Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - AG Praktische Informatik

Speicherverwaltung - Algorithmen

Eines der wichtigsten Betriebsmittel ist der Hauptspeicher . Die Verwaltung der Speicherressourcen ist deshalb auch ein kritisches und lohnendes Thema, das über die Leistungsfähigkeit eines Rechnersystems entscheidet. Dabei unterscheiden wir drei Bereiche, in denen unterschiedliche Strategien zur Speicherverwaltung angewendet werden:

1. Anwenderprogramme
2. Hauptspeicher
3. Massenspeicher

1. Anwenderprogramme

- Der Speicherraum eines Prozesses
- Die Hauptaufgabe besteht hierbei darin, interne Speicherbereiche des Prozesses (beispielsweise den Heap) mit dem Wissen um die speziellen Speicheranforderungen des Programms optimal zu verwalten.
- Dies erfolgt durch spezielle Programmteile, z. B. den Speichermanager, oder durch sog. garbage collection Programme in Virtuellen Maschinen.

2. Hauptspeicher

- Hier besteht das Problem in der optimalen Aufteilung des gesamten knappen Hauptspeicherplatzes auf die einzelnen Prozesse.
- Die Aufgabe wird in der Regel durch spezielle Hardwareeinheiten unterstützt (MMU).
- Besonders in Multiprozessorsystemen ist dies wichtig, um Konflikte zu vermeiden, wenn mehrere Prozessoren auf dieselben Speicherbereiche zugreifen wollen.
    - In diesem Fall helfen Techniken wie Non-Uniform Memory Access (NUMA) weiter, die zwischen einem Zugriff des Prozessors auf lokalen und auf globalen Speicher unterscheiden.

3. Massenspeicher

- Abgesehen von der Dateiverwaltung, die noch besprochen wird, gibt es auch spezielle Dateien (z. B. bei Datenbanken), bei denen der Platz innerhalb der Datei optimal eingeteilt und verwaltet werden muss.
- Ein Beispiel dafür ist die Swap-Datei, auf die Prozesse verlagert werden, die keinen Platz mehr im Hauptspeicher haben.

Direkte Speicherbelegung

- In den frühen Tagen der Computernutzung war es üblich, für jeden Job den gesamten Computer mit seinem Speicher zu reservieren.
  - Der Betriebssystemkern bestand dabei oft aus einer Sammlung von Ein- und Ausgabeprozeduren, die als Bibliotheksprozeduren zusätzlich an den Job angehängt waren.
  - Musste man einen Job zurückstellen/wechseln, um erst einen anderen durchzuführen, so wurden alle Daten des dazugehörenden Prozesses auf den Massenspeicher verlagert und dafür die neuen Prozessdaten vom Massenspeicher in den Hauptspeicher befördert. (Swapping)
- Viele Nachteile (Overhead, "Reibung"). Besser die Daten mehrerer Prozesse gleichzeitig im Speicher halten, sobald genügend Speicher vorhanden war.
  - Allerdings brachte dies weitere Probleme mit sich: Wie kann man den Speicher so aufteilen, dass möglichst viele Prozesse Platz haben? Diese Frage stellt sich prinzipiell nicht nur für den Hauptspeicher , sondern gilt natürlich auch für den Auslagerungsplatz auf der Festplatte, den swap space.

Zuordnung durch feste Tabellen

Die einfachste Möglichkeit besteht darin, ein verkleinertes Abbild des Speichers zu erstellen: die Speicherbelegungstabelle . Jede Einheit einer solchen Tabelle (beispielsweise ein Bit) ist einer größeren Einheit (z. B. einem 32-Bit-Wort) zugeordnet. Ist das Wort belegt, so ist das Bit Eins, sonst Null.

[BSG]

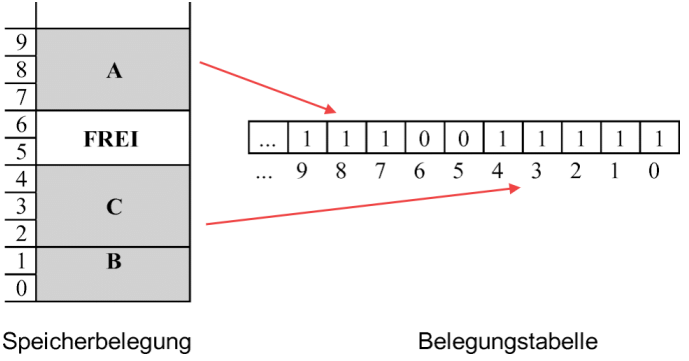


Abb. 1. Eine direkte Speicherbelegung und ihre Belegungstabelle

Zuordnung durch feste Tabellen

- Eine solche Belegungstabelle benötigt also bei 32 MiB Speicher selbst 1 MiB.
- Soll ein freier Platz belegt werden, so muss dazu die gesamte Tabelle auf eine passende Anzahl von aufeinanderfolgenden Nullen durchsucht werden.
  - Komplexitätsklasse O(N) bei N Bytes
- Besser: Paging (Einteilung des Speichers in Blöcke), reduziert die Tabellengröße

Zuordnung durch verzeigerte Listen

- Der belegte bzw. freie Platz ist meist wesentlich länger als die Beschreibung davon.
  - Die Belegung des gesamten Hauptspeichers mit belegten und freien Plätzen kann dünn besetzt sein.
- Aus diesem Grund lohnt es sich, anstelle einer festen Tabelle eine Liste von Einträgen über die Speicherbelegung zu machen, die in der Reihenfolge der Speicheradressen über Zeiger miteinander verbunden sind.

- Ein Eintrag der Liste besteht aus drei Teilen:
  - der Startadresse des Speicherteils,
  - der Länge und
  - dem Index (Zeiger) des nächsten Eintrags.

[BSG]

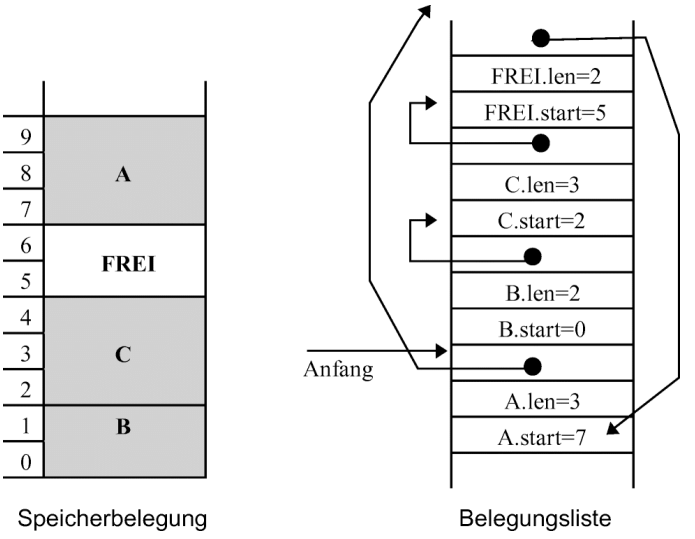


Abb. 2. Eine Speicherbelegung und die verzeigerte Belegungsliste

Zuordnung durch verzeigerte Listen

- Eine solche Belegungsliste lässt sich auch in zwei Listen zerteilen:
  - eine für die belegten Bereiche, deren Einträge im Fall von Prozessen mit dem Prozesskontrollblock PCB verzeigert sind, und
  - eine nur für die unbelegten Bereiche.
- Ordnen wir diese Liste noch zusätzlich nach der Größe der freien Bereiche, so muss normalerweise nie die ganze Liste durchsucht werden.
- Allerdings wird auch dadurch das Verschmelzen angrenzender, freier Bereiche erschwert. Eine doppelte Verzeigerung ermöglicht das Durchsuchen der Liste in beiden Richtungen.

Beispiel: Heap Liste

```
typedef unsigned int memoryptr;
typedef int size;
struct heap_entry {
  memoryptr  adress;
  memorysize size;
  struct heap_entry next;
};
struct heap_entry *allocated;
struct heap_entry *free;
```

Def. 1. Eine listenbasierte Speicherverwaltung ist beispielsweise für das Speichermanagement eines Heaps sinnvoll. Es wird meistens in Blockgrößen gerechnet (z.B. 4kB), nicht in Byteeinheiten

Es ist einfacher und schneller, nur eine einzige Liste zu führen, die Liste der freien Plätze.

- Allerdings ist es sicherer, auch eine Liste der belegten Plätze zu führen, um bei der Platzrückgabe die Angaben über die Lage und die Länge des zurückgegebenen Platzes überprüfen zu können und so Fehlprogrammierungen des Benutzerprogramms aufzudecken.
- Besonders in der Debugging-Phase ist dies durchaus sinnvoll; eine solche Überprüfung kann dann in späteren Phasen bei der Laufzeitoptimierung abgeschaltet werden.

Zuordnung durch Baumstrukturen

Listen haben eine Such- und Bearbeitungslaufzeit der Klasse O(N) bei N Elementen, Bäume hingegen (im Mittel) von log(N)N.

- Daher bieten sich Bereichsbäume an um belegte und freie Speicherbereiche zu verwalten.
- Jedoch bedürfen wie in der Natur Bäume einer Gartenpflege, Listen i.A. nicht.
  - Ein Baum ist nur dann effizient wenn er ausbalanziert ist.
  - Wenn er balanziert ist, dann ist die Suche schneller als bei Listen
  - Aber beim Speichermanagement müssen häufig Änderungen vorgenommen werden, was zum Ungleichgewicht führen kann, was dann wieder korrigiert werden muss.
  - Extremfall eines Baums: Die Liste!

Belegungsstrategien

Unabhängig von dem Mechanismus der Speicherbelegungslisten gibt es verschiedene Strategien, um aus der Menge der unbelegten Speicherbereiche den geeignetsten auszusuchen.

- Ziel der Strategien ist es, die Anzahl der freien Bereiche möglichst klein zu halten und ihre Größe möglichst groß.

Die wichtigsten Strategien sind:

FirstFit
: Das erste, ausreichend große Speicherstück, das frei ist, wird entsprechend belegt. Dies bringt meist ein Reststück mit sich, das unbelegt übrig bleibt.

NextFit
: Die FirstFit-Strategie führt dazu, dass in den ersten freien Bereichen nur Reststücke (Verschnitt) übrigbleiben, die immer wieder durchsucht werden. Um dies zu vermeiden, geht man wie FirstFit vor, setzt aber beim nächsten Mal die Suche an der Stelle fort, an der man aufgehört hat.

Belegungsstrategien

BestFit

Die gesamte Liste bzw. Tabelle wird durchsucht, bis man ein geeignetes Stück findet, das gerade ausreicht, um den zu belegenden Bereich aufzunehmen.

WorstFit

Sucht das größte vorhandene freie Speicherstück, so dass das Reststück möglichst groß bleibt.

QuickFit

Für jede Sorte von Belegungen wird eine extra Liste unterhalten. Dies gestattet es, schneller passende Freistellen zu finden. Werden beispielsweise in einem Nachrichtensystem regelmäßig Nachrichten der Länge 1 KB verschickt, so ist es sinnvoll, eine extra Liste für 1-KB-Belegungen zu führen und alle Anfragen damit schnell und ohne Verschnitt zufriedenzustellen.

[bsdp]

Belegungsstrategien

```
free={addr:0,
      size:N,
      next:null}

function FirstFit(size) {
  block=free
  while(block.size<size) {
   block=block.next
  }
  if (!block)
    return null;
  addr=block.addr
  block.size-=size
  block.addr+=size
  return addr
}
```

```
last=free
function NextFit(size) {
  if (last && last.size>=size)
    block=last
  else
    block=free
  while(block.size<size) {
   block=block.next
  }
  if (!block)
   return null;
  addr=block.addr
  block.size-=size
  block.addr+=size
  last=block
  return addr
}
```

```
function WorstFit(size) {
  block=free; biggest=null
  while(block) {
    if (block.size>=size) {
      if (!biggest ||
          biggest.size<block.size)
        biggest=block;
    }
    block=block.next
  }
  if (!biggest) return null;
  addr=biggest.addr
  biggest.size-=size
  biggest.addr+=size
  return addr
}
```

Alg. 1. FirstFit versa NextFit versa WorstFit

Belegungsstrategien

Buddy-Systeme

Den Gedanken von QuickFit kann man nun dahin erweitern, dass man für jede gängige Belegungsgröße, am besten Speicher in den Größen von Zweierpotenzen, eine eigene Liste vorsieht und nur Speicherstücke einer solchen festen Größe vergibt.

- Alle Anforderungen müssen also auf die nächste Zweierpotenz aufgerundet werden. Ein Speicherplatz der Länge 280 Bytes = 256 + 16 + 8 = $2^8 + 2^4 + 2^3$ Bytes muss also auf 2^9 = 512 Bytes aufgerundet werden.

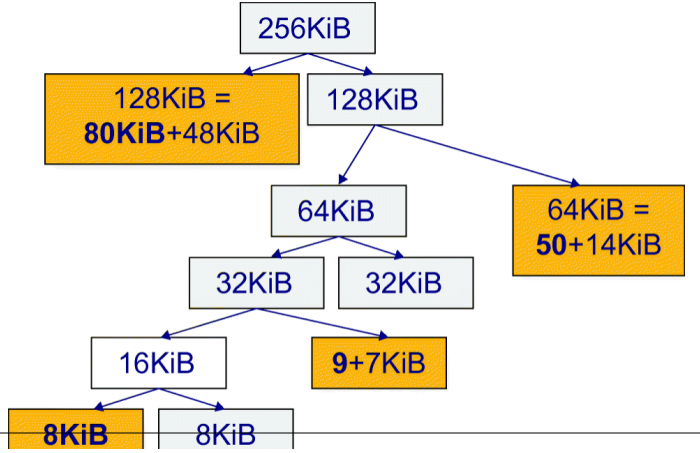
- Ist kein freies Speicherstück der Größe 2^k vorhanden, so muss ein freies Speicherstück der Größe 2^{k+1} in zwei Stücke von je 2^k Byte aufgeteilt werden. Beide Stücke, die Partner (Buddy), sind genau gekennzeichnet: Ihre Anfangsadressen sind identisch bis auf das k-te Bit in ihrer Adresse, das invertiert ist. Beispiel: … XYZ0000 … und … XYZ1000 … sind die Anfangsadressen von Partnern.

- Dies lässt sich ausnutzen, um sehr schnell (in einem Schritt!) prüfen zu können, ob ein freigewordenes Speicherstück einen freien Partner in der Belegungstabelle hat, mit dem es zu einem (doppelt so großen) Stück verschmolzen werden kann, ohne freie Reststücke zu hinterlassen.

Beispiel Buddy-Speicherzuweisung

Angenommen, wir benötigen Speicherstücke der Größen 80 KB, 8 KB, 9 KB, 3 KB und 50 KB in der genannten Reihenfolge von einem Speicherstück von 256 KB.

- Dann lässt sich die Zuordnung ganz gut mit Hilfe eine Binärbaumes visualisieren, bei dem jedes Speicherstück durch seine beiden Buddy notiert ist.
  - Da es sich jedes Stück in zwei Unterstücke aufspalten lässt, bilden alle Speicherstücke zusammen einen Binärbaum.



[bsdp]

- Man sieht, dass jedes Speicherstück, das benutzt wird, meist noch ein unbenutztes Reststück, den Verschnitt, enthält.

Beide Vorgänge, sowohl das Suchen eines passenden freien Stücks (bzw. das dafür nötigen Auseinanderbrechen eines größeren) als auch das Zusammenfügen zu größeren Einheiten lässt sich rekursiv über mehrere Partnerebenen (mehrere Zweierpotenzen) durchführen.

Bewertung der Strategien

Der Vergleich der verschiedenen Strategien führt zu einer differenzierten Bewertung. So stellt sich heraus, dass FirstFit von der Platzausnutzung etwas besser als NextFit und WorstFit ist und überraschenderweise auch besser als BestFit, das dazu neigt, nur sehr kleine, unbelegbare Reste übrig zu lassen. Eine Anordnung der Listenelemente nach Größe der Bereiche erniedrigt dabei die Laufzeit von BestFit und WorstFit.

Wissen wir mehr über die **Verteilung der Speicheranforderungen nach Zeit oder Beleggröße**, so können mit QuickFit und anderen, speziellen Algorithmen bessere Resultate erzielt werden!

- Die Leistungsfähigkeit des Buddy-Systems lässt sich mit folgender Rechnung kurz abschätzen: Nehmen wir an, für den Hauptspeicher der Größe 2n werden alle 2n möglichen Stückgrößen S daraus mit gleicher Wahrscheinlichkeit verlangt (was sicher nicht vorkommt, aber mangels plausiblerer Annahmen vorausgesetzt sei). Dann ist die mittlere Speicheranforderung ⟨S(n)⟩ mit

- Profiling zur Laufzeit kann helfen den richtigen Algorithmus auszuwählen.