden Multiplikators müßten die Zeigeradressen in diesem Speicherabschnitt bestimmt werden. Das bedeutet, daß dem Anwender verboten werden müßte, die Größe eines von ihm benötigten Speicherabschnitts, der Zeiger in bereits angeforderte Speicherabschnitte enthalten soll, selbst zu bestimmen und eine konkrete Anzahl an Bytes in den Größenparameter von `dsm_malloc` einzutragen. Die Entfernung der Zeiger aus der Zeigertabelle im Falle einer Freigabe dieses Speicherabschnitts ist etwas unkomplizierter, wenn auch mit nicht geringem Aufwand verbunden. Dies könnte erledigt werden, indem die Zeigertabelle nach Adressen, die innerhalb dieses Speicherabschnitts liegen, durchsucht wird und diese gegebenenfalls gelöscht werden.

Da die Zeigerregistrierung keinen allzu großen Anteil innerhalb dieser Arbeit beanspruchen soll, wird die Umsetzung der eben vorgeschlagene Lösungsansätze für die vollständige Registrierung der Zeiger in den DSM-Heap auf eine spätere Arbeit verschoben. In der jetzigen Implementierung ist es also nur möglich, Zeiger zu registrieren und damit zu übersetzen, die ihre Werte von `dsm_malloc` zugewiesen bekommen. Demzufolge muß dem Anwender verboten werden, andere als solche Zeiger in den DSM-Heap zu verwenden, sofern diese Zeiger auf allen Knoten gültig sein sollen.

Eine Sonderbehandlung der Zeiger der Elemente der Freelist ist auch mit dieser Zeigerregistrierungsmethode notwendig, was allerdings nichts mit ihren eben genannten Unzulänglichkeiten zu tun hat. Diese Zeiger können nicht über den gerade beschriebenen Mechanismus registriert werden, weil ihnen ihre Werte natürlich nicht via `dsm_malloc` zugewiesen werden. Darum müssen diese Zeiger quasi per Hand registriert und wieder freigeben werden, indem die entsprechenden Funktionsaufrufe an den richtigen Stellen in die Funktionen `kr_malloc`, `kr_free` und `kr_morecore` eingetragen werden.

5.2.2 Propagierung neuer Tabelleneinträge

Wird auf einem Knoten ein neuer Zeiger in der Tabelle registriert, so muß er auch auf den anderen Knoten registriert werden, damit er auf allen Knoten beim Versenden oder Empfangen der Seite auf der er sich befindet, entsprechend übersetzt wird. Zum Zeitpunkt des Empfangs einer Seite muß dieser Knoten die Adressen aller Zeiger kennen, die bisher von irgendwelchen Knoten auf dieser Seite eingerichtet worden sind. Das führt zu der Frage, auf welche Art und Weise die Informationen über neu registrierte Zeiger am besten verschickt werden. Diese Frage kann für die Zeigertabelle genauso beantwortet werden, wie sie für den Transfer der Heapinformationen (siehe Kapitel 3.3.2) beantwortet wurde.

Eine weitere Möglichkeit des Transfers der Informationen, die für die Heapinformationen nicht diskutiert wurde, soll hier kurz dargestellt werden. Die Zeigertabelle, wie auch die globalen Heapinformationen, könnte auch als gemeinsam genutzte Datenstruktur im DSM-Datenbereich angelegt werden. Da diese Informationen auf allen Knoten zur Verfügung stehen sollen, es sich also quasi um gemeinsame Daten handelt, liegt diese Lösung nahe. Ein Knoten, auf dem die Informationen benötigt werden, holt sich diese dann über den normalen DSM-Seitentransfermechanismus von dem Knoten, der ihr aktueller Besitzer ist, der also über die aktuellsten Informationen verfügt, indem er bei Bedarf auf sie zugreift.

Mit diesem Ansatz sind jedoch zwei Probleme verbunden. Zum einen resultiert aus ihm ein erhöhter Seitentransfer. Mit jedem normalen Transfer einer Seite, auf der sich das eigentlich benötigte Objekt befindet, ist der Transfer von mindestens einer weiteren Seite verbunden, auf der sich die Informationen über die Heapstruktur und die Zeigertabelle befinden. Da sich der Umfang dieser Informationen keinesfalls auf eine Seite beschränken muß, kann durchaus der Transfer von mehreren solcher Folgeseiten nötig sein.

Das zweite Problem ergibt sich daraus, daß die Informationen zur Heapstruktur und zur Zeigertabelle nur mit erheblichem Aufwand über die Speicherverwaltung des DSM-Heap in diesem angelegt werden können. Da der Umfang dieser Informationen nicht beschränkt werden darf, dies hätte nicht hinnehmbare Einschränkungen zur Folge, die dem Anwender bezüglich der Anzahl der verwendbaren Zeiger und der Größe des dynamischen DSM-Speichers auferlegt werden müßten, können diese Daten nicht im statischen DSM-Speicher stehen, sondern müßten im DSM-Heap angelegt werden. Dies ist jedoch über die in Kapitel 4 vorgestellte Speicherverwaltung des DSM-Heap nicht ohne weiteres möglich. Während der Ausführung der Funktionen zur DSM-Heapspeicherverwaltung zur Abarbeitung einer Speicheranforderung durch den Anwender wird die Zeigertabelle in jedem Fall und die Heapinformationen eventuell ergänzt. Dazu müßte ein neuer Speicherabschnitt aus dem DSM-Heap angefordert werden. Dies ist zu diesem Zeitpunkt aber nicht möglich, weil die rekursive Ausführung von `kr_malloc` nicht möglich ist, da sonst die Freelist zerstört werden würde. Die Erweiterung der Zeigertabelle und der globalen Heapinformationen wäre also erst nach der Beendigung der Bearbeitung der Anwenderanforderung durch `kr_malloc` möglich. Dies könnte zwar geschehen, indem die dabei angefallenen Daten über neue Zeiger und Heapblöcke vorerst zwischengespeichert werden. Aber da bei dem nachfolgenden `kr_malloc` wieder neue Zeiger entstehen und eventuell neue Heapblöcke erstellt werden, entsteht dabei ein Kreislauf, der sich nur mit recht großem Aufwand durchbrechen läßt. Zwar könnten die Zeigertabelle und die globalen Heapinformationen auch ohne die Speicherverwaltung im DSM-Heap angelegt werden, indem für sie Sonderseiten erstellt werden, doch auch dafür ist erheblicher Aufwand nötig.

Es spricht noch ein weiteres Problem dagegen, die globalen Heapinformationen im DSM-Speicher abzulegen. Diese Daten werden zur Abarbeitung eines Seitenzugriffsfehlers, z.B. während der Übersetzung der Zeiger auf einer gerade empfangenen Seite, benötigt. Würde es sich dabei um DSM-Daten handeln, würde daraus folgen, daß während der Abarbeitung eines Zugriffsfehlers auf einem Knoten durch den Zugriff auf diese Daten ein weiterer Zugriffsfehler auf diesem Knoten ausgelöst werden würde, sofern dieser Knoten nicht zufällig gerade der Besitzer der Seiten mit diesen Daten ist. Solche geschachtelten Seitenzugriffsfehler sind in der gegenwärtigen Implementation der DSM-Threads nicht vorgesehen. Man müßte also die DSM-Threads entsprechend erweitern, was mit einem nicht gerechtfertigtem Aufwand verbunden wäre, da diese Erweiterung sonst nicht benötigt wird. Aufgrund der genannten Probleme wurde dieser Lösungsansatz zur Informations-Propagierung hier sowohl bezüglich der Heapinformationen als auch der Zeigertabelle verworfen.

Es gibt über diese und die in Kapitel 3.3.2 genannten Möglichkeiten hinaus noch weitere Möglichkeiten, neue Einträge in der Zeigertabelle eines Knotens den anderen Knoten bekannt zu machen. Als am einfachsten, elegantesten und effizientesten wird hier angesehen, die Informationen zur Zeigertabelle über denselben Mechanismus zu verbreiten, wie die Informationen zur Heapstruktur (siehe Kapitel 3.3.2). Im Gegensatz zu den Heapinformationen muß dabei nicht jedesmal die komplette Zeigertabelle mit einer Seite mitgeschickt werden, sondern nur die Anzahl der Zeiger, die sich auf der zu verschickenden Seite befinden, und deren Adressen.

Um die betreffenden Informationen nicht bei jedem Seitentransfer umständlich aus der Zeigertabelle heraussuchen zu müssen, wird diese folgendermaßen strukturiert: die Zeigertabelle enthält für jede Seite einen Eintrag, wobei die einzelnen Seiten in der Zeigertabelle den gleichen Index haben wie in der Seitentabelle, und jeder dieser Seiteneinträge enthält die Anzahl und die Adressen aller Zeiger, die sich auf dieser Seite befinden. Es werden dann, wann immer eine Seite von einem Knoten verschickt wird, die Informationen, die in der Zeigertabelle zu dieser Seite gespeichert sind, in der Nachricht mitgeschickt, von der die Seite übertragen wird. Dazu werden sie in denselben Puffer kopiert wie die Heapinformationen und dann in die Nachricht eingefügt. Da die Adressen der einzelnen Zeiger, wie ihre Werte, auf den unterschiedlichen Knoten variieren können, werden sie vor dem Kopieren in eine global gültige Normalform übersetzt (Funktion `dsm_ToGlobalAddr` in Abschnitt 5.3). Auf einem Knoten, auf dem eine Seite empfangen wird, wird der betreffende Seiteneintrag in der Zeigertabelle anhand der mitgesandten Zeigeradressen aktualisiert, nachdem diese in lokal gültige Werte übersetzt worden sind (Funktion `dsm_ToLocalAddr` in Abschnitt 5.3).

Die Größe der Nachrichten, mit denen DSM-Seiten verschickt werden, nimmt auch durch das zusätzliche Mitschicken der Zeigerinformationen nicht wesentlich zu. Im ungünstigsten Fall, wenn auf einer gesamten DSM-Seite nichts anderes als Zeiger stehen, wird die Nachricht um soviel Bytes, wie eine DSM-Seite groß ist, für das Einfügen der Zeigeradressen (für eine Zeigeradresse werden im Speicher wie in der Zeigertabelle 4 Bytes benötigt) zuzüglich 4 Bytes für die Anzahl der Zeiger vergrößert. Doch das Eintreten dieses Falls ist sehr unwahrscheinlich, so daß im Normalfall für die Zeigeradressen weitaus weniger Platz in der Nachricht benötigt wird.

Auch hier stellt sich die Frage, ob die verschickten Informationen aus der Zeigertabelle vollständig sind, ob sie also immer die Adressen aller Zeiger enthalten, die bisher von irgendwelchen Knoten auf der betreffenden Seite erstellt wurden. Dabei gilt folgende Einschränkung: ein Knoten benötigt Informationen über einen von anderen Knoten erstellten Zeiger erst, wenn von ihm auf diesen Zeiger zugegriffen wird. Das bedeutet, die Zeigerinformationen, die mit einer Seite mitgeschickt werden, müssen nicht unbedingt alle Zeiger beinhalten, die bisher erstellt worden sind, sondern nur jene, auf die auf dem seitenempfangenden Knoten zugegriffen werden soll. Darüber kann der sendende Knoten natürlich nichts wissen, trotzdem spielt diese Einschränkung eine wichtige Rolle. Es kann immer nur ein Knoten der aktuelle, schreibberechtigte Besitzer einer bestimmten Seite sein (innerhalb der DSM-Threads können bisher ausschließlich die Seitenzugriffsprotokolle „SingleReaderSingleWriter“ oder „MultiReaderSingleWriter“ verwendet werden), und nur auf diesem Knoten können neue Zeiger auf dieser Seite erstellt werden. Werden diese Zeiger in der Zeigertabelle dieses Knotens registriert, bevor er diese Seite verschickt, so sind sie in jedem Fall in den Zeigerinformationen bei einem darauffolgenden Verschicken der Seite enthalten und werden dann von Seitenbesitzer zu Seitenbesitzer weitergegeben. Es kann jedoch passieren, daß die Seite verschickt wird, bevor alle neuen Zeiger registriert wurden. Dabei muß zwischen den Zeigern der Freelist und den vom Anwender erstellten Zeigern unterschieden werden.

Wenn die Seite, auf der während der Bearbeitung der Freelist ein neuer Zeiger erstellt wird, verschickt wird, bevor dieser Zeiger registriert werden konnte, dann geschieht das in keinem Fall, weil auf einem anderen Knoten ebenfalls auf die Freelist zugegriffen wurde, denn der Mutex, der diesen Zugriff regelt, ist noch besetzt (siehe dazu Abbildung 5.1). Also werden die Informationen über den neuen Zeiger in der Freelist auf den anderen Knoten noch nicht benötigt. Wird der Zeiger dann auf dem Knoten registriert, auf dem die Freelist gerade bearbeitet wird, so wird auf diesem Knoten, dazu zwingt der Ablauf innerhalb der Speicherverwaltungsfunktionen, unbedingt noch einmal schreibend auf den Zeiger zugegriffen. Also wird dieser Knoten noch einmal Besitzer der Seite mit dem neuen Zeiger und propagiert bei einem folgenden Besitzerwechsel dieser Seite den neuen Zeiger. Das bedeutet, erst nachdem ein Knoten den Heap-Mutex freigegeben hat, werden die Informationen über solche neuen Zeiger benötigt. Zu diesem Zeitpunkt sind sie unbedingt registriert und werden propagiert.

Im Falle der Anwender-Zeiger, d.h. der Zeiger, denen der Anwender mittels `dsm_malloc` einen Speicherabschnitt aus dem DSM-Heap zuweisen will, verhält es sich ähnlich. Diese Zeiger werden, unmittelbar bevor der Heap-Mutex

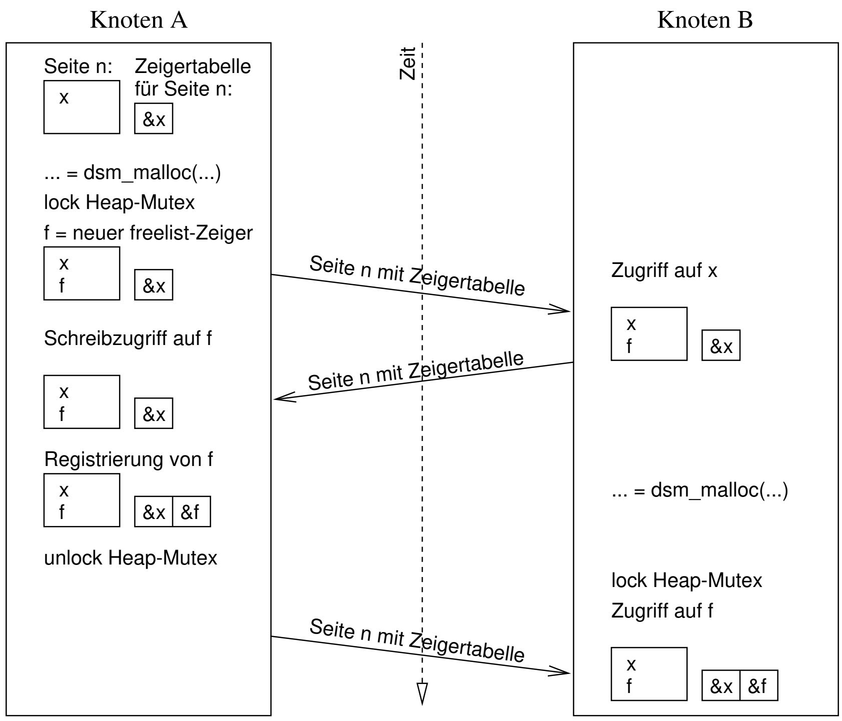


Abbildung 5.1: Rechtzeitige Propagierung von Informationen über neue Zeiger

freigegeben und mit dem Wert des Zeigers zum Anwendungsprogramm zurückgekehrt wird, von `dsm_malloc` registriert. Hat dieser Knoten die Seite, auf der sich der Anwender-Zeiger befindet, zwischen dem Aufruf von `dsm_malloc` und der Registrierung dieses Zeigers bereits wieder verschickt, so muß er noch einmal Besitzer dieser Seite werden, um dem Anwender-Zeiger den Rückgabewert von `dsm_malloc` zuweisen zu können. Ab diesem Zeitpunkt wird die Information über den neuen Zeiger ordnungsgemäß propagiert, und erst ab diesem Zeitpunkt ist dies nötig, denn erst ab dann hat der Zeiger einen Wert, auf den zugegriffen werden kann. Wird vorher vom Anwender auf diesen Zeiger zugegriffen, so geschieht das, weil dieser die Reihenfolge von Wertzuweisung und Zugriff auf den Wert nicht so gesteuert hat, wie es sich in einem verteilten, nebenläufigen System bzgl. sequentieller Konsistenz gehört. Das Ergebnis eines solchen verfrühten Zugriffs ist undefiniert, und die bis dahin unterbliebene Übersetzung des Zeigerwertes spielt keine Rolle. Bezüglich der Wiederfreigabe der Zeiger aus der Zeigertabelle gelten die gleichen Bedingungen.

5.2.3 Konsistenz der Zeigertabelle

Wie die globalen Heapinformationen und die Freelist muß die Zeigertabelle vor gleichzeitiger Bearbeitung durch mehrere Knoten geschützt werden. Bezüglich dessen gelten die in Kapitel 4.2.1 gemachten Aussagen über die vom Heap-Mutex mitgelieferte Garantie des Ausschlusses der gleichzeitigen Bearbeitung jeglicher zur Verwaltung des DSM-Heap notwendiger Daten auch für die Zeigertabelle.

Denn Zeiger werden ausschließlich auf dem Knoten, der gerade den Heap-Mutex besitzt, in der Zeigertabelle registriert oder aus ihr freigegeben.

Weiterhin muß dafür Sorge getragen werden, daß die Zeigertabelle zum Zeitpunkt ihrer Veränderung sämtliche Veränderungen enthält, die zuvor auf irgendwelchen Knoten an ihr vorgenommen wurden. Genauer gesagt muß, wenn auf einem Knoten ein Zeiger registriert oder freigegeben werden soll, der betreffende Seiteneintrag in der Zeigertabelle dieses Knotens auf dem aktuellen Stand sein, d.h., er muß alle bisherigen Registrierungen und Freigaben beinhalten. Diese Aktualität der Zeigertabelle zum Zeitpunkt ihrer Veränderung ist jedoch im Gegensatz zu den globalen Heapinformationen nicht in jedem Fall automatisch garantiert, da ein Knoten nicht zwingend der Besitzer der Seite sein muß, für die er einen Zeiger registrieren oder freigeben will.

Um die Vollständigkeit der Heapinformationen zu gewährleisten ist es hinreichend, wenn der Knoten, auf dem diese ergänzt werden sollen, eine beliebige DSM-Seite von dem Knoten geschickt bekommt, auf dem die Heapinformationen zuletzt bearbeitet werden konnten. Dies ist über den Zugriff auf die Freelist gesichert (siehe Kapitel 3.3.2). Um auf einem Knoten die Aktualität eines Seiteneintrags in der Zeigertabelle sicherzustellen, müßte dieser Knoten nicht eine beliebige, sondern genau die betreffende Seite von dem Knoten geschickt bekommen, auf dem dieser Seiteneintrag zuletzt verändert wurde. Und zwar müßte dies geschehen, während der erstere Knoten den Heap-Mutex besitzt, denn erst dann kann die Zeigertabelle auf keinem anderen Knoten mehr verändert werden.

Diese Bedingung wird aber nicht in jedem Fall automatisch erfüllt. Dazu folgendes Beispiel. Der Anwender führt `x=dsm_malloc(sizeof(list_t),&x)` auf einem Knoten A aus (Abbildung 5.2, oben). Selbst wenn Knoten A zu diesem Zeitpunkt der Besitzer der Seite ist, die `&x` enthält, bedeutet das nicht, daß er auch noch zu dem Zeitpunkt über die aktuellen Daten bezüglich dieser Seite in der Zeigertabelle verfügt, an dem auf ihm `&x` registriert werden soll. Denn bevor auf Knoten A der mit dem Heap-Mutex geschützte Bereich in `dsm_malloc` betreten wird, kann es vorkommen, daß diese Seite, z.B. während auf Knoten A auf den Heap-Mutex gewartet wird, zu einem Knoten B transferiert wird, der vielleicht gerade den Heap-Mutex besitzt, und daß dann auf Knoten B der entsprechende Seiteneintrag in der Zeigertabelle verändert wird. Bekommt Knoten A dann den Heap-Mutex, ist es keinesfalls gesichert, daß er noch einmal auf die Seite zugreift, die `&x` enthält, bevor `&x` registriert wird. In diesem Fall erfährt er auch nichts über eventuell inzwischen von anderen Knoten vorgenommene Zeigerregistrierungen oder -freigaben bezüglich dieser Seite. Dies führt zu Inkonsistenzen in den Zeigertabellen auf den verschiedenen Knoten, deren einzelne Variationen hier nicht diskutiert werden sollen, da keine von ihnen geduldet werden kann. Für die Freigabe eines Zeigers aus der Zeigertabelle während der Ausführung von `dsm_free` ergibt sich das Problem in gleichem Maße.

Es kann auf einfache Art und Weise gelöst werden, indem innerhalb der vom Heap-Mutex geschützten Bereiche in `dsm_malloc` und `dsm_free` mittels der übergebenen Adresse des Zeigers auf die betreffende Seite zugegriffen wird, bevor diese Adresse in der Zeigertabelle eingetragen bzw. aus ihr entfernt wird (Abbildung 5.2, unten). Dann bekommt dieser Knoten diese Seite, einschließlich der letzten sie betreffenden Änderungen in der Zeigertabelle, noch einmal geschickt. Also wird der entsprechende Seiteneintrag in der Zeigertabelle dieses Knotens noch einmal aktualisiert, bevor ein auf dieser Seite befindlicher Zeiger auf diesem Knoten registriert oder freigeben wird.

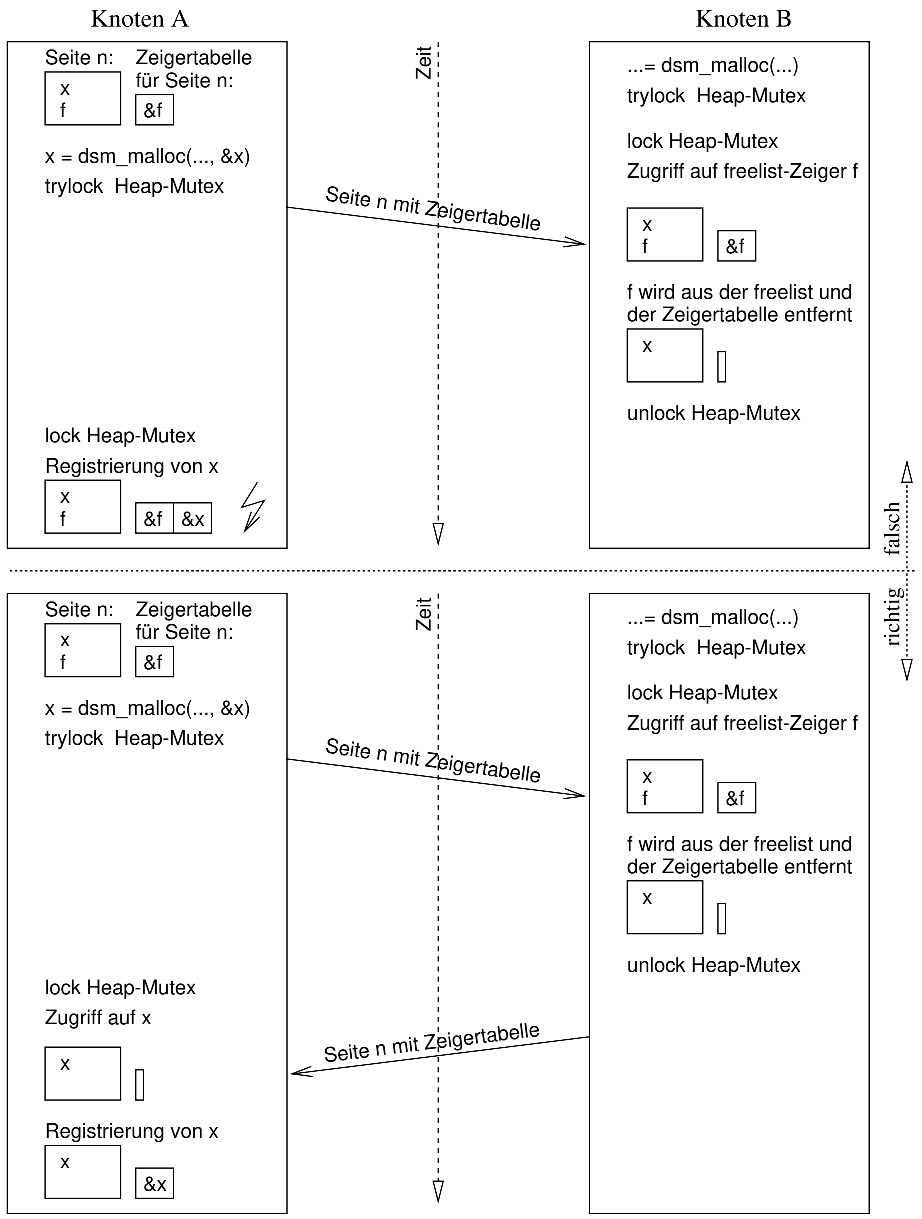


Abbildung 5.2: Aktualität der Zeigertabelle zum Zeitpunkt ihrer Veränderung

Die Konsistenz der Zeigertabelle zu gewährleisten beinhaltet auch, dafür zu sorgen, daß kein Zeiger mehrfach in ihr registriert wird. Bei einem Seitenempfang werden die in der empfangenen Seite stehenden global gültigen Werte der einzelnen Zeiger in lokal gültige Werte übersetzt (siehe Abschnitt 5.3) und dann mit diesen überschrieben. Ist ein Zeiger mehr als einmal in der Zeigertabelle registriert, so führt das dazu, daß dabei versucht wird, einen bereits übersetzten Zeigerwert noch einmal zu übersetzen. Daraus resultiert unbedingt ein Segmentation-Fehler und das DSM-System stürzt ab.

Derartige Mehrfachregistrierungen eines Zeigers können ausschließlich geschehen, wenn der Anwender nicht die richtige Reihenfolge bei dem Aufruf von `dsm_malloc` und `dsm_free` einhält, und sei es nur, weil er nicht in genügendem Maße die Bedingungen eines verteilten, nebenläufigen Systems bezüglich des parallelen Ablaufs seiner Programmteile bedacht hat. Will der Anwender z.B. auf einem Knoten `x=dsm_malloc(...,&x)`, etwas später auch auf diesem Knoten `dsm_free(x,&x)` und danach auf einem weiteren Knoten wieder `x=dsm_malloc(...,&x)` ausführen, und er stellt diese Ausführungsreihenfolge nicht über geeignete Werkzeuge zur parallelen Programmierung sicher, so kann es beispielsweise dazu kommen, daß erst zweimal `x=dsm_malloc(...,&x)` ausgeführt wird und dann `dsm_free(x,&x)`. Das führt dazu, daß `&x` zweimal in der Zeigertabelle registriert wird, was zum Absturz des DSM-Systems führt.

Obwohl dieser Absturz aus einem Fehler des Anwenders resultieren würde, soll diesem das nicht zugemutet werden, da er die Ursache des Absturzes in dieser Situation nur schwer nachvollziehen kann. Darum wird etwas Fehlerfreundlichkeit implementiert, indem in der Zeigertabelle vor jeder Registrierung eines Zeigers kontrolliert wird, ob dieser nicht bereits registriert ist.

Die mehrfache Hintereinanderausführung von `dsm_free` für ein und denselben Zeiger schadet nichts, da ein Zeiger sowieso nur aus der Zeigertabelle freigegeben werden kann, wenn er noch in ihr registriert ist, d.h., eine mehrfache Freigabe eines Zeigers ist von vornherein nicht möglich.

In diesem Zusammenhang soll, auch wenn dies nichts mit der Zeigerübersetzung zu tun hat, auf ein weiteres Problem hingewiesen werden, das entstehen kann, wenn der Anwender nicht die richtige Reihenfolge von `dsm_malloc` und `dsm_free` beachtet. Genauer gesagt kann dieses Problem auftreten, wenn `dsm_realloc`, welches wiederum `dsm_malloc` und `dsm_free` aufruft, mehrmals gleichzeitig für ein und denselben Zeiger aufgerufen wird.

Auch hierzu folgt ein Beispiel. Ohne dies über einen Mutex oder ähnliche Werkzeuge zu steuern, will der Anwender `x=dsm_realloc(x,size,&x)` auf Knoten A und dieselbe Anweisung etwas später auch auf Knoten B ausführen (Abbildung 5.3, oben). Dann kann die Ausführung beider Anweisungen zeitlich so zusammenfallen (Abbildung 5.3, Mitte), daß `dsm_realloc` erst auf Knoten A aufgerufen wird, auf Knoten A dann auf den in `dsm_realloc` ebenfalls angewendeten Heap-Mutex gewartet wird, dann auf Knoten B `dsm_realloc` aufgerufen und ebenfalls auf den Heap-Mutex gewartet wird. Bekommt Knoten A dann den Heap-Mutex, so wird auf ihm via `dsm_malloc` ein neuer Abschnitt aus dem DSM-Heap angefordert, der Inhalt des Objekts, auf das der übergebene Zeiger zeigt, in diesen Abschnitt kopiert, dann `dsm_free` für diesen Zeiger aufgerufen, der Heap-Mutex freigegeben und bei der Rückkehr von `dsm_realloc` der neue Speicherabschnitt `x` zugewiesen. Bekommt dann Knoten B den Heap-Mutex, findet der gleiche Ablauf noch einmal statt, und zwar mit dem bereits an `dsm_realloc` übergebenen alten Wert von `x`, obwohl der Anwender beabsichtigte, `dsm_realloc` für den neuen Wert von `x` auszuführen, den `x` auf Knoten A erhalten hat. Das bedeutet, daß versucht wird den Speicherabschnitt, auf den `x` anfänglich zeigte, noch einmal freizugeben. Da dieser jedoch bereits auf Knoten A freigegeben wurde, enthält ihn die Freelist bereits. Um einen Speicherabschnitt in die nach den Adressen der Abschnitte geordnete Freelist aufzunehmen, wird diese solange durchsucht, bis zwei, in ihr benachbarte, Einträge gefunden sind, zwischen die der neue Abschnitt eingefügt werden kann. In dem hier besprochenen Beispiel kann diese Suche zu keinem Ende kommen, da kein Zwischenraum für den freizugebenden Abschnitt gefunden werden kann, weil es bereits einen Eintrag mit dieser Adresse in ihr gibt. Das Ergebnis des Ganzen ist eine Endlosschleife.

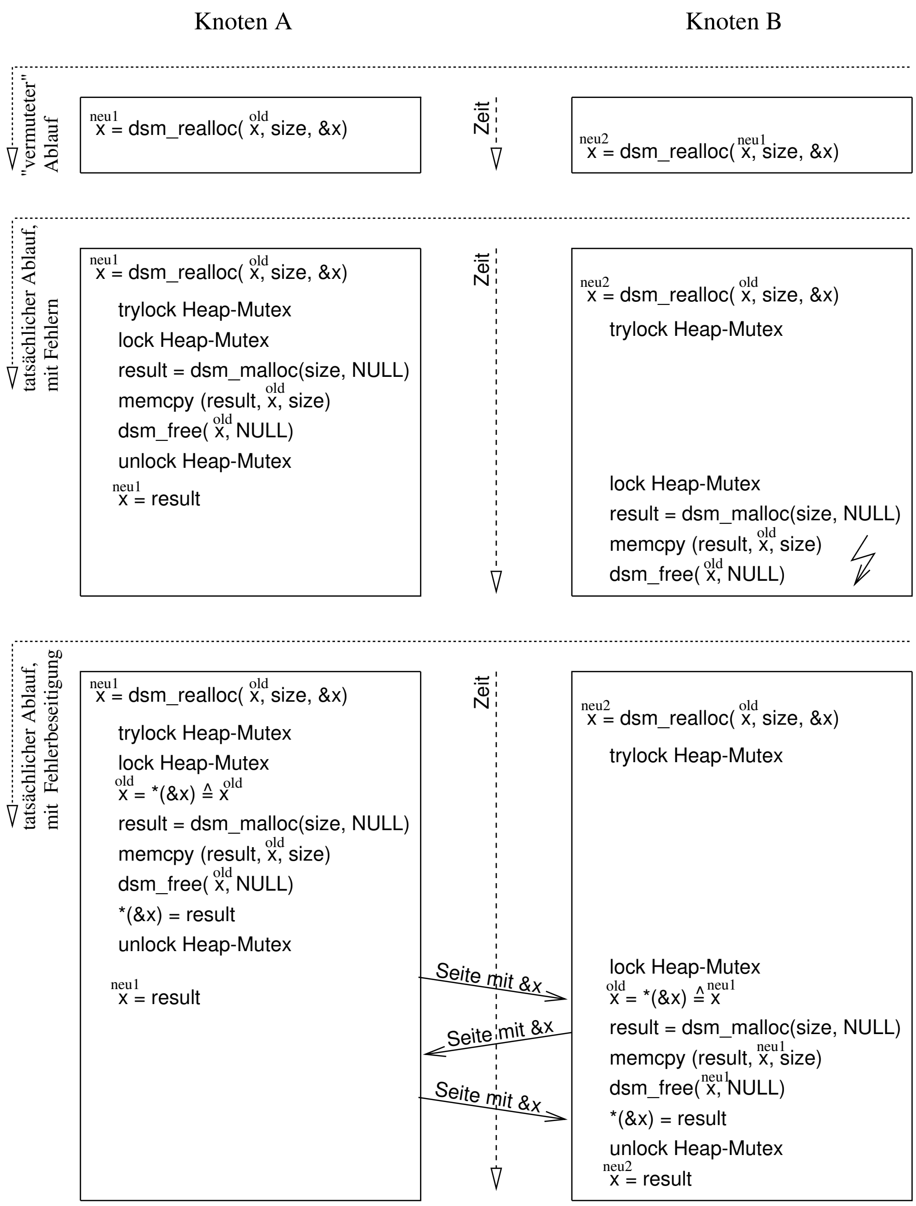


Abbildung 5.3: Parallele Ausführung mehrerer `dsm_realloc`-Aufrufe

Da der Anwender auch in dieser Situation nicht mit einer derartigen Reaktion des Systems rechnen kann, soll er nicht mit diesem Problem konfrontiert werden, obwohl auch dieses Problem ausschließlich entstehen kann, wenn der Anwender nicht mit genügender Sorgfalt den parallelen Ablauf seiner Programmteile (in dem Fall den Aufruf von `dsm_realloc`) in einem verteilten, nebenläufigen System gesteuert hat. Um es zu lösen, wird in `dsm_realloc`, nachdem der Mutex-geschützte Bereich betreten wurde, der übergebene Zeiger noch einmal aktualisiert, bevor `dsm_free` usw. ausgeführt wird. Dazu wird der aktuelle Wert des Zeigers über seine ebenfalls übergebene Adresse ermittelt und ihm zugewiesen. Durch den Zugriff auf die Adresse des Zeigers bekommt dieser Knoten die Seite, die diese Adresse beinhaltet, und damit auch den aktuellen Wert des Zeigers (Abbildung 5.3, unten). Zusätzlich wird am Ende von `dsm_realloc`, bevor der Heap-Mutex wieder abgegeben wird, dem Zeiger bereits sein neuer Wert zugewiesen. Damit wird verhindert, daß diese Seite zwischen der Mutexrückgabe und der Zuweisung des Rückgabewertes von `dsm_realloc` an den Zeiger noch mit dem alten Wert des Zeigers verschickt wird.

5.3 Adreßübersetzung

Es wurde bereits erwähnt und dies begründet, daß die Werte aller Zeiger auf einer zu transferierenden bzw. transferierten DSM-Seite und die mitzuschickenden bzw. mitgeschickten Adressen dieser Zeiger in eine „global“ bzw. „lokal“ gültige Form, im folgenden globale bzw. lokale Adresse genannt, übersetzt werden müssen. Eine globale Adresse ist auf allen Knoten gültig, eine lokale gilt nur auf einem bestimmten Knoten. Solch eine globale Adresse ist keine tatsächliche Adresse, sondern eine ganzzahliger, positiver Wert, der lediglich eine Adresse im DSM-Speicher beschreibt. Er beinhaltet den Seitentabellenindex einer DSM-Seite und einen Offset zu der Anfangsadresse dieser Seite. Dieser Wert kann auf allen Knoten in eine lokal gültige Adresse umgerechnet werden. Eine lokale Adresse ist die tatsächliche Adresse der bezeichneten DSM-Speicherstelle auf einem bestimmten Knoten.

Die Adressen werden mittels der beiden in Listing 5.1 gezeigten Funktionen übersetzt. `dsm_ToGlobalAddr` übersetzt eine lokal gültige Adresse in eine global gültige und `dsm_ToLocalAddr` umgekehrt. Dabei bezeichnet `DSM_PAGESIZE` die Größe der DSM-Seiten in Bytes, `dsm_GetPageIndex(addr)` liefert den Seitentabellenindex der Seite, die die Adresse `addr` enthält, und `dsm_GetPage(index)` liefert die Anfangsadresse der Seite mit dem Seitentabellenindex `index`.

Listing 5.1 Funktionen zur Adreßübersetzung

```
int dsm_ToGlobalAddr(int localaddr)
{
  int pageindex = dsm_GetPageIndex(localaddr);
  return (localaddr - dsm_GetPage(pageindex) + pageindex * DSM_PAGESIZE);
}

int dsm_ToLocalAddr(int globaladdr)
{
  int pageindex = globaladdr / DSM_PAGESIZE;
  return (globaladdr - pageindex * DSM_PAGESIZE + dsm_GetPage(pageindex));
}
```

In `dsm_ToGlobalAddr` wird erst der Seitentabellenindex der Seite ermittelt, auf der sich die zu übersetzende Adresse befindet, dann der Offset dieser Adresse zur Anfangsadresse dieser Seite berechnet, indem die Seitenanfangsadresse von der zu übersetzenden Adresse subtrahiert wird, dann auf den Offset das Produkt des Seitentabellenindex mit der Seitengröße addiert und letztlich dieser Wert als globale Adresse zurückgegeben (siehe Abbildung 5.4, linke Seite).

In `dsm_ToLocalAddr` wird der Index der Seite ermittelt, die die lokal gültige Adresse beinhaltet, indem die übergebene globale Adresse durch die Seitengröße dividiert wird, dann der Offset dieser Adresse zum Seitenanfang berechnet, indem das Produkt des Seitenindex mit der Seitengröße von der Adresse subtrahiert wird, dann zum Offset die Seitenanfangsadresse addiert und dieser Wert als lokale Adresse zurückgegeben (siehe Abbildung 5.4, rechte Seite).

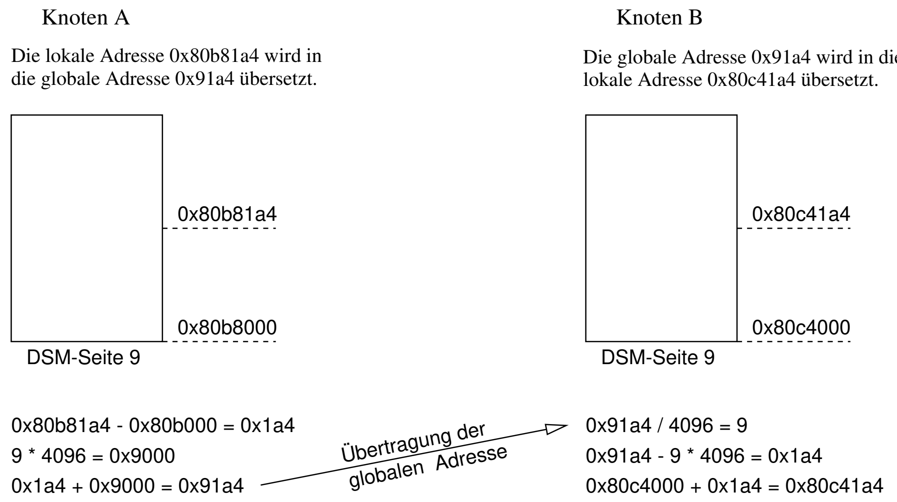


Abbildung 5.4: Übersetzung einer DSM-Adresse vor und nach ihrer Übertragung

Vor dem Versenden einer Seite wird, nachdem diese Seite bereits in die seitenübertragende Nachricht kopiert wurde, der lokal gültige Wert eines jeden Zeigers, der in der Zeigertabelle für diese Seite gespeichert ist, in eine global gültige Adresse übersetzt und mit dieser der lokale Wert des Zeigers in der Nachricht überschrieben. Das heißt, die globalen Werte der Zeiger werden direkt an ihre entsprechenden Adressen in der bereits im Nachrichtenpuffer befindlichen Seite kopiert und so transferiert. Beim Empfangen einer Seite wird, nachdem die Seite bereits in den DSM-Speicher kopiert wurde und die Heapinformationen und die Zeigertabelle aktualisiert wurden, mittels der in der Zeigertabelle für diese Seite gespeicherten Zeigeradressen für jeden dieser Zeiger zunächst dessen globaler Wert aus der Seite gelesen, dann der Seitenindex der Seite, auf die er verweist, ermittelt und überprüft, ob die Seite bereits existiert. Ist das nicht der Fall, so wird der DSM-Heap auf diesem Knoten entsprechend vervollständigt. Dann wird der globale Wert des Zeigers in eine lokal gültige Adresse übersetzt und mit diesem Wert überschrieben.

An welcher Stelle die Adressen der Zeiger übersetzt und wie die übersetzten Adressen transferiert werden, wurde bereits in Abschnitt 5.2.2 beschrieben.

Zusammenfassung

In dieser Arbeit wurde ein Modell zur dynamischen Verwaltung von gemeinsamem Speicher in einem homogenen, verteilten System entwickelt und innerhalb der DSM-Threads implementiert. Der verteilte, gemeinsam genutzte DSM-Speicher wird während der Laufzeit entsprechend den Bedürfnissen des Anwenders vergrößert, indem einzelne Speicherblöcke vom Betriebssystem angefordert und zum dynamischen DSM-Speicher hinzugefügt werden.

Mit den Funktionen bzgl. des DSM-Heap aus der Anwenderschnittstelle der DSM-Threads (siehe Anhang A) stehen dem Anwender Möglichkeiten zur Verfügung, über die er DSM-Speicher in gewohnter Weise anfordern und wieder freigeben kann. Dies kann er für beliebig viele und beliebig große Abschnitte dynamischen DSM-Speichers tun. Bezüglich der Objekte, die im DSM-Speicher angelegt werden können, gibt es keinerlei Einschränkungen, wobei auch die Nutzung von gemeinsamen Zeigern möglich ist. Als einzige Bedingung muß dem Anwender auferlegt werden, daß er allen Zeigern, die in den dynamischen DSM-Speicher verweisen sollen, einzeln über eine der entsprechenden Funktionen aus der Schnittstelle zum DSM-Heap Speicher zuweist, da in der vorliegenden Implementation nur so die automatische Registrierung aller Zeiger in den DSM-Heap gewährleistet ist. Die Funktionen zur Zeigerregistrierung und -freigabe könnten auch dem Anwender selbst zu Verfügung gestellt werden, womit eine noch flexiblere Anwendung von Zeigern in den DSM-Heap ermöglicht werden würde, aber auch das Risiko einer unkorrekten Anwendung dieser Funktionen entsteht.

Wird kein dynamischer DSM-Speicher benötigt, d.h., der Anwender fordert solchen an keiner Stelle seines Programms an, so treten keinerlei bzw. nur sehr geringe Performance-Einbußen der DSM-Threads mit implementierter dynamischer Speicherverwaltung gegenüber den DSM-Threads ohne diese auf. Schließlich werden ausschließlich dann Heapblöcke erstellt, Zeiger registriert, Heap- und Zeigerinformationen verschickt, Zeiger übersetzt usw., wenn vom Anwender während der Laufzeit DSM-Speicher angefordert wird. Geschieht das nicht, so werden während eines Seitentransfers zwar die Funktionen zum Transfer der Heap- und Zeigerinformationen und der Zeigerübersetzung aufgerufen, diese aber sofort wieder verlassen, da dort als erstes, und zwar mit nur sehr geringem Aufwand, geprüft wird, ob überhaupt entsprechende Informationen bzw. Zeiger existieren.

Wenn jedoch dynamischer DSM-Speicher benutzt wird, so müssen gewisse Performance-Einbußen innerhalb der DSM-Threads in Kauf genommen werden. Für die Erstellung und Verwaltung der Informationen bezüglich der Struktur des DSM-Heap und der Zeiger sowie für die Zeigerübersetzung und für die Verwaltung des Speichers im DSM-Heap ist nur ein recht geringer Aufwand nötig. Auch der Transfer der Heapinformationen und der Zeigeradressen zwischen den Knoten fällt nicht weiter ins Gewicht, da dafür keinerlei zusätzliche Nachrichten versendet werden müssen, sondern nur die Nachrichten, die sowieso bei einem Seitentransfer entstehen, etwas vergrößert werden. Es wurde bereits erwähnt, daß die Nachrichten dabei nicht in sehr starkem Maße wachsen. Für den Transfer der Zeigeradressen wird im ungünstigsten, aber zugleich auch unwahrscheinlichsten Fall, der Platz einer ganzen DSM-Seite in der Nachricht benötigt. Und erst bei 511 vorhandenen größeren Heapblöcken wird noch einmal ebensoviel Platz beansprucht (wenn eine DSM-Seite 4096 Bytes groß ist). Aber selbst wenn die Nachricht um die Menge von zwei DSM-Seiten größer wird, ist dies immer noch besser, als wenn statt dessen beispielsweise zwei extra Nachrichten gesandt werden müßten.

Schwerer wiegt der nicht unerhebliche Aufwand für die Erstellung eines Heapblocks, der durch das Anfordern von Speicher vom Betriebssystem und durch das `realloc` für die Vergrößerung der Seitentabelle entsteht. Der Aufwand für das Bereitstellen eines Speicherabschnitts durch das Betriebssystem wird dabei durch das Anfordern größerer Blöcke, d.h. dadurch, daß nicht für jedes Objekt einzeln Speicher angefordert wird, bereits weitgehend reduziert und kann vom Anwender durch eine sinnvolle, auf die Erfordernisse seines Programms abgestimmte Auswahl der Standardgröße für die Heapblöcke weiter minimiert werden. Auch der Aufwand für das `realloc` der Seitentabelle, der dabei jedesmal hauptsächlich durch das notwendige Kopieren der kompletten Seitentabelle entsteht, kann vermindert werden. Dies kann z.B. erfolgen, indem die Seitentabelle bei der Initialisierung des DSM-Systems und bei nachfolgenden Vergrößerungen bereits mit einer gewissen Anzahl zusätzlicher, freibleibender Einträge für eventuell später neu entstehende Seiten erstellt wird, und so die Anzahl der notwendigen Neuerstellungen der Seitentabelle reduziert wird.

Am negativsten wirkt sich auf die Performance jedoch aus, daß jedesmal, wenn ein Objekt im DSM-Heap angelegt oder eines wieder freigegeben werden soll, die Freelist, deren Elemente im DSM-Heap liegen, durchsucht werden muß. Das heißt, jedes Anlegen bzw. Freigeben eines Objekts im DSM-Heap zieht den Transfer von DSM-Seiten nach sich, deren Anzahl lediglich von der Verteilung der Elemente der Freelist im DSM-Heap abhängt und bis zu welcher Stelle sie durchsucht werden muß. Im schlimmsten Fall, dessen Eintreten jedoch nicht unwahrscheinlich ist, müssen sämtliche Seiten aus dem DSM-Heap zuzüglich der Seite im statischen DSM-Datenbereich, auf der der Kopf der Freelist ist, zu dem Knoten, auf dem ein Objekt im DSM-Heap erstellt bzw. freigegeben werden soll, transferiert werden. Dieser Fall tritt ein, wenn auf jeder Seite im DSM-Heap ein Element der Freelist steht und sie vollständig durchsucht werden muß. Dieses Performance-Problem kann lediglich vermieden bzw. vermindert werden, wenn die Verteilung der Elemente der Freelist im DSM-Heap eingeschränkt wird. Dies wiederum kann nur erreicht werden, indem die Größenangabe und die „next"-Zeiger in den Elementen der Freelist nicht, wie bisher, am Anfang der freien Speicherabschnitte selbst stehen, sondern an anderer Stelle abgelegt werden, z.B. auf wenigen, extra dafür eingerichteten Seiten im DSM-Heap. Dann muß nicht mehr auf die im DSM-Heap verteilten, freien Speicherabschnitte selbst zugegriffen werden.

Abzüglich des zuletzt genannten Problems, könnten all diese Performance-Nachteile abgemildert oder ganz vermieden werden, wenn die Gesamtgröße des DSM-Heap bereits während der Initialisierungsphase bekannt wäre. Es wäre weder die Speicherung von Informationen über die Heapstruktur bzw. über Zeiger und damit auch nicht deren Transfer, noch die Vergrößerung der Seitentabelle oder eine Zeigerübersetzung notwendig (siehe dazu auch Kapitel 2.1). Die Möglichkeit einer festen, vor der Laufzeit festgelegten Gesamtgröße des DSM-Heap könnte darum zusätzlich implementiert werden. Dem Anwender könnte dann die Wahl überlassen werden, ob er einen DSM-Heap unbeschränkter Größe oder wegen der höheren Ausführungsgeschwindigkeit einen DSM-Heap mit fester Größe benutzen will. Dann muß er allerdings in der Lage sein, die von ihm maximal benötigte Menge an dynamischem DSM-Speicher vor der Laufzeit abschätzten zu können.

Als ein Manko der vorliegenden Implementation des dynamischen DSM-Speichers ist anzuführen, daß kein Speicher aus dem DSM-Heap während der Laufzeit an das Betriebssystem zurückgegeben wird. Obwohl die dazu in Kapitel 4.1 gemachten Anmerkungen gelten, kann sich dies bei einzelnen Anwendungen recht negativ auswirken. Das Ziel dieser Arbeit bestand jedoch nicht in erster Linie darin, eine möglichst effiziente Verwaltung von dynamischen DSM-Speicher zu entwickeln, die eine Rückgabe von nicht mehr benötigtem DSM-Speicher an das Betriebssystem beinhalten müßte, sondern diesen überhaupt erst einmal zur Verfügung zu stellen. Darum kann dieser Nachteil vorerst akzeptiert werden und die Anwendung der Speicherverwaltungsmethode nach Kernighan und Ritchie als vorerst durchaus zweckmäßig bezeichnet werden.

Die Beschränkung der hier vorgestellten Implementation des gemeinsamen dynamischen Speichers auf ein homogenes, verteiltes System, kann dagegen nicht für eine endgültige Lösung hingenommen werden. Darum soll die Entwicklung des dynamischen DSM-Speichers mit dem Ziel seiner Anwendbarkeit in einem heterogenen System im Rahmen einer Diplomarbeit fortgesetzt werden.

Anhang A

Schnittstellenbeschreibung

`void dsm_heap_size(size_t size)`
Setzt die Größe des ersten DSM-Heapblocks auf `(size/DSM_PAGESIZE+1)` Speicherseiten. Wird `dsm_heap_size` nicht aufgerufen, wird dafür ein Standardwert genommen. `dsm_heap_size` muß vor `dsm_init` aufgerufen werden. Der Parameter `size` muß `>=0` sein. Bei `size=0` unterbleiben einige Initialisierungen für den DSM-Heap und er kann nicht benutzt werden.

`void dsm_heap_incr_size(size_t size)`
Wie `dsm_heap_size`, setzt jedoch die Größe der inkrementell anzulegenden DSM-Heapblöcke. `size` muß `>0` sein.

`void *dsm_malloc(size_t size, void *ptr_addr)`
Liefert einen Zeiger auf einen DSM-Speicherabschnitt der Größe `size` bzw. `NULL`, falls der Anforderung nicht nachgekommen werden kann. In `ptr_addr` muß die Adresse des Zeigers übergeben werden, dem der Rückkehrwert zugewiesen wird.

`void dsm_free(void *ptr, void *ptr_addr)`
Gibt den DSM-Speicherabschnitt, auf den `ptr` zeigt, wieder frei. In `ptr_addr` muß die Adresse von `ptr` übergeben werden.

`void *dsm_realloc(void *ptr, size_t size, void *ptr_addr)`
Ändert die Größe des DSM-Speicherabschnitts, auf den `ptr` zeigt, in `size`. Zu `ptr_addr` und Rückgabe von `NULL`: siehe `dsm_malloc`. Der Rückkehrwert muß wieder `ptr` zugewiesen werden.

`void *dsm_calloc(size_t count, size_t size, void *ptr_addr)`
Liefert einen Zeiger auf einen DSM-Speicherabschnitt mit `count` Elementen, jeweils der Größe `size`. Zu `ptr_addr` und `NULL`-Rückgabe: siehe `dsm_malloc`.

`void *dsm_memalign(size_t alignment, size_t size, void *ptr_addr)`
Liefert einen Zeiger auf einen DSM-Speicherabschnitt der Größe `size`, dessen Anfangsadresse ein Vielfaches von `alignment` ist. Zu `ptr_addr` und Rückgabe von `NULL`: siehe `dsm_malloc`.

`void *dsm_valloc(size_t size, void *ptr_addr)`
Liefert einen Zeiger auf einen DSM-Speicherabschnitt der Größe `size`, dessen Anfangsadresse ein Vielfaches von `DSM_PAGESIZE` ist. Zu `ptr_addr` und Rückgabe von `NULL`: siehe `dsm_malloc`.

Literaturverzeichnis

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

[2] Frank Mueller: On the Design and Implementation of DSM-Threads. In *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] Frank Mueller: A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29-41, Jan. 1993.

[5] Andrew S. Tannenbaum: *Verteilte Betriebssysteme*, Prentice Hall, 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. In *ACM Trans. on Computer Systems*, vol. 7, pages 321-359, Nov. 1989.

[8] Brian W. Kernighan, Dennis M. Ritchie: *Programmieren in C*, Carl Hanser Verlag und Prentice Hall, 1990.

[9] David Cronk, Matthew Haines, Piyush Mehrotra: Thread Migration in the Presence of Pointers. In *Technical Report, Institute for Computer Applications in Science and Engineering*, Number TR-96-73, Dec. 1996.

[10] Sun Microsystems, Inc., Mountain View, California. *External Data Representation Protocol Specification*, Feb. 1986, Part Number 800-1324-03.