Universität Hamburg
Fachbereich Informatik
Scientific Computing / Wissenschaftliches Rechnen

Seminar
Effiziente Programmierung WS2017/18

Seminararbeit

Speicherverwaltung und Optimierung

Tim Wischhof

Betreuerin: Kira Duwe

2018-03-01

Inhaltsverzeichnis

1 Einleitung ... 2

2 Speicherorganisation ... 2
2.1 Speicherzugriff ... 4
2.2 Virtueller Speicher ... 4

3 Statische Speicherverwaltung ... 5
3.1 Der Stack ... 5
3.2 Limitierungen des Stacks ... 6

4 Dynamische Speicherverwaltung ... 7
4.1 Dynamische Speicherverwaltung in C ... 7
4.2 Risiken dynamischer Speicherverwaltung ... 9

5 Optimierung der Speicherverwaltung ... 10
5.1 Data Alignment ... 10
5.2 Compileroptimierung ... 12
5.3 Weitere Optimierungsmöglichkeiten ... 13

6 Zusammenfassung ... 14

Bibliographie ... 15

1 Einleitung

Ein Computer besteht auf grundlegender Ebene aus zwei interagierenden Komponenten: dem Speicher und dem Prozessor, der oft auch als CPU (central processing unit) bezeichnet wird. Dabei ist der Speicher für die Aufbewahrung von Programmen und Daten in Binärform verantwortlich, während der Prozessor die Abarbeitung von Programmen im Speicher sowie sämtliche logischen und arithmetischen Rechenoperationen übernimmt. Aufgrund der Zentralität dieser Komponenten ist ihre optimale Nutzung von wesentlichem Interesse, um die Laufzeiten und benötigten Rechenleistungen von Computerprogrammen zu reduzieren. Diese Seminararbeit gibt einen Überblick über die Organisation sowie Nutzung des Speichers in modernen Computerarchitekturen und geht darauf ein, wie ein Programmierer auf dieses Verfahren Einfluss nehmen kann. Der Prozessor wird dabei lediglich hinsichtlich seiner Interaktion mit dem Speicher betrachtet.

Zu diesem Zweck werden zunächst Aufbau und Organisation moderner Speicherarchitekturen beschrieben, bevor auf die grundlegende Interaktion von CPU und Speicher eingegangen wird. Basierend auf diesen Grundlagen wird daraufhin verdeutlicht, wie der Speicher für die Ausführung von Computerprogrammen verwaltet wird und welche Möglichkeiten der Entwickler eines solchen Programms hat, darauf Einfluss zu nehmen. Letzteres wird anhand der Programmiersprache C verdeutlicht, da diese dem Programmierer die direkte Manipulation der Speicherverwaltung ermöglicht.

2 Speicherorganisation

In modernen Computerarchitekturen ist der Speicher nicht ein einziges Modul, sondern besteht aus unterschiedlichen, miteinander interagierenden Komponenten. Der Grund dafür liegt in den verschiedenen Anforderungen an den Speicher, die bisher nicht alle von einer einzigen Komponente realisiert werden konnten. Der ideale Speicher würde aus nur einem einzigen Datenträger bestehen, der eine große Menge an Daten permanent speichern kann und gleichzeitig einen schnellen Zugriff auf diese Daten bietet. Um die drei wichtigsten Anforderungen, die große Datenmenge, die Persistenz der Daten und den schnellen Zugriff, zu realisieren, wurde eine Speicherarchitektur entwickelt, die zwischen Primär- und Sekundärspeicher unterscheidet. Diese wird in Abbildung 1 dargestellt.

Der Primärspeicher ist für alle Daten zuständig, die entweder vor kurzem vom Prozessor bearbeitet worden sind oder kurz davor stehen, von diesem bearbeitet zu werden. Er besteht in der Regel aus dem Cache und dem Arbeitsspeicher, der auch als Hauptspeicher oder RAM (random access memory) bezeichnet wird. Der Arbeitsspeicher dient dabei als Ablage für alle Daten, die direkt vom Prozessor bearbeitet werden können. Seine Größe bestimmt somit die Menge an Daten, mit denen der Prozessor arbeiten kann. Er zeichnet sich dadurch aus, dass Lese- und Schreibzugriffe des Prozessors schnell und zudem wahlfrei sind.

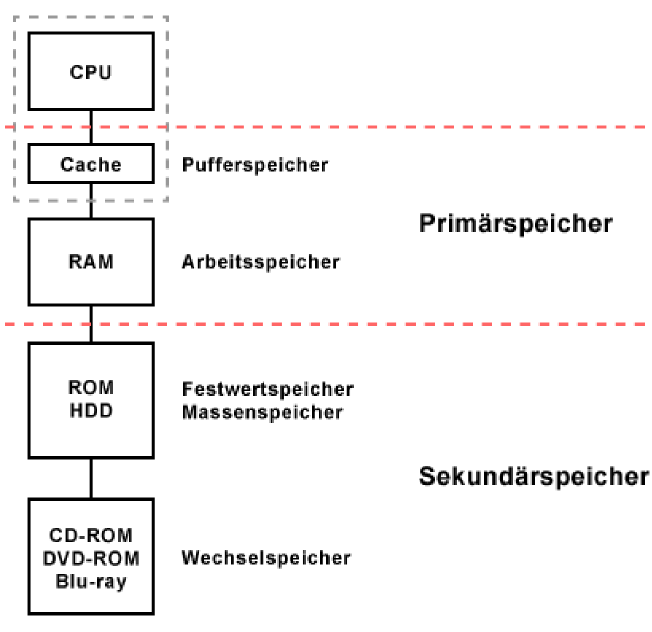

Abbildung 1: Speicherarchitektur
https://www.elektronik-kompendium.de/sites/com/1812051.htm

Letzteres bedeutet, dass der Zugriff auf all seine Daten gleich lange dauert. Der größte Nachteil des RAM liegt darin, dass darin gespeicherte Daten nicht beständig sind, also der Inhalt beim Ausschalten des Computers verloren geht. Diesen Nachteil hat auch die andere Komponente des primären Speichers, der Cache. Dieser ist direkt in den Prozessor integriert und bietet eine noch geringere Zugriffszeit als der Arbeitsspeicher im Austausch für eine vergleichsweise geringe Größe. Er enthält einen Teil der Daten im RAM und dient als Zwischenspeicher, um die Zugriffszeit des Prozessors auf benötigte Daten weiter zu reduzieren. Daher soll der Cache zu jedem Zeitpunkt Daten enthalten, die der Prozessor in naher Zukunft benötigen wird. Greift der Prozessor auf Daten zu, von denen sich keine Kopien im Cache befinden, wird dies als Cache Miss bezeichnet. Um die Anzahl der Cache Misses zu verringern, nutzt man das Prinzip der zeitlichen Lokalität. Dieses besagt, dass Daten, die vor kurzem vom Prozessor angesprochen wurden, mit hoher Wahrscheinlichkeit bald erneut benötigt werden. Im ersten Schritt werden Kopien aller angesprochenen Daten im Cache gespeichert. Dabei werden nicht nur die angesprochenen Informationen, sondern auch ein Teil der umliegenden Daten gespeichert. Die Größe der so gespeicherten Daten werden als Cache Lines bezeichnet. Durch seine geringe Größe wird dieser allerdings recht schnell voll und so müssen Cache Lines wieder entfernt werden. Der Cache enthält also nicht alle Daten, die der Prozessor benötigen könnte, sondern versucht, vorherzusagen, für welche Daten dies zutrifft und diese zu speichern. So kann die insgesamt für Speicherzugriffe benötigte Zeit deutlich reduziert werden.

Der Sekundärspeicher wird zur Speicherung großer Datenmengen benutzt, die zurzeit nicht vom Prozessor verarbeitet werden. Er kann aus verschiedenen Komponenten bestehen, wobei Festplatten und Wechselspeichermedien wie CDs, DVDs und Blu-Rays zu den prominentesten zählen.

Komponenten des Sekundärspeichers zeichnen sich dadurch aus, dass sie große Datenmengen auch ohne stetige Energieversorgung speichern können. Allerdings ist die Zugriffszeit im Vergleich zum Primärspeicher deutlich langsamer. Aus diesem Grund interagiert der Prozessor nicht direkt mit Daten des Sekundärspeichers. Stattdessen werden vom Prozessor benötigte Daten vor ihrer Benutzung in den RAM geladen. Der Prozessor arbeitet dann mit den Daten im Arbeitsspeicher und nach der vollständigen Abarbeitung aller Anweisungen werden die Daten zur Persistierung wieder auf der Festplatte gespeichert. Durch das Zusammenspiel von Primär- und Sekundärspeicher lassen sich auf diese Weise sowohl schnelle Zugriffszeiten als auch die permanente Speicherung großer Datenmengen realisieren.

2.1 Speicherzugriff

Prozessoren greifen typischerweise nicht einzeln auf die Bits im Arbeitsspeicher zu, sondern adressieren ihn stattdessen in Segmenten bestimmter Größe. Die Größe dieser Segmente ist prozessorabhängig und beträgt in modernen Maschinen typischerweise 4 Byte (32 Bit) oder 8 Byte (64 Bit). Jedes dieser Segmente hat eine Adresse, über die der Zugriff auf die dort gespeicherten Daten ermöglicht wird. Durch das Zusammenfassen mehrerer Bytes zu einer Adresse lässt sich der nutzbare Speicherplatz trotz gleichbleibenden Adressenraums drastisch erhöhen. Möchte ein Prozess nun Daten in den Arbeitsspeicher schreiben, wird zunächst ein nicht anderweitig reservierter, kontinuierlicher Adressblock entsprechender Größe reserviert (auch: allokiert). Erst danach können die Daten an diese Stelle im Speicher geschrieben werden. Dieser Speicherbereich kann ausschließlich von dem Prozess genutzt werden, der ihn allokiert hat und muss von diesem erst wieder freigegeben werden, bevor er anderweitig nutzbar ist. Dies stellt sicher, dass kein Prozess die Daten anderer Prozesse lesen oder überschreiben kann.

2.2 Virtueller Speicher

Der Arbeitsspeicher stellt in modernen Computern oft nur einen Bruchteil des insgesamt zur Verfügung stehenden Speichers dar. Da Prozessoren nicht direkt auf den Sekundärspeicher zugreifen können, können sie daher mit einem großen Teil der Daten gar nicht interagieren. Um dieses Problem zu bewältigen gibt es die sogenannte virtuelle Speicherverwaltung. Dieser Begriff ist ein wenig irreführend, weil nicht die Verwaltung, sondern vielmehr der Speicher selbst virtuell ist. Jedem Prozess wird dabei ein scheinbar zusammenhängender Speicherbereich zur Verfügung gestellt, der als virtueller Speicher des Prozesses bezeichnet wird und unabhängig von der tatsächlichen Größe des Arbeitsspeichers ist. Dieser Speicherbereich bildet den virtuellen Adressraum des Prozesses und ist in sogenannte Seiten (englisch: Pages) uniformer Größe aufgeteilt. Dasselbe gilt für den physischen Adressraum und bei jedem Zugriff eines Prozesses auf eine virtuelle Seite wird diese auf eine physische Seite abgebildet.

Da der virtuelle Adressraum oft mehr Adressen beinhaltet als im Arbeitsspeicher umsetzbar sind, werden einige physische Pages vorübergehend an sekundäre Speichermedien ausgelagert. Greift der Prozessor nun auf eine Page zu, deren physische Abbildung sich nicht im Arbeitsspeicher befindet, kommt es zu einem Page Fault. Das Betriebssystem wählt dann eine Seite aus dem Arbeitsspeicher aus und tauscht diese mit der benötigten Seite aus. Erst danach kann der Prozess normal fortgesetzt werden. Dieses Verfahren kostet Zeit und kann die Laufzeit eines Prozesses maßgeblich erhöhen. Aus diesem Grund ist es von Interesse, die Anzahl der Page Faults möglichst gering zu halten. Dieses Problem deckt sich mit der Reduzierung von Cache Misses und nutzt dieselben Lösungsstrategien. Mithilfe der virtuellen Speicherverwaltung lässt sich so die Anzahl der für den Prozessor zugänglichen Daten deutlich erhöhen.

3 Statische Speicherverwaltung

Sobald ein Programm vom Sekundärspeicher in den RAM geladen wird, allokiert das Betriebssystem automatisch einen Speicherbereich gewisser Größe im Arbeitsspeicher, der während der Ausführung dieses Programms benutzt werden darf. Zu den Daten, die in diesen Speicherbereich geschrieben werden, zählen unter anderem der binäre Programmcode und sämtliche in dem Programm also solche deklarierten globalen Variablen, also Daten, auf die über den gesamten Programmablauf hinweg zugegriffen werden kann. Durch die separate Speicherung dieser Daten, die in vielen Fällen nur einen geringen Anteil des insgesamt für das Programm benötigten Speichers ausmachen, lassen sie sich unabhängig vom Programmablauf im Cache speichern. Da sowohl der Programmcode als auch globale Variablen relativ häufig angesprochen werden müssen, lässt sich auf diese Weise die Programmlaufzeit verringern.

3.1 Der Stack

Zusätzlich wird für das Programm ein Speicherbereich reserviert, der für die Speicherung von Daten genutzt wird, die während des Programmablaufs entstehen. Dieser Bereich wird vom Prozessor wie die Datenstruktur Stack behandelt und deshalb auch geläufig als solcher bezeichnet. Der Stack arbeitet nach dem LIFO-Prinzip: last-in first-out. Gemäß diesem werden die Daten, die zuletzt zum Stack hinzugefügt worden sind, zuerst wieder entfernt. Der Stack arbeitet mit lediglich zwei verschiedenen Operationen: Die push-Operation fügt Daten zum Stack hinzu, während die pop-Operation die zuletzt hinzugefügten Daten vom Stack entfernt. Dieses Verhalten eignet sich gut für die Verwaltung temporärer Programmdaten, da es der logischen Abarbeitungsreihenfolge von Programmen entspricht. Bei verschachtelten Aufrufen werden so die Daten des innersten Aufrufs zuerst wieder gelöscht.

Dies sorgt für eine effiziente Verwaltung des Programmspeichers. Die Speicherung von Daten auf dem Stack geschieht in Form von temporären Datenmengen, sogenannten Stack Frames. Das Erstellen eines Stack Frames entspricht der push-Operation des Stacks. Jeder dieser Frames repräsentiert dabei einen Funktionsaufruf und beinhaltet all seine Parameter, lokalen Variablen und die Adresse der aufrufenden Operation auf dem Stack. Sobald der entsprechende Funktionsaufruf endet, wird der für den entsprechenden Stack Frame allokierte Speicherbereich wieder freigegeben und kann für andere Variablen des Stacks genutzt werden. Dieser Vorgang entspricht der pop-Operation des Stacks. Während die maximale Größe des Stacks zu Beginn des Programms bereits feststeht, geschieht das Allokieren und Freigeben des Speichers für Stack Frames erst zur Laufzeit, da der Programmablauf zur Übersetzungszeit nicht zwangsläufig deterministisch ist. Dies hat zur Folge, dass die Größe des Stacks zur Laufzeit variiert, da dieser durch push- und pop-Operationen wächst bzw. schrumpft. Ein großer Vorteil der Verwendung von Stack Frames liegt in der Funktionsweise des Caches. Die Frames sind oft von relativ geringer Größe, sodass sie in ihrer Gänze in den Cache passen. Aus diesem Grund sind Zugriffe auf den Stack sehr schnell. Zusätzlich muss sich der Programmierer keine Gedanken um die Verwaltung des Speichers für sein Programm machen, da der Prozessor dies automatisch tut.

3.2 Limitierungen des Stacks

Neben der effizienten Verwaltung des Stacks bringt seine Verwendung auch einige Nachteile mit sich. So sind Stack Frames zwar eine effiziente Methode, um temporäre Daten zu speichern, bilden aber gleichzeitig unveränderliche Lebensdauern dieser Daten. Das Beenden einer Funktion hat immer eine pop-Operation des Stacks zur Folge und somit gehen alle Daten, die sich in diesem Stack Frame befanden, verloren. Möchte man aber, dass gewisse Daten auch nach Abschluss der berechnenden Funktion gespeichert bleiben, geht dies nur, indem man diese Daten als Ergebnis der Funktion zurückgibt. Dies hat zur Folge, dass die Daten in den Stack Frame der aufrufenden Funktion kopiert werden müssen – eine Operation, die vor allem für große Datenmengen aufwendig ist.

Ein weiterer Nachteil des Stacks liegt zudem in seiner beschränkten Größe. Werden die im Laufe des Programms berechneten Datenmengen zu groß, kann der Stack sie ab einem gewissen Punkt gar nicht mehr speichern. Dies liegt daran, dass die maximale Größe des Stacks bereits zur Übersetzungszeit festgelegt ist, der benötigte Speicherplatz für die Stack Frames aber erst zur Laufzeit offenbart wird. Als Resultat kann die Größe der Stack Frames das Limit des Stacks zur Laufzeit überschreiten. Dieses Phänomen wird als Stack Overflow bezeichnet und hat zur Folge, dass das entsprechende Programm beendet wird. Beispielsweise können so große, verschachtelte Berechnungen durch die Anzahl der Stack Frames zu einem Programmabsturz führen und Datenmengen, die zu groß sind, als dass der Stack sie erfassen kann, sind ohnehin nicht handhabbar.

Aus diesen Gründen ist der Stack für die Speicherung großer Datenmengen ungeeignet. Zu diesem Zweck wird daher eine Alternative für die Verwaltung temporärer Programmdaten genutzt – die dynamische Speicherverwaltung.

4 Dynamische Speicherverwaltung

Dynamische Speicherverwaltung erlaubt die freie Nutzung des Speichers für Daten, die zur Laufzeit eines Programmes entstehen. Sie stellt weniger eine Alternative zum Stack dar und wird viel mehr in Zusammenarbeit mit diesem benutzt, um große Datenmengen zu verwalten, deren Speicherung auf dem Stack sonst unmöglich wäre. Anders als der Stack, muss dynamische Speicherverwaltung im Programmcode explizit angesprochen werden. Daraufhin wird ein freier Speicherbereich der benötigten Größe ermittelt und für das Programm allokiert. Erst danach können Daten in diesem Bereich gespeichert werden. Dynamische Speicherverwaltung erlaubt es einem Programm, beliebig viel Speicher für das Aufbewahren seiner Daten zu verwenden und die Lebensdauer dieser frei zu steuern. Die Gesamtheit des auf diese Weise reservierten Speichers wird als Heap bezeichnet. Da die Menge des dynamisch allokierten Speichers zur Laufzeit variiert, ist die Größe des Heaps nicht konstant. Zudem ist der Heap, anders als der Stack, in seiner Größe nicht begrenzt. Der Heap kann theoretisch so lange wachsen, bis der gesamte verfügbare Speicherplatz aufgebraucht ist. Während dynamische Speicherverwaltung nicht durch die automatische Verwaltung des Stacks limitiert ist, bringt sie auch nicht dessen Vorteile mit sich. So ist es möglich, Daten beliebig lange zu speichern, ohne durch die Lebensdauer des entsprechenden Stack Frames eingeschränkt zu sein. Außerdem erlaubt er die Speicherung von deutlich mehr Daten als im Stack überhaupt möglich wäre. Allerdings muss die Verwaltung des Heaps durch den Programmierer manuell gesteuert werden und geschieht im Gegensatz zum Stack nicht automatisch. Zudem geht die schnelle Datenverwaltung mittels Stack Frames bei Benutzung des Heaps verloren. Zugriffe auf den Heap sind also im Vergleich zum Stack langsamer. Die Art seiner Verwendung ist abhängig von der Programmiersprache, in der das jeweilige Programm geschrieben wurde. Einige Sprachen übernehmen die Verwaltung des Heaps, ähnlich wie beim Stack, automatisch. Beispielsweise nutzt Java den Heap für die Speicherung sämtlicher Objekte und erlaubt dem Programmierer keinen expliziten Zugriff darauf. Die Programmiersprache C hingegen erlaubt dem Programmierer, die Verwaltung des Heaps manuell zu steuern. Aus diesem Grund wird C wird im Folgenden für die detailliertere Erklärung dynamischer Speicherverwaltung verwendet.

4.1 Dynamische Speicherverwaltung in C

Die Programmiersprache C stellt dem Programmierer Methoden zur Verfügung, um zur Laufzeit dynamisch Speicher auf dem Heap zu verwalten. Es gibt vier Funktionen, die zu diesem Zweck benutzt werden können:

- void* malloc(size_t size),
- void* calloc(size_t n, size_t size)
- void* realloc(void* ptr, size_t size)
- void free(void* ptr)

Die Funktionen malloc (kurz für memory allocation) und calloc werden benutzt, um während der Laufzeit Speicher auf dem Heap zu allokieren. Sie unterscheiden sich zum einen durch die leicht unterschiedliche Parametrisierung und zum anderen durch die Initialisierung des allokierten Speichers. Der einzige Parameter der Funktion malloc ist die Größe des zu reservierenden Blocks im Speicher. Calloc hingegen erwartet nicht die gesamte Größe des zu allokierenden Speichers, sondern die Anzahl der Werte sowie die Größe eines dort zu speichernden Datentyps. Die Größe des Speicherblocks wird dann anhand dieser Werte berechnet. Der Typ aller genannten Parameter, size_t, ist ein vorzeichenloser Integer-Typ. Er stellt sicher, dass keine negativen Werte für die Speichergrößen angegeben werden können. Um die korrekte Größe des zu allokierenden Speicherbereichs anzugeben, wird häufig die C-native Funktion sizeof benutzt. Übergibt man dieser Funktionen einen Datentypen, so gibt sie Auskunft über die Größe dieses Datentyps. Dies ist hilfreich, weil Datentypen im Speicher nicht immer dieselbe Größe haben, sondern von Compiler und Betriebssystem abhängen. Somit ist sizeof eine sichere und einfache Methode, um die Größe von Datentypen anzugeben. Beispielsweise würde die Allokation eines Speicherbereiches für zehn Integer-Werte mittels der Aufrufe malloc(10*sizeof(int)) oder calloc(10,sizeof(int)) durchgeführt werden.. Durch die Nutzung von sizeof sind diese Funktionsaufrufe also sehr ähnlich. Der wesentlichere Unterschied dieser beiden Funktionen ist, dass malloc lediglich einen Speicherbereich allokiert und nichts an den Werten ändert, die dort bereits stehen, während calloc sämtliche Bits des reservierten Speicherbereichs sofort mit Nullen initialisiert. Dies kann für einige Anwendungen praktisch sein, bringt aber einen zusätzlichen Zeitaufwand mit sich. Daher wird in der Regel malloc benutzt, wenn die Initialisierung des Speicherbereiches nicht von Interesse ist.

Der Rückgabetyp beider Funktionen ist ein Zeiger (englisch: Pointer) auf die erste Adresse des allokierten Speicherblocks. Mithilfe dieses Zeigers kann auf den allokierten Speicher zugegriffen werden, um dort Daten zu speichern oder zu lesen. Die Pointer macht sich aber auch die Funktion realloc zunutze. Im Gegensatz zu obigen Funktionen allokiert diese keinen neuen freuen Speicherbereich, sondern wird benutzt, um die Größe eines bereits reservierten Speicherbereichs zu verändern. Dafür wird ihr ein sowohl ein Pointer auf einen zuvor mittels malloc oder calloc allokierten Speicherbereich als auch die Größe des neuen Speicherbereichs übergeben. Falls sich der allokierte Speicherbereich dadurch vergrößert, kann es sein, dass der Pointer sich durch einen Aufruf von realloc verändert. Dies liegt daran, dass die Adressen hinter dem zuvor reservierten Bereich nicht zwangsläufig frei sind und die Daten deswegen an eine andere Stelle kopiert werden müssen, um den Speicherbereich zu erweitern.

Sobald ein allokierter Speicherbereich nicht weiter benötigt wird, muss dieser wieder freigegeben werden bevor er für andere Zwecke verwendet werden kann. Dies geschieht mithilfe der Funktion free. Übergibt man ihr einen Pointer auf einen zuvor mithilfe dynamischer Speicherverwaltung allokierten Speicherbereich, so gibt free diesen wieder frei.

4.2 Risiken dynamischer Speicherverwaltung

Mittels dynamischer Speicherverwaltung lassen sich die Limitierungen des Stacks, also die begrenzte Größe und die eingeschränkte Lebensdauer der Daten, umgehen. Letzteres ergibt sich durch die Funktionalität der Pointer. Anders als beim Stack, kann mithilfe von Pointern so lange auf Daten im Heap zugegriffen werden, bis der zugehörige Datenbereich wieder freigegeben wurde. Möchte man also, dass diverse Daten auch über ihre Funktion hinaus nutzbar sind, so ist es möglich, anstelle der Daten einfach den Pointer auf den entsprechenden Speicherbereich an andere Funktionen zu übergeben. Da ein Pointer in modernen Systemen oft der Größe eines einzigen Integer-Wertes entspricht, ist dies vor allem für große Datenmengen sehr effizient.

Neben diesen Vorteilen bringt die Freiheit der dynamischen Speicherverwaltung allerdings auch Risiken mit sich. Die Verwaltung des Heaps liegt in der Verantwortung des Programmierers und wird nicht automatisch reguliert. Daher ist es wichtig, einmal auf dem Heap allokierten Speicherplatz wieder freizugeben, sobald man diesen selbst nicht mehr benötigt. Andernfalls kann dieser Speicherbereich nicht anderweitig verwendet werden. Dieses Phänomen wird als Memory Leak bezeichnet und kann bei häufigem Auftreten eine gravierende Verlangsamung des Systems zu Folge haben. Dadurch, dass weniger Speicher im RAM zur Verfügung steht, kann es bei der neuen Allokation von Speicherbereichen vermehrt zu Page Faults kommen. Da Speicherbereiche, die von Memory Leak betroffen sind, zu einem gerade aktiven Programm gehören, werden diese mit hoher Wahrscheinlichkeit erst spät an den Sekundärspeicher ausgelagert. Es kommt daher vermehrt dazu, dass Seiten ausgelagert werden, die zu anderen aktiven Programmen zählen. Sobald auf diese zugegriffen wird, kommt es zu weiteren Page Faults. Die Existenz von Memory Leak kann mithilfe von Werkzeugen festgestellt werden. Ein bekanntes Programm, das häufig zu diesem Zweck eingesetzt wird, ist Valgrind.

Ein weiteres wichtiges Detail der dynamischen Speicherverwaltung in C ist, dass es keine Überprüfung der Grenzen der allokierter Speicherbereiche gibt. Es ist also möglich, über die allokierten Speicherbereiche hinaus auf Daten zuzugreifen. Dies nennt man Buffer Overflow und resultiert laut malloc-Schnittstelle in undefiniertem Verhalten. Aus diesem Grund sollte Buffer Overflow in jedem Fall vermieden werden.

Zusammenfassend lässt sich festhalten, dass der Stack durch seine automatische und schnelle Verwaltung so oft wie möglich verwendet werden sollte. Dynamische Speicherverwaltung bedeutet Mehraufwand für den Programmierer, der zudem mit Risiken wie Memory Leak und Buffer Overflow verbunden ist. Seine Verwendung bietet sich daher nur dann an, wenn die zu verwaltende Datenmenge zu groß ist, um sie effektiv auf dem Stack zu speichern. Der Heap stellt somit eine sinnvolle Erweiterung für die Speicherverwaltung des Stacks dar, die benutzt werden kann, um die Limitierungen des Stacks zu umgehen.

5 Optimierung der Speicherverwaltung

Die Optimierung der Speicherverwaltung findet auf verschiedenen Ebenen statt. Aus Sicht des Entwicklers sind sowohl Optimierungen auf Programmebene als auch auf Compilerebene von besonderem Interesse, da man auf diese direkt Einfluss nehmen kann. Zur Verdeutlichung beider Sichtweisen wird im Folgenden erneut die Programmiersprache C verwendet. Auf Programmebene bietet C einige interessante Datenstrukturen zur Manipulation der Speicherverwaltung und auf Compilerebene stellt der populäre Compiler GCC diverse Optimierungsmöglichkeiten zur Verfügung, auf die der Programmierer zugreifen kann.

5.1 Data Alignment

Wie in Kapitel 2.1 bereits angesprochen, geschieht der Zugriff auf den Speicher nicht Byteweise, sondern auf Segmente fester Größe. Um die Anzahl der Speicherzugriffe zu minimieren und damit die Effizienz zu erhöhen, manipulieren moderne Computer die Positionen der Daten im Speicher, sodass Daten an diesen Segmenten ausgerichtet sind. Dies nennt sich Data Alignment. Das Prinzip dieses Verfahrens soll anhand einiger Abbildungen erklärt werden. Das angeführte Beispiel benutzt eine 32-Bit Architektur. Der Speicherzugriff geschieht daher auf Segmente der Größe 4 Byte.

Das Speichern eines int der Größe 4 Byte im Speicher eines 32-Bit Computers wird eines der Segmente perfekt ausfüllen, da der int die gleiche Größe wie eines der Segmente hat.

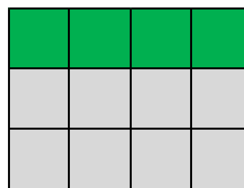

Abbildung 2: int(grün) im Speicher eines 32-Bit Computers

Wenn stattdessen erst ein char (1 Byte), dann ein short (2 Byte) und zuletzt ein int (4 Byte) gespeichert wird, führt dies dazu, dass die Daten des int über zwei Segmente verteilt werden. Man sagt, die Daten sind misaligned.

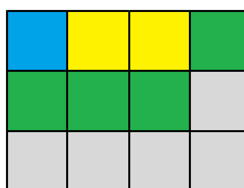

Abbildung 3: Speicher mit jeweils einem char(blau), short(gelb) und int(grün)

Um den int zu lesen, werden nun zwei verschiedene Speicherzugriffe benötigt, da die Daten über zwei Segmente verteilt sind. Dies bedeutet, dass sich die Dauer für das Lesen des Wertes im Vergleich zu dem Fall in Abbildung 1 verdoppelt hat. Um dies zu verhindern und den gesamten int mit nur einem einzigen Speicherzugriff lesen zu können, wird im Speicher ein Leerraum erzeugt. Dieser Vorgang nennt sich Padding und sorgt dafür, dass der int komplett in einem Segment liegt und somit mit nur einem Zugriff gelesen werden kann.

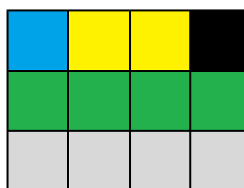

Abbildung 4: Alignment der Daten durch Padding(schwarz)

Die Daten in Abbildung 3 sind nun aligned und jede Variable ist mit nur einem Speicherzugriff lesbar. Das Padding des Speichers wird in der Regel automatisch durch den Compiler übernommen. Für aufeinanderfolgende Variablendeklarationen optimiert der Compiler auch die Reihenfolge der Variablen im Speicher um möglichst wenig Speicherplatz zu verbrauchen.

In C bildet die Datenstruktur struct eine Ausnahme hierfür. Structs garantieren dem Programmierer, dass die Reihenfolge der enthaltenen Variablen im Speicher der Deklarationsreihenfolge entspricht. Der Compiler optimiert diese deshalb nicht automatisch. Abbildung 5 zeigt die Speicherung eines char, int und shorts in dieser Reihenfolge.

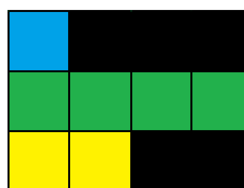

Abbildung 5: Speicher mit jeweils einem char, int und short nach Padding

Um für das Alignment der Daten zu sorgen, wird hier vor und nach dem int Padding angewendet. Wie man im Vergleich mit Abbildung 4 sieht, ist dies nicht optimal, da für das Speichern derselben Daten auch nur zwei Segmente benutzt werden könnten. Die Deklaration dieser drei Datentypen würde durch den Compiler normalerweise dahin optimiert werden, durch das Benutzen eines structs geschieht dies hier jedoch nicht. In diesem Fall liegt es also am Programmierer, für die optimale Reihenfolge der Variablen zu sorgen. Diese lässt sich allgemein erreichen, indem man die größten Datentypen zuerst deklariert. In obigem Beispiel bedeutet das, erst den int, dann den short und zuletzt den char zu deklarieren.

5.2 Compileroptimierung

Moderne Compiler stellen neben der Übersetzung von Programmen in Maschinencode zahlreiche Codeoptimierungen zur Verfügung. Einige dieser Optimierungen werden im Folgenden anhand des C-Compiler GCC erläutert. Dieser stellt verschiedene Optimierungsstufen zur Verfügung, wobei die einzelnen Optimierungsschritte der Optionen relativ undurchsichtig sind. Einige Methoden und generelle Funktionalitäten der Stufen lassen sich aber trotzdem feststellen:

- O0
  - Führt keine Optimierung durch
- O1
  - Versucht, Laufzeit des Programms und Größe des Codes zu reduzieren, ohne die Übersetzungszeit stark zu erhöhen
- O2
  - Mehr Optimierungen als O1, aber kein Inlining oder Loop Unrolling. Laufzeit sinkt aber Übersetzungszeit steigt
- O3
  - Wie O2, aber zusätzlich mit Inlining und Loop Unrolling
- Os
  - Wie O2, aber ohne Optimierungen, die die Codegröße erhöhen

Abhängig von dem Ziel der Kompilierung kann eine dieser Optionen ausgewählt werden. O0 wird in der Regel nur dann angewendet, wenn der Compiler nichts an dem Code verändern soll. Dies ermöglicht die Nutzung eines Debuggers für die Fehlersuche und wird meistens für Programme benutzt, die sich noch in der Entwicklung befinden. O1 sorgt für geringere Laufzeit und geringere Größe des Programms, bietet aber im Vergleich zu O2 keine wesentlichen Vorteile und wird deshalb nur selten verwendet. O2 wiederum sorgt für eine deutliche Reduzierung der Programmlaufzeit und beinhaltet einige Optimierungsschritte wie das oben beschriebene Data Alignment für Funktionen, Sprünge und Schleifen sowie die Entfernung gemeinsamer Teilausdrücke zur Reduzierung der Codegröße. Zudem werden Teile des Codes umgeordnet, um beispielsweise die Anzahl der Sprünge bei Verzweigungen zu reduzieren.

Diese Option wird meistens benutzt, wenn ein Profiler für das Programm benutzt wird, weil einige der Optimierungen in O3 mit Profilern nicht kompatibel sind. Für den Betrieb wird hingegen meistens die Optimierungsstufe O3 verwendet. Zusätzlich zu allen Funktionen von O2 sorgt diese Option zusätzlich für Inlining und Loop Unrolling. Inlining sorgt dafür, dass Aufrufe kurzer Funktionen durch die Körper selbiger Funktion ersetzt werden. Loop Unrolling wiederum ersetzt Schleifen mit wenigen Iterationen durch die mehrfache Wiederholung des Codes im Schleifenkörper. Beide Optimierungen reduzieren die Programmlaufzeit, da sie Funktionsaufrufe bzw. Schleifeniterationen entfernen, erhöhen dafür aber die Größe des Programms. Die letzte der oben angeführten Optimierungsoption ist Os. Diese legt den Fokus auf die Codegröße und führt dafür daher die Optimierungen, die eine verringerte Programmlaufzeit auf Kosten der Programmgröße erzielen, nicht aus. Sie kann verwendet werden, wenn das Programm so klein wie möglich sein soll.

Moderne Compiler nehmen dem Programmierer viel Arbeit ab. Zusätzlich erlaubt die Optimierung durch den Compiler, dass der Programmierer seinen Fokus auf die gute Lesbarkeit seines Codes legen kann, weil dieser ohnehin bei seiner Übersetzung in Binärcode optimiert wird. Da gute Wartbarkeit des Codes in der Realität ein wichtiges Kriterium ist, werden Compileroptimierungen sehr gerne und häufig benutzt. Trotzdem gibt es einige Möglichkeiten für den Programmierer, sein Programm ohne Verringerung der Lesbarkeit zusätzlich zu optimieren.

5.3 Weitere Optimierungsmöglichkeiten

Obwohl Compiler Programmierern heutzutage viel Arbeit abnehmen, gibt es verschiedene Möglichkeiten, manuell Programme zu optimieren. Eine simple Maßnahme ist die Auswahl optimaler Algorithmen und Datenstrukturen in Bezug auf Zeit- und Platzkomplexität. Zusätzlich bieten viele Programmiersprachen verschiedene Datentypen mit unterschiedlichen Größen und Wertebereichen. Beispielsweise beinhaltet C mehrere verschiedene Datentypen, um ganze Zahlen zu repräsentieren. Wenn feststeht, dass für eine Variable nur der Wertebereich von 0 bis 255 benötigt wird, kann so statt einem int der Datentyp uint8_t verwendet werden. Dieser kann zwar nur den oben genannten, begrenzten Wertebereich annehmen, braucht im Speicher aber auch deutlich weniger Platz.

Auch die Verwaltung multidimensionaler Arrays kann auf Laufzeit optimiert werden. Diese werden unabhängig von ihrem logischen Aufbau sequentiell im Speicher gehalten. Wird nun auf dieses Array zugegriffen, wird ein Teil des umliegenden Inhalts in den Cache geladen. Die Funktionalität des Caches kann ausgenutzt werden, indem man entsprechend der Position der Daten im Speicher über den Array iteriert. Dadurch werden die Folgedaten bei jedem Zugriff schon im Voraus in den Cache geladen und die Laufzeit kann deutlich reduziert werden. In der Praxis bedeutet dies, dass man über die innerste Schleife zuerst iterieren sollte, um die Dauer für Speicherzugriffe zu minimieren.

6 Zusammenfassung

Die effiziente Verwaltung des Speichers ist von entscheidender Bedeutung, um die Leistung von Computern zu erhöhen. Aus diesem Grund bestehen moderne Speicherarchitekturen aus verschiedenen Schichten, die dafür sorgen, dass sowohl geringe Zugriffszeiten als auch die Speicherung großer Datenmengen gleichzeitig möglich sind. Die automatische Verwaltung von Programmdaten mittels des Stacks nimmt Programmierern viel Arbeit ab und ist zudem sehr effizient. Da diese aber nicht für alle Anwendungsfälle ausreicht, haben Programmierer zusätzlich die Möglichkeit, dynamisch Speicher zu verwalten. Dies erlaubt eine manuelle Organisation der Programmdaten, die sich sowohl durch ihren großen Manipulationsfreiraum als auch durch ihren deutlich erhöhten Speicherplatz auszeichnet. Allerdings ist dynamische Speicherverwaltung mit Sorgfalt zu behandeln, da ihre fehlerhafte Handhabung gewisse Risiken mit sich bringt.

Die Optimierung des Programmcodes zur Verbesserung der Laufzeit und des Speicherbedarfs geschieht heutzutage größtenteils durch Compiler. Da Maßnahmen wie Umstrukturierungen des Codes, Data Alignment und Inlining nicht manuell durch den Programmierer umgesetzt werden müssen, können Programme mit einem erhöhten Fokus auf gute Lesbarkeit geschrieben werden. Trotzdem gibt es auch einige Optimierungsmöglichkeiten, über die sich Programmierer Gedanken machen müssen. So kann durch die Auswahl effizienter Algorithmen, Datenstrukturen und Datentypen die Performanz eines Programmes erhöht werden. Auch das Ausnutzen der Cache-Lokalität kann die Programmlaufzeit verbessern. Daher ist ein grundlegendes Verständnis moderner Speicherarchitekturen auch für Programmierer von Bedeutung.

Bibliographie

Paul Gribble. Memory: Stack vs Heap. Summer 2012.

Bharat Kinariwala, Tep Dobry. Programming in C. University of Hawai`i at Manoa, 1993.

Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. Beyond Physical Memory: Policies. Arpaci-DusseauBooks, 2014.

Thomas Schwarz. Buffer Overflow Attack. Santa Clara University, 2014.

Song Ho Ahn. Data Alignment. 2011.

Emertxe Information Technologies Pvt Ltd. Embedded C – Optimization techniques. 2014-3-25.