Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Seminar “Effiziente Programmierung”

Speicherverwaltung und -optimierung

vorgelegt von

Mirko Hartung

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 6947844

Betreuer: Kira Duwe

Hamburg, 2019-02-25

Abstract

In diesem Dokument werden Technologien und Praktiken zur Speicheroptimierung beim Programmieren diskutiert. Hierzu wird abgewogen Anwendungen auf geringe Speichernutzung oder schnelleren Speicherzugriff zu tunen. Als Grundlage wird zudem ein Verständnis für Speicher und die Verwaltung dessen durch das Betriebssystem vermittelt.

Inhaltsverzeichnis

1 Motivation

Programmieren ist zeitintensiv. Jeder, der schon einmal ein mittel großes Projekt geschrieben hat, kennt diese Problematik. Mit zunehmender Komplexität eines Programmes, muss ein immer größerer Teil der Zeit darin aufgewandt werden, um durch diese verursachte Fehler zu finden. Gerade die häufig durch Speicherverwaltung verursachten Fehler könnten durch den Wechsel zu einer abstrakteren Sprache verhindert werden. Doch lohnt sich für den Programmierer die gesparte Zeit, wenn die Abstraktion einer Hochsprache auf Kosten der Performance und des Speicherfußabdrucks gehen?

Beispielhaft sei hier Java zu erwähnen. Die automatische Speicherverwaltung verhindert eine ganze Kategorie von Fehlern. Allerdings führt genau dies auch zu Einbußen der Performance. Zur automatischen Verwaltung wird im Falle von Java ein sogenannter Garbage-Collector verwendet, welcher periodisch allen Speicherreferenzen folgt und daraus schließt, welche Datenstrukturen vom Programm aus erreichbar sind und den Speicher der Unerreichbaren freigibt. Für den Anwender ist es deutlich einfacher zum Löschen einer Datenstruktur die Referenz auf die selbige einfach zu verwerfen. Hier lässt sich das zentrale Problem erkennen. Allen Referenzen zu folgen ist eine sehr teure Operation und sollte daher möglichst selten ausgeführt werden. Dies hat allerdings zur Folge, dass Datenstrukturen mit einer kurzen Lebenszeit lange im Speicher gehalten werden. Dies ist nur eins der vielen Argumente dafür, Speicher manuell verwalten zu wollen.

In dem Fall, in welchen ein Projekt in einer hardwarenahen Programmiersprache geschrieben werden soll, müssen Programmierer ein Grundverständnis haben, wie der Speicher eines Computers überhaupt verwaltet wird. Im Abschnitt 2.1 wird auf die Betriebssystemseite eingegangen, da diese für den Programmierer gewöhnlich nicht einsehbar ist, aber dennoch Relevanz für die Leistung eines Programmes hat. Wenn in C programmiert wird, lohnt es sich die Praktiken aus Abschnitt 2.2 und dem Kapitel 3 um die Vorteile der Sprache optimal auszunutzen.

2 Speicherverwaltung

In diesem Kapitel wird vermittelt, wie der Speicher eines Computers verwaltet wird. Besonders im Fokus steht dabei der Unterschied der Verwaltung durch das Betriebssystem und der Perspektive einer von einem Nutzer programmierten Anwendung.

2.1 Verwaltung durch das Betriebssystems

2.1.1 Problembeschreibung

Zu Beginn steht die Frage im Raum, weshalb überhaupt fast jeder Computer heutzutage ein Betriebssystem besitzt. Zunächst wird ein veraltetes Computermodell vorgestellt, in welchen keine Betriebssysteme vorhanden sind. Im Kontrast mit modernen Systemen lässt sich so der Aufgabenbereich Betriebssysteme im Schwerpunkt Speicherverwaltung erkennen.

In frühen Rechnern ließ sich der Speicher als eine Reihe von linear Adressierbaren Speicherzellen betrachten. Unter dieser Betrachtungsweise des Speichers ist das Schreiben eines Programmes trivial. Da der Entwickler die Größe des physisch vorhandenen Speichers und der somit der letzten adressierbaren Stelle kennt, kann ohne Probleme jeder Variable eine feste Speicheradresse zugewiesen werden. In der Zeit der Stapelverarbeitung, zu welcher jeder Computer maximal ein Programm gleichzeitig ausführte, war dies ein valider Ansatz. Doch mit Aufkommen des Multiprogramming, also den nebenläufigen Ausführen vieler Programme auf einem Computer und einem geteilten physischen Speicher, gerät dieses Paradigma an Bedeutung. Zum Beispiel liegt im Multiprogramming Programm A ab der Adresse 0 und Programm B in der oberen Hälfte des Speichers. So muss ein Programm nun jedes Mal umgeschrieben oder neu kompiliert werden, wenn es an einer andere Speicherstelle liegt. Zudem wird nun klar, dass die Menge an Speicher, die ein Programm adressieren kann bereits vor der Ausführung feststeht.

2.1.2 Paging

Eine Variante wie ein Betriebssystem Speicher verwalten kann ist diesen in viele, kleine und gleichgroße zusammenhängende Intervalle einzuteilen. Diese fortan Seiten (englisch Page) genannten Speicherabschnitte können nun einzeln vom Betriebssystem einzelnen Prozessen zugeordnet werden. Prozess ist hierbei der Name des Betriebssystems für eine zur Ausführung geladene Programminstanz. Für einen jeden Prozess verwaltet das Betriebssystem eine sogenannte Seitentabelle, in welcher dokumentiert wird, welche Seite, welchen Prozess zugeordnet ist. Möchte ein Prozess nun schrumpfen oder weiteren

Speicher anfordern, kann dies das System darum bitten, die Seitentabelle entsprechend anzupassen. Das Betriebssystem kann bei Bedarf Seiten eines Prozesses auf die Festplatte auslagern und so freigewordene Seitenrahmen im Speicher einen anderen Prozess zuordnen. Informationen darüber, welche Seiten gerade in den Speicher geladen sind, in welchen diese Seitenrahmen diese liegen, werden auch in der Seitentabelle vermerkt.

Falls alle Seitenrahmen gefüllt sind und ein Prozess dennoch eine weitere Seite anfordert, muss das Betriebssystem die Entscheidung treffen, welche Seite auf die Festplatte ausgelagert wird, um einen Seitenrahmen freizumachen. Diesen Vorgang nennt man Swapping. Verdrägungsalgorithmen genauer zu behandeln würde den Rahmen dieses Dokumentes sprengen. Allgemein kann aber gesagt werden, dass möglichst Seiten ausgelagert werden sollten, wenn diese wahrscheinlich eh eine ganze Weile nicht mehr benutzt werden würden. Durch die hohen Kosten der Verdrängungsoperation wird es jedoch relevant um zu verstehen, woher manche Performanceengpässe kommen könnten.

2.1.3 Virtuelle Adressräume

Allein Paging löst allerdings nicht die in 2.1.1 beschriebene Problematik. Zudem stellt sich die Frage, wie die statisch im Programmcode vorhandenen Speicheradressen in reale Adressen umgewandelt werden. Seien alle Adressen die innerhalb eines Programmes referenziert werden virtuell. Das bedeutet, dass bevor der Speicherzugriff ausgeführt wird, eine Adressübersetzung ausgeführt werden muss. Um dies umzusetzen werden die Seitentabellen so erweitert, dass zu jeder Seite die Information vorhanden ist, unter welchen Teil des virtuellen Adressraums die Seite eingebunden ist. Moderne Rechnerarchitekturen verfügen über eine sogenannte Memory Managment Unit (MMU). Wenn das Betriebssystem einen Zeiger auf eine Seitentabelle in ein spezielles Register lädt, übersetzt die MMU automatisch Speicherzugriffe mit virtueller Adresse. Programme selber merken von diesem Vorgang nichts, die MMU liefert direkt die Speicherinhalte des Arbeitsspeichers zurück.

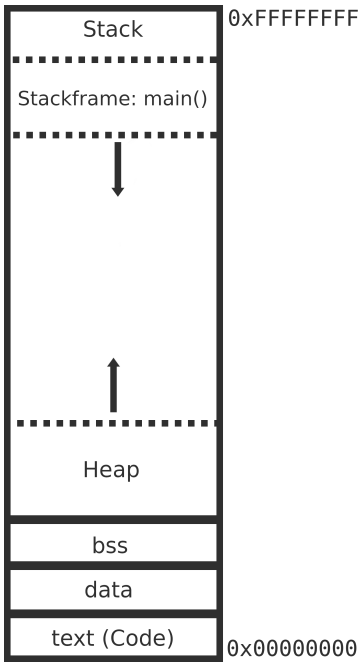

Abbildung 2.1: Speicherlayout eines Prozesses

```
long counter; // bss
const short number = 20; // data

int main(void)
```

```
{
  int i; // stack

  int *p; // stack

  p = malloc(sizeof (int)); // heap

  return 0;
}
```

Listing 2.1: Position verschiedener Variablen im virtuellen Adressraum

Jeder Prozess besitzt so seinen eigenen virtuellen Adressraum, denkt also auch dass der eigene Speicher bei der Adresse 0 beginnt. In der Abbildung 2.1 ist ein typischer Aufbau eines Adressraum eines Prozesses dargestellt. Der Programmcode beginnt an der niedrigsten Adresse, darauf folgend ist Platz für statische Daten und globale Variablen (Data und BSS). Zur Laufzeit neu angeforderte Seiten werden gewöhnlich in dem nach oben wachsenden Heap eingebunden. Lokale Variablen werden hingegen auf dem nach unten wachsenden Stack hinterlegt.

Ausführlichere Informationen lassen sich in der Quelle [Tan07] finden.

2.2 Programmieren mit dynamischer Speicherallokation

In Programmiersprachen wie C, die keine automatische Speicherverwaltung implementieren, muss eine Anwendung explizit neuen Speicher anfordern und später wieder freigeben. Damit beim Anfordern und freigeben nicht die Größe der Seiten geachtet werden muss, wird die eigentliche Speicherreservierung durch Bibliotheksfunktionen abstrahiert. Die Funktion *malloc()* nimmt als einzigen Parameter die Größe des zu reservierenden Speichers und liefert einen Zeiger auf die Anfangsadresse diesen zurück. Wenn ein Prozess Speicher wieder abgeben will, tut er dies mit der Funktion *free()*. Als Parameter wird der zuvor von *malloc()* produzierte Zeiger verwendet [Rie14].

2.2.1 Speicherlecks

Die feine Kontrolle über den Speicher in C bringt nicht nur potenzielle Performancegewinne, sondern stellt gerade neuen Programmierern häufig Fallen. Wenn dynamisch allokierte Datenstrukturen hinreichend komplex sind, ist die Aufgabe Speicher dieser freizugeben nicht trivial. Listing 2.2 zeigt eine fehlerfrei Implementation einer verketteten Liste. Es fällt auf, dass *deleteList()* rekursiv arbeitet. Wenn stattdessen versucht werden würde, dass erste Kettenglied zuerst freizugeben, würde das Programm nicht funktionieren. Der Grund ist, dass die Speicherfreigabe technisch nur möglich ist, wenn dessen Anfangsadresse bekannt ist. In beschriebener Situation würde aber der Zeiger auf den Nachfolger durch das Löschen des ersteren verloren gehen. Als Konsequenz würde der Prozess die Restliste nicht freigeben und im Laufe der Zeit immer mehr Speicher

allokieren. Dieses endlose Wachsen eines Programms nennt man Speicherleck. Im besten Fall lässt die Performance durch exzessives Swapping nach, im schlimmsten Fall stützt das Programm ab oder wird vom Betriebssystem beendet.

```
struct listElement {
  int value;
  struct listElement *next;
};

void deleteList(listElement* list) {
  if (list->next != NULL) {

    deleteList(list->next);

  }

  free(list);
}
```

Listing 2.2: Korrekte Speicherverwaltung einer verschachtelten Datenstruktur

3 Speicheroptimierung

Dieses Kapitel diskutiert die Optimierung von Programmen in Bezug auf Speicherfußabdruck und der Speicherzugriffsgeschwindigkeit. Die vorgestellten Techniken werden nach Umsetzbarkeit im Programmieralltag eingeordnet.

3.1 Optimieren von Structs in C

3.1.1 Structurepadding

Bei der Verwendung von Structs in C kann es dazu kommen, dass unnötig Speicher ungenutzt bleibt. Entgegen der allgemeinen Annahme werden die Membervariablen eines Structs nicht immer direkt im Speicher aufeinanderfolgend abgelegt. Das Listing 3.1 demonstriert dieses Verhalten an zwei Structs. Obwohl *listElementB* eine weitere Membervariable besitzt, nehmen beide Structs auf einem Computer mit 64-Bit Adressen eine genau gleich große Speicherstelle ein. Grund hierfür ist, dass in C versucht wird eine möglichst günstige Speicherausrichung zu erwirken.

```
struct listElementA {
  int value;
  struct listElement *next;
};

struct listElementB {
  int value;
  char pad[4];
  struct listElement *next;
};
```

Listing 3.1: Structs mit impliziten und expliziten Packing

Manche Prozessoren realisieren Speicherzugriffe effizienter, wenn die angeforderte Speichereinheit an einer Adresse liegt, welche ein Vielfaches der Größe des angeforderten Speichers ist. Daher versucht C in diesem Beispiel durch Padding, also dem Einfügen von ungenutzten Bytes, dafür zu sorgen, dass alle Membervaribalen optimal im Speicher liegen. Ein 64 Bit Zeiger hat in C eine Größe von 8 Byte. Integer hingegen nur 4 Byte. Um sicherzustellen, dass *next* an einer durch 8 teilbaren Adresse gespeichert wird, wird auch der Integer so ausgerichtet, dass er an einer Adresse beginnt die dieses Kriterium erfüllt. Als Konsequenz entstehen so 4 Byte Padding.

```
struct containerD
{
  char c;
  // char pad1[3];
  int i;
  short s;
  // char pad2[6];
  long l;
};
```

Listing (3.2) Mit Padding **24 Bytes**

```
#pragma pack(1)
struct containerD
{
  char c;
  int i;
  short s;
  long l;
};
```

Listing (3.3) Ohne Padding **15 Bytes**

3.1.2 Deaktivieren von Padding

Da Structurepadding lediglich eine versteckte Optimierung seitens des Compilers ist, ist es relativ einfach dieses Verhalten zu unterbinden. Das auf Speichergröße abzielende Tuning von Structs bezeichnet man auch als Structurepacking [Ray19]. Mithilfe eines Pragmas kann zum Beispiel der Compiler *GCC* angewiesen werden, kein Padding in Structs einzufügen. Listing 3.3 zeigt einen Beispielcode. Es ist leicht zu erkennen, dass ohne das Pragma viel Speicherplatz verschenkt wird.

Es gibt neben den Compileranweisungen auch die Möglichkeit, durch manuelles Eingreifen, dass Padding zu verhindern. Im Beispiel ist erkennbar, dass nach dem Member *c* nur 3 Byte Padding folgen. Der Integer muss lediglich eine durch 4 teilbare Adresse besitzen. Mit Umsortieren der Variablen ist daher eine effizientere Speicherung möglich, ohne, den Compiler zu zwingen ineffiziente Speicherzugriffe zu verwenden. Die Strategie sollte hier sein, die kleinsten Datentypen oben im Struct stehen zu haben und danach die nächst größeren Member.

3.1.3 Padding in Arrays

Warum müssen überhaupt auch kleine Datentypen auf für diese eigentlich unnötig strengen Adressen liegen? Die Überlegung ist, dass auch Arrays von Structs das Kriterium der optimalen Speicherposition erfüllen sollen. Durch die Abbildungen 3.2 und 3.2 ist leicht erkennbar, dass Packen bei den Nachfolgern zu nicht am Speicher ausgerichteten Zugriffen führt. Es bleiben aber immer noch einige korrekt ausgerichtete Structs über. In diesem Fall ist bei jeden vierten Struct eine günstige Adressausrichtung gegeben.

```
struct container
{
  char c;
  int i;
};
```

Abbildung 3.2

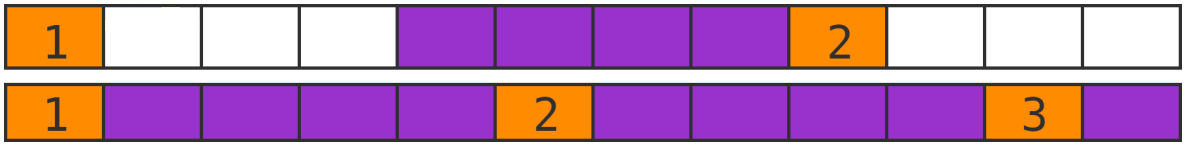

Abbildung 3.3: Oben normal, unten mit Pragma

3.1.4 Vergleich der Effizienz

Es stellt sich die Frage, ob die Speicherzugriffe wirklich effizienter sind. Um dies zu analysieren sollte zunächst der aus dem C-Code generierten Assembler betrachtet werden (Listing 3.4 und 3.5) . Auffällig ist, dass durch das packen keine zusätzlichen Operationen generiert werden. Dennoch zeigen synthetische Benchmarks [San08], dass der Zugriff wirklich langsamer ist.

```
push    rbp
mov     rbp, rsp
mov     BYTE PTR [rbp-4], 97
mov     DWORD PTR [rbp-8], 20
mov     eax, 0
pop     rbp
ret
```

Listing 3.4: Mit Padding

```
push    rbp
mov     rbp, rsp
mov     BYTE PTR [rbp-1], 97
mov     DWORD PTR [rbp-5], 20
mov     eax, 0
pop     rbp
ret
```

Listing 3.5: Ohne Padding

3.1.5 Fazit

Wie besonders in den Listings 3.2 und 3.3 sichtbar ist, kann bei unachtsamer Nutzung schnell viel Speicher verschenkt werden. Dennoch gibt es Fälle, in welchen die hier aufgezeigten Techniken nicht passen. In Fällen in welchen genügend Speicher vorhanden ist oder wenige Exemplare eines Structs erstellt werden, lohnt sich der Aufwand zum Umsortieren der Membervariblen eventuell nicht. Zudem nimmt hier die Lesbarkeit des Codes ab.

Interessant ist auch zu erkennen, das das Pragma *pack(1)* nicht zwingend die Performance verringern muss. Wenn die Datenabhängigkeiten in einem Programm relativ lokal sind, passen die Daten durch das Schrumpfen eventuell in den Cache. Das hätte einen gegenteiligen Effekt und wird in dem Benchmark von [San08] explizit umgangen.

3.2 Datenkompression

Ein weiterer Ansatz zur Optimierung von Programmen ist Daten zu komprimieren. Dabei können zwei Ziele verfolgt werden. Zum einen die Beschleunigung der Ausführung einer bestimmten Aufgabe und zum anderen die Reduktion der benötigten Speicherkapazitäten. Allgemein ist die Praxistauglichkeit Datenkompression schwer zu beurteilen. Die erste Voraussetzung ist, dass die Daten überhaupt genügen Redundanz besitzten um komprimiert zu werden. Zudem bringt der Einsatz auf dynamischen Daten durch häufiges Komprimieren kaum Mehrwert. Wenn allerdings ein Problem die Dimensionen vorhandenen Speichermedien übersteigt macht dieses Werkzeug dennoch Sinn.

3.2.1 Kompression des sekundären Speichers

Zugriffe auf sekundäre Speichermedien sind im Vergleich mit dem Arbeitsspeicher sehr teuer. Besonders Festplatten weisen eine geringe Datenrate und eine hohe Latenz auf. Daher kann es sich lohnen Daten auf diesen Medien komprimiert zu lagern und erst bei Bedarf im Arbeitsspeicher zu entpacken. Ein Beispiel für solch ein Vorgehen ist der Linux Kernel. Hier wird zur Beschleunigung des Bootvorgangs ein komprimierter Kernel geladen, der sich im selbst im Speicher dekomprimiert [Meu08]. Abhängig vom Kompressionsalgorithmus kann so viel Zeit eingespart werden [Lee].

3.2.2 Kompression im Hauptspeicher

In 2.1.2 wurde erwähnt, dass bei zu hoher Auslastung Daten aus dem Hauptspeicher ausgelagert werden müssen. Kompression kann bei korrekter Anwendung verhindern, das auf teures Swapping zurückgegriffen werden muss. [Riv18] demonstriert anhand von Datenbanken, wie eine Balance zwischen Kosten für die Dekompression und der nutzbaren Daten gefunden werden kann.

3.3 Unions in C

Für Anwendungen, wo Speicher besonders knapp ist, lohnt es sich die Verwendung von Unions zu erwägen. Dieses Sprachkonstrukt erlaubt es ein einzige Speicherstelle als verschiedene Datentypen zu interpretieren. 3.6 zeigt, wie Unions genutzt werden können. Der Anwedungsfall kann sein, wenn erst zur Laufzeit festgestellt werden kann, welcher Typ von Daten gespeichert werden muss. Alternativ hierzu kann die Wiederverwertung einer Speicherstelle, um eventuell einen Schleifenrumpf klein genug für den Cache zu machen.

In allen Anwendungen benötigen Unions viel Planung seitens der Entwickler und sind eventuell sind für jede Problemstellung anwendbar. Außerdem wird das Programm bei fehlender Sorgfalt schnell fehlerhaft. Diese Aspekte machen Unions zu einer nicht praxisrelevanten Optimierungstechnick.

```
union floatOrInt
{
  float float;
  int int;
};

int main (void)
{
  union floatOrInt number;

  number.float = 4.0f;

  number.int = 17;
}
```

Listing 3.6: Beispiel einer Union in C

4 Zusammenfassung

Die Auseinandersetzung mit dem Thema Speicherverwaltung durch Nutzer ist wichtig, da zum Beispiel ein großer Teil der in diesem Dokument vorgestellten Optimierungsverfahren nur mithilfe manueller Speicherverwaltung möglich ist. Das Mikromanagement des Speichers kann aber auch Ursache vieler Fehler sein oder für manche Aufgaben zu aufwendig. Optimierung ist immer zeitintensiv, während die potenziellen Verbesserungen nur schwer prognostiziert werden können. Die Empfehlung ist daher mit bewussten Tuning erst zu beginnen, wenn die Anwendung vollständig ist und korrekte Ergebnisse liefert. Mit Werkzeugen wie *gprot* lassen sich zudem Teile von Programmen identifizieren, welche den größten Anteil der Laufzeit haben.

Literaturverzeichnis

[Lee] Kyungsik Lee. Lz4 compression and improving boot time. https://events.static.linuxfound.org/sites/events/files/lcjpcojp13_klee.pdf.

[Meu08] Frans Meulenbroeks. About compression. https://elinux.org/About_Compression, 2008.

[QZ12] Ying Qiao Qi Zhu. A survey on computer system memory management and optimization techniques. http://article.sapub.org/10.5923.j.ajca.20120103.01.html, 2012.

[Ray19] Eric S. Raymond. The lost art of structure packing. http://www.catb.org/esr/structure-packing, 2019.

[Rie14] Jakob Rieck. Pointers and dynamic memory management in c. https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2014/cgk-14-rieck-zeig-e-und-dynamische-_speicherverwaltun-report.pdf, 2014.

[Riv18] Andy Rivenes. Database in-memory compression. https://blogs.oracle.com/in-memory/database-in-memory-compression, 2018.

[San08] Alexander Sandler. Aligned vs. unaligned memory access. http://www.alexonlinux.com/aligned-vs-unaligned-memory-access, 2008.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[Wis17] Tim Wischhof. Memory management and optimizations. https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2017_2018/ep-1718-wischhof-memory-management-praesentation.pdf, 2017.