Speichermanagement

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Speicher ist eine wichtige Ressource für Softwaresysteme. Viele nicht-funktionale Eigenschaften hängen vom angemessenen Umgang mit Speicher ab.
Wir betrachten grundlegende Aspekte der Speicherverwaltung:

- Verwendung von Speicher
- (Automatische) Speicherbereinigung

Wie viel Speicher ein Programm (in Abhängigkeit von der Eingabe) genau braucht, hängt von Details der Sprachimplementierung und Plattform ab. In diesem Kapitel diskutieren wir, wieviel Speicher für die Repräsentierung von Daten benötigt wird. Wir diskutieren außerdem die unterschiedlichen Ansätze zum Speichermanagement in C und Java.

Lernziele dieses Kapitels:

- Zu beschreiben, wofür Speicher bei der Ausführung eines Programms benötigt wird.
- Die Notwendigkeit und die Funktionsweise von Speicherbereinigung zu erläutern.

1 Speicherverwendung

Wie viel Speicher braucht ein Programm?
Wofür wird Speicher benötigt?
Wie ist der Speicher organisiert?
Wie wird der Speicher verwaltet?

Ein Programm benötigt für die Ausführung Speicher, um verschiedene Daten und Information zu speichern:

- Programmcode (unabhängig von Eingabedaten)
- Verwaltung von Prozedur-/Methodenaufrufen
- Werte, Objekte, Arrays, etc.

Der tatsächliche Speicherbedarf hängt von der konkreten Laufzeitumgebung ab. In Java beispielsweise kann sich der Speicherbedarf für die Verwaltung von Methodenaufrufen und Objekten in verschiedenen Implementierungen und Versionen der JVM unterscheiden. Wir betrachten im Folgenden typische Größen.

Speicherbedarf: Primitive Datentypen Die Spezifikation der Java Virtual Machine[1] legt den Speicherbedarf (und damit den Wertebereich) für die primitiven Datentypen folgendermaßen fest:

Datentyp	Bytes (1 Byte = 8 Bit)
boolean	1
char	2
int	4
float	4
long	8
double	8

In C ist der Speicherbedarf für Werte primitiver Datentypen *maschinenabhängig*.
Die Größe einer Referenz auf ein Objekt in Java bzw. eines Pointers in C ist typischerweise abhängig von der konkreten Prozessorarchitektur. In 32-Bit Architekturen können Speicheradressen in 4 Byte, in 64-Bit Architekturen in 8 Byte kodiert werden.

Speicherbedarf: Strukturen und Objekte Um den Speicherbedarf einer Struktur in C zu bestimmen, ermittelt man den Speicherbedarf für die einzelnen Komponenten. [2]
In Java benötigt jedes Objekte Speicher für jedes seiner Objektattribute (abhängig von dessen Typ) sowie einmalig 8 Byte Overhead zur Verwaltung des Objekts (z.B. Information zur Klasse).[3]

Speicherbedarf: Arrays Wenn ein Array N Elemente eines Typs enthält und M Byte zum Speichern eines Elements benötigt werden, braucht das Array mindestens $N \times M$ Byte zur Repräsentierung der Elemente. In Java ist darüber hinaus zusätzlicher Speicher benötigt, um u.a. Größe des Arrays zu speichern.

2 Speicherverwaltung

Die Speicherorganisation ist bei den meisten prozeduralen Programmiersprachen ähnlich (vgl. Kap. 20):

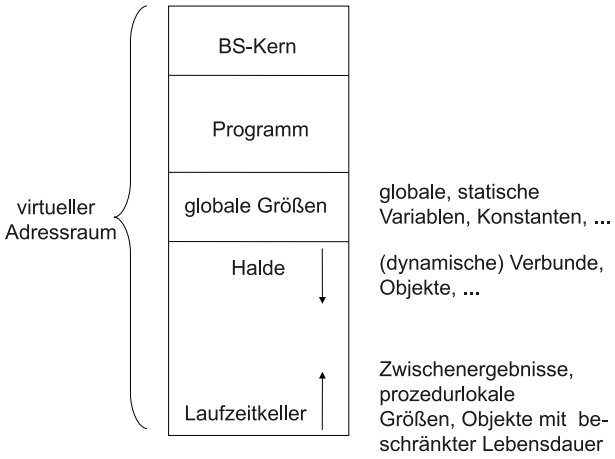
- Der ***Programmbereich*** enthält den kompilierten Programmcode Maschinenbefehle.

[1] Für numerische Werte siehe z.B. Abschnitt 2.3 der JVM Spezifikation `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`

[2] Der tatsächliche Speicherbedarf kann größer sein, da bisweilen Leerräume eingefügt werden (*alignment*), siehe Vorlesung ReSy.

[3] Der Overhead ist abhängig von der jeweiligen verwendeten JVM. Wir beschreiben hier ein typisches Layout.

- Im ***globalen Datenbereich*** befinden sich alle Daten, die vom Beginn des Programms bis zum Programmende verfügbar sind (u.a. globale Variablen, String-Literale).
- Im ***Stack*** (dt. Laufzeitkeller) werden die Funktionsinkarnationen mit ihren lokalen Variablen verwaltet.
- Der ***Heap*** (dt. Halde) stellt Speicherplatz für die dynamischen Speicherverwaltung zur Verfügung.



Der Bereich für den Programmcode und Betriebssystem-Kern werden während der Programmausführung i.d.R. nicht verändert. Der globale Speicher und Stack werden automatisch verwaltet; der Compiler erzeugt dafür Code (evtl. Unterstützung von der Laufzeitumgebung).
Der Heap wird je nach Programmiersprache aber sehr unterschiedlich verwaltet. Die Verwaltung basiert auf den folgenden Operationen:

- ***Allokation*** (engl. *allocation*): Anfordern von Speicher
- ***Deallokation*** (engl. *deallocation*): Freigabe von Speicher

Wenn angeforderter Speicher nicht mehr verwendet wird, wird dies ***Speicherleck*** (engl. memory leak) genannt. Dies kann bei länger laufenden Programmen zu Problem führen, wenn das Programm mit der Zeit immer mehr Speicher benötigt. Wenn kein Arbeitsspeicher vorhanden ist, wird das Betriebssystem gezwungen zu handeln und muss entweder Daten vom Arbeitsspeicher auf die Festplatte speichern oder Programme plötzlich beenden. Das Auslagern auf die Festplatte führt dabei in der Regel zu Performance-Problemen, da die Zugriffszeit deutlich höher ist, als beim Arbeitsspeicher.
Nicht mehr verwendeter Speicher sollte daher immer freigegeben werden. Man unterscheidet bei der Freigabe von Speicher zwei unterschiedliche Herangehensweise:

- Speicherbereinigung durch den Programmierer (Beispiel: C)
- Automatische Speicherbereinigung (Beispiel: Java)

2.1 Speicherbereinigung durch den Programmierer

In C ist die Speicherverwaltung in der Verantwortung des Programmierers. Speicherbereiche können auf dem Stack oder auf dem Heap reserviert werden. Die Standardbibliothek `stdlib` stellt Funktionen (`malloc`, `calloc`, `realloc`, `free`) zur Verwaltung des Heaps bereit. Der Stack wird von C automatisch verwaltet. Dies ermöglicht eine sehr effiziente und flexible, aber fehleranfällige Benutzung von Speicher.

2.2 Automatische Speicherbereinigung

Manuelle Verwaltung des Speichers bedeutet zusätzlichen Programmieraufwand und ist eine potentielle Fehlerquelle, die oft schwierig zu debuggen ist (z.B. mehrfaches Freigeben von Speicherbereichen, fehlende Initialisierung, etc.).
Immer mehr Programmiersprachen haben daher ***automatischer Speicherbereinigung*** (engl. *automatic garbage collection*). Dabei wird periodisch oder bei Bedarf ermittelt, welche Objekte (o.ä.) nicht mehr erreichbar sind. Diese Objekte werden dealloziert; gegebenenfalls wird außerdem der Speicherbereich umgeordnet und kompaktifiziert (defragmentiert).
Der wichtigste Vorteil ist dabei die Vereinfachung der Programmierung. Moderne Laufzeitumgebungen erlauben es automatische Speicherbereinigung parallel zur Ausführung des Programms auszuführen, um die Programmausführung nicht unterbrechen zu müssen.

Erreichbarkeit Ein Objekt X heißt von einer Variablen v ***direkt erreichbar***, wenn v eine Referenz auf X enthält. X heißt von v ***erreichbar***, wenn es von v direkt erreichbar ist oder wenn es ein Objekt Y mit Attribut w gibt, so dass X von w direkt erreichbar ist und Y von v erreichbar ist.
Die Menge der ***Wurzelvariablen*** zu einem Ausführungszustand A umfasst alle globalen Variablen (in Java z.B. auch Klassenvariablen) sowie die lokalen Variablen und Parameter der aktuellen Prozedurinkarnationen. Ein Objekt heißt ***erreichbar*** *in einem Ausführungszustand* A, wenn es von einer Wurzelvariablen zu A erreichbar ist.
Sind Objekte nicht mehr erreichbar, kann deren Speicherplatz freigegeben werden.

Speicherlecks sind also auch mit automatischer Speicherbereinigung noch möglich. Programmierer müssen darauf achten, dass nicht mehr benötigte Objekte nicht mehr erreichbar sind. Ein Beispiel dafür ist die `remove`-Methode in der Klasse `ArrayList` der Java Collections. Hier wird nach dem Löschen aus der Liste das nicht mehr benötigte Array-Element auf `null` gesetzt, um zu verhindern, dass es noch als erreichbar gezählt wird.

```
public E remove(int index) {
    rangeCheck(index);
```

```
    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

Beispiel: Automatische Speicherbereinigung Das folgende Programm erzeugt Objekte (hier: Arrays).

```
public static void main (String[] args) {
  int count = 1;
  while (true) {
    int[] ar = new int[1000000];
    StdOut.println(count++);
  }
}
```

Dieses Programm bekommt keine Speicherprobleme. Bei jedem Schleifendurchlauf wird ein Array alloziert, welches von einer Variable `ar` referenziert wird. Das Array aus dem vorherigen Durchlauf ist nicht mehr erreichbar, da die Variable `ar` nur während des aktuellen Durchlaufs verwendet werden kann. Der Speicherbereich des Arrays kann daher nach jedem Schleifendurchlauf vom Garbage Collector freigegeben werden.
Im folgenden Programm bleiben die erzeugte Objekte jedoch erreichbar:

```
public static void main (String[] args) {
  List<int[]> la = new LinkedList<int[]>();
  int count = 0;
  while (true) {
    la.add(new int[1000000]);
    count++;
    StdOut.println(count);
  }
}
```

Dieses Programm terminiert mit einem `OutOfMemoryError`, da die Arrays über die Listenreferenz `la` auch nach Beendigung der Schleife erreichbar sind.

Bitte testen Sie das Programm **nicht** auf den SCI-Rechnern!

3 Zusammenfassung

Das Laufzeitverhalten eines Programms wird bestimmt durch:

- die Anweisungen des Programms
- den Übersetzer
- die Laufzeitumgebung (insbesondere Speicherverwaltung und Systemumgebung)

Zur Gesamtbeurteilung des Laufzeitverhaltens eines Softwaresystems muss auch die Systemumgebung berücksichtigt werden. Hierzu zählen außerdem Aspekte wie:

- Benutzerinteraktion
- Anzahl von Benutzern
- Kommunikationszeiten
- Laufzeitverhalten der Plattform
- Interaktion mit anderen Systemen

Speicherverwaltung ist aufwendig und kostet Zeit, wenn der verfügbare Speicher knapp wird. Der Garbage Collector muss dann häufig aufgerufen werden; und auch das Lokalisieren ausreichend großer freier Speicherbereiche wird aufwendiger. Problematisch ist dies insbesondere bei Echtzeitanforderungen.
In der Praxis ist der Aufwand von Programmen oft nicht einfach abzuschätzen, weil der Speicherverbrauch der Bibliotheksklassen und anderer fremder Programmteile häufig nicht klar spezifiziert ist und die Details der Speicherverwaltung eine wichtige Rolle spielen.