Buddy-Speicherverwaltung

Benedikt Möller, Florijan Ajvazi

Frankfurt University of Applied Sciences

Nibelungenplatz 1

60318 Frankfurt am Main

Zusammenfassung

Die folgende Dokumentation gibt einen Überblick über die Entwicklung und Implementierung einer Buddy-Speicherverwaltungssimulation, welche als Bash-Skript umgesetzt wurde. Ziel ist es beliebige Zuweisung und die Freigabe der entsprechenden Allokationen im Konzept der Buddy-Speicherverwaltung zu verdeutlichen.

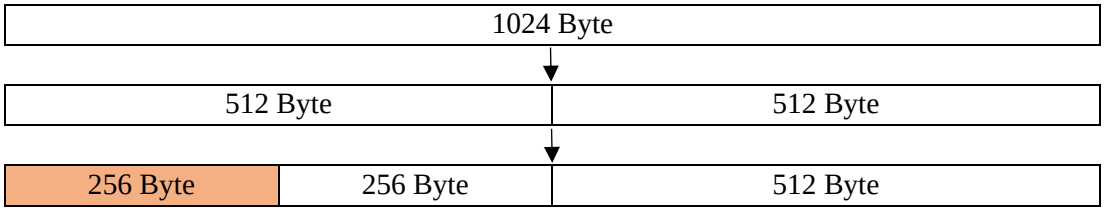
Inhaltsverzeichnis

1 Einleitung

Die Buddy-Speicherverwaltung ist ein Konzept, welches versucht die Nachteile der statischen und dynamischen Partitionierung zu minimieren. Zu den Nachteilen zählen die interne Fragmentierung bei der statischen und der hohe Verwaltungsaufwand bei der dynamischen Variante.

Im Buddy-System gibt es Speicherblöcke der Größe 2^k. Hierbei gilt, $n \leq k \leq m$ wenn n als kleinster zustellbarer und m als größter zustellbarer Block definiert wird. Fordert ein Prozess Speicher an, wird dieser auf die nächst größere Zweierpotenz aufgerundet. Sofern sich der aufgerundete Wert in den vorherig genannten Grenzen befindet, wird ein Block in der Größe 2^k besetzt. Ist kein solcher Block vorhanden, wird versucht, ein Block der Größe 2^{k+1} zu finden und diesen zu halbieren. Die Prozedur wiederholt sich bis 2^m. Falls kein teilbarer Block gefunden wurde, kann kein Speicher alloziert werden. Die aufgeteilten Blöcke werden Buddies genannt und nur genau jene können nach einer Freigabe von Speicher wieder zusammengefasst werden.

Bsp.: Anforderung von 234 Byte (auf 2^k aufgerundet: 256 Byte)



2 Simulationsaufbau

Die Simulation besteht aus einem Skript und kann direkt vom Terminal mit einem optionalen Parameter aufgerufen werden, welcher die gewünschte Gesamtspeichergröße angibt. Sofern keine Zahl einer Zweierpotenz gewählt wurde, wird der gewünschte Speicher auf die nächst Größere gesetzt. Sofern kein Parameter vorhanden ist, wird der Standardwert von 1024 Byte verwendet. Die kleinste adressierbare Blockgröße ist auf 64 Byte festgelegt, kann aber ohne negative Auswirkung im Skript angepasst werden.

```
./buddy.sh <optional: Größe des Gesamtspeichers>
```

Das Skript läuft in einer Schleife und wird Programmseitig nur durch die Option Exit „e“ abgebrochen. In Abbildung 1 wird der schematische Simulationsdurchlauf dargestellt.

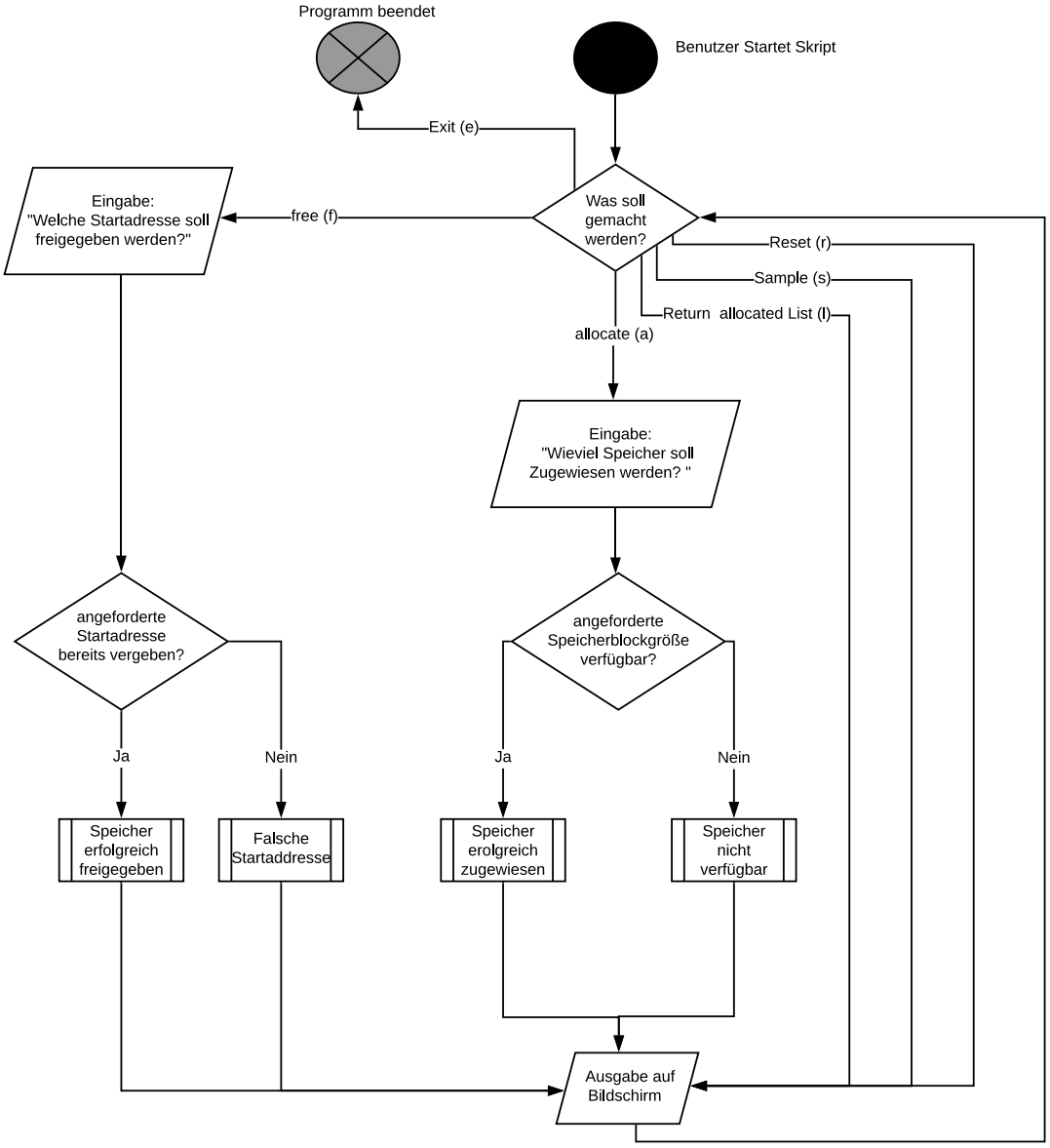


Abbildung 1 - Ablaufdiagramm

3 Implementierung

3.1 Konzept

Um die Buddy-Speicherverwaltung auf Basis eines Bash-Skriptes umsetzen zu können, wurde zuvor eine Lösung in einer Multiparadigmen Programmiersprache getätigt, wodurch die funktionellen Grenzen wesentlich geringer waren.

Das Hauptkonzept baut auf drei Tabellen auf. Eine Tabelle verwaltet die Freien Speicheradressen (folgend *Free Buddy Array*), eine Weitere stellt den reell genutzten Gesamtspeicher dar (folgend *Memory Array*) und eine Tabelle speichert die Ergebnisse der angeforderten Allokation. Der freie Speicher wird genutzt, um eine einfache Verkettung gleich großer Blöcke darstellen zu können, da das *Free Buddy Array* nur einen Wert pro Zeile sinnhaft abspeichern kann. Dies bedeutet, der Wert in dem *Free Buddy Array* verweist auf den Speicherblock im *Memory Array*. Sofern an der Stelle ein Wert ungleich „-1“ gespeichert ist, bedeutet dies, dass weitere Blöcke der gleichen Größe vorhanden sind. Die einfach verkettete Liste (folgend *Linked List*) kann beliebig lang werden. Der letzte freie Block wird schlussendlich durch eine „-1“ gekennzeichnet. Als weiteres Konzept wird ein Kopfbereich (auch Vorspann, engl.: *header*) der Größe eines Bytes an der ersten Stelle des belegten Blocks beansprucht. In diesem wird die Größe des belegten Blocks gespeichert, was für die Deallokation essenziell ist. In der Abbildung sieht man sowohl die *Linked List* als auch die schematische Umsetzung des *Headers*.

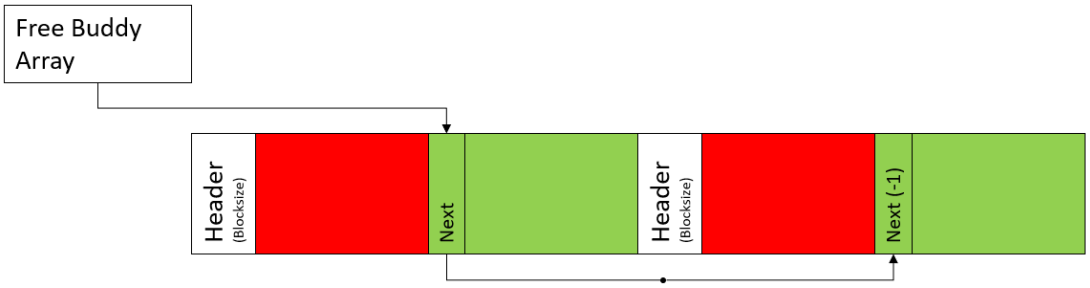


Abbildung 2 - Schematisches Konzept

Diese schematische Darstellung wird durch die folgende Ausgabe in unserem Programm unterstrichen und zeigt die *Linked List* anhand des Beispiels von zwei freien 256 Byte Blöcken.

```
_____________________________
Size    | Free Startaddresses (-1 = no free Block)
64      | -1
128     | -1
256     | 256   | 768   | -1
512     | -1
1024    | -1
_____________________________
Chart (#-used Space // I-Splitpoint)

[################|               |################|               ]
```

Abbildung 3 – Programmauszug Bsp. zu einfach Verketteter Liste

Für Berechnungen in der Bash wurden als besondere Varianten der *Basic Calculator* und arithmetische Operationen, wie die logische Verschiebung (engl.: *logical shift*), bitweises ODER, bitweises UND sowie Negation verwendet. Letztere zeigen bei hardwarenaher Umsetzung ihre enorme Effizienz.

3.2 Funktionen

3.2.1 Speicherzuweisung

Um den Speicher zu allozieren (engl.: *to allocate*), wird die angeforderte Speichergröße (engl.: *page size*) geprüft und gegebenenfalls auf die nächst höhere Zweierpotenz gesetzt. Im Folgenden sorgen IF-Anweisungen dafür, dass invalide Eingaben als Fehlerhafte behandelt werden. Wenn eine Größe kleiner der minimalen Blockgröße angefordert wird, wird automatisch mit der minimalen Größe weitergerechnet. Des Weiteren wird für Berechnungen lediglich der Exponent der Zweierpotenz, welcher äquivalent zur notwendigen Anforderungsgröße ist, verwendet.

Im nächsten Schritt wird in einer FOR-Schleife geprüft, ob bereits ein Bereich in der angeforderten Größe existiert und ob bei extern fragmentiertem Speicher die angeforderte Größe adressiert werden kann. So kann es beispielsweise vorkommen, dass noch zwei Blöcke der Größe 256 Byte frei zur Verfügung stehen, jedoch kein Block der Größe 512 Byte alloziert werden kann.

Darauffolgend wird das Ergebnis der Startadresse gespeichert und der Wert im *Memory Array* in das *Free Buddy Array* als neuer Listenstartpunkt am Anfang hinzugefügt, wodurch die Konsistenz der *Linked List* erhalten bleibt.

Falls kein Buddy in der Größe existiert, wird ab dem generierten Teilungspunkt (engl.: *splitpoint*) zur passenden Buddygröße iteriert und die Teilungen des Speichers vollzogen. Danach wird die Speichergröße im *Header* im *Memory Array* gespeichert. Abschließend werden Ausgabeparameter aufbereitet und da für das Programm nur die tatsächliche Startadresse der Daten relevant ist, der Rückgabewert, entsprechend der Headergröße, um 1 inkrementiert.

3.2.2 Speicherfreigabe

Am Anfang der Funktion wird die Startadresse des Blocks, welche den *Header* einschließt, hergestellt und geprüft, ob an der freizugebenden Startadresse vorher tatsächlich eine Allokation stattfand.

In der folgenden WHILE-Schleife wird der Zusammenschluss (engl.: *merge*) der Buddies und als eigentliches Ziel der Deallokation die Freigabe des nicht weiter benötigten Speichers behandelt.

Am Anfang werden drei weitere Hilfsvariablen eingeführt, mit welchen das Konzept der *Linked List* erhalten und konsistent bleiben soll, sowie die zusammenzuführenden Buddies identifiziert werden können. In einer weiteren inneren Schleife wird anhand der *Linked List* geprüft, ob zwei der in der Liste vorhanden Blöcke Buddies sind. Dies wird mit der später erläuterten Funktion der Buddyprüfung umgesetzt. Sobald ein passender Buddy gefunden wurde, wird die Schleife vorzeitig abgebrochen. Falls das Ende der *Linked List* erreicht ist, ohne dass ein passender Buddy gefunden wurde, wird die Schleife abgebrochen. In dem Fall das kein Buddy zum Zusammenschluss existiert, wird durch einen Eintrag im nicht genutzten Bereich des *Memory Arrays* der Zeiger auf das neue Element gesetzt und ein Vermerk in der Free Buddy Liste auf die neue freie Startadresse hinterlegt. Danach wird die Schleife und somit die Funktion erfolgreich beendet. Dies ist der einzige Ausstiegspunkt der Funktion.

Falls ein Buddy zum Zusammenschluss existiert wird im Folgenden sichergestellt, dass das Ergebnis der neuen Startadresse immer die niedrigere Startadresse zurückgibt.

Am Ende der äußeren Schleife werden die Einträge der neuen Buddies in das *Memory Array* und in die *Free Buddy Array* geschrieben. Da im Folgenden die nächst größeren Bereichen auf einen Zusammenschluss geprüft werden müssen, werden die entsprechenden Variablen inkrementiert und die Schleife wird von neuem durchlaufen, bis es keine zu verschmelzenden Buddies mehr vorhanden sind.

3.2.3 Buddyprüfung

Die Funktion der Buddyprüfung wurde der Übersicht halber aus der Speicherfreigabe herausgezogen.

Als Leitsatz zur Buddyprüfung gilt: Zwei Blöcke der Größe 2^i sind genau dann Buddies, wenn sich ihre Speicheradressen lediglich in der Bitposition i unterscheiden.

An dem folgenden Beispiel, welches in einer übersichtlichen tabellarischen Form zu vergleichende Speicheradressen binär darstellt, wird deutlich, dass sich die Adressen verglichen zu Adresse 768 jeweils nur an einer Position unterscheiden. Der korrekte Buddy identifiziert sich jedoch an der Position 2^8, was die zu vergleichende Größe beschreibt.

Beispiel: *Vergleichende Größe* = 2^8, *Darstellung der Startadressen in Binärform*

Speicheradressen	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	0	1	0	0	0	0	0	0	0	0
768	1	1	0	0	0	0	0	0	0	0
512	1	0	0	0	0	0	0	0	0	0

Im *High Level Design* wird dieses Beispiel mit der Darstellung eines Binärbaums noch plakativer. So können nur die Blöcke Buddies sein, die auch vom selben Knotenpunkt ausgehen.

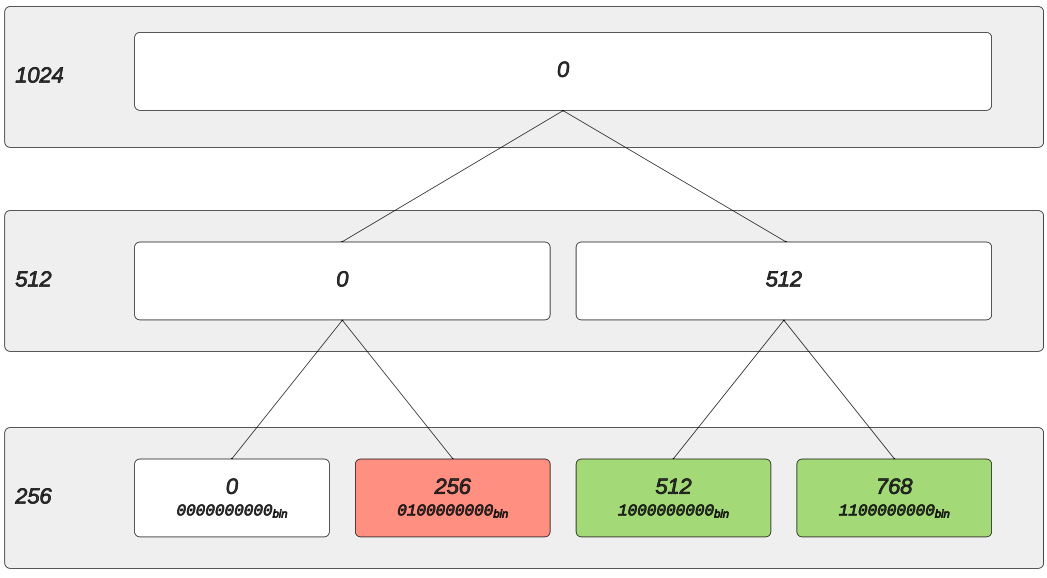


Abbildung 4 - Binär Baum zum Buddyprüfungsbeispiel

In unserer Umsetzung erhalten wir drei Übergabeparameter aus der Überliegenden Funktion. Zum einen die zwei zu prüfenden Adressen und zum anderen den Exponenten des freigebenden Buddys. Aus diesen Daten wird eine Maske generiert, welche die variierende Bitposition anzeigt. Folgend wird durch eine bitweise ODER-Operation und den direkten Vergleich der resultierenden maskierten Speicheradressen geprüft, ob es sich um zwei Buddies handelt. Im positiven Fall wird eine „1“, im Negativen eine „0“ an die Funktion zurückgegeben.

3.2.4 Initialisierung der Arrays

Die Funktion dient der initialen Erstellung aller relevanter Variablen und Arrays. Ebenso ist es mit dieser Funktion möglich, ohne Neustart des Skripts die bisherigen Anfragen zurückzusetzen und den Initialzustand wiederherzustellen. Umgesetzt wird dies durch das Löschen der genutzten Variablen und Arrays. Darauffolgend werden die zum Start benötigten Arrays mit einem Initialzustand wieder erzeugt.

3.2.5 Ausgabe

Die Darstellung teilt sich wie in Abbildung 5 zu sehen in fünf Bereiche ein. Am Anfang der Ausgabe wird ein variabler Infotext angezeigt. Dieser setzt sich aus den Ergebnissen der ausgeführten Funktionen zusammen. Sofern die Option $-l$ beim Aufruf der Ausgabefunktion gesetzt wurde, werden die Ergebnisse zusätzlich zur Bildschirmausgabe in eine Logdatei umgeleitet und angehängt. Im zweiten Teil wird das *Free Buddy Array* mit den entsprechenden freien Blöcken in der jeweiligen Größe angezeigt. Die Zahl „-1“ zeigt, sollte sie an erster Stelle stehen, die Nichtverfügbarkeit der entsprechenden Speichergrößen oder das Ende einer *Linked List* an. Im dritten Abschnitt wird die Diagrammfunktion aufgerufen, welche die belegten und freien Bereiche sowie die *Splitpoint*s visualisiert. Direkt unter dieser Darstellung wird der zusammengezählte freie Speicher ausgegeben, welcher jedoch nicht der zu allozierenden Größe eines Blocks entsprechen muss, was durch die externe Fragmentierung des Gesamtspeichers zustande kommt. Als Letztes werden die möglichen Programm-Optionen angezeigt und die Ausgabefunktion beendet. Der Simulationsdurchlauf erwartet nun wieder neue Eingaben.

```
Info   Init: Total Memory 1024 Byte - Date 2020-06-18 09:08:07
_____________________________
Size    | Free Startaddresses (-1 = no free Block)
64      | -1
128     | -1
256     | -1
512     | -1
1024    | 0     | -1
_____________________________
Chart (#-used Space // I-Splitpoint)

[                                                                ]
Actual summerized free Space: 1024


Do you want to allocate (a), free (f) or use a sample (s)
Other Options: Exit (e), return allocated List (l), reset actual status (r)
```

Abbildung 5 - Programmauszug Darstellungsfunktion

3.2.6 Diagramm

Das Balkendiagramm wurde mit einer statischen Größe definiert und ist so für Darstellungen bis 2048 Byte Speichergröße bei einer minimalen Blockgröße von 64 Byte geeignet. Dies wird durch die natürlichen Grenzen der Shell und der Forderung nach einem harmonischen Gesamtbild in der Ausgabe bedingt.

Durch drei FOR-Schleifen werden die Inhalte des Arrays proportional zur Diagrammgröße gefüllt. Im ersten Durchlauf werden die freien Bereiche gesetzt. In der zweiten Schleife überschreiben die errechneten besetzten Bereiche die entsprechenden Arrayeinträge. Danach werden die *Splitpoints* ausgelesen und im Array gespeichert. Am Ende der Funktion werden die gespeicherten Werte auf den Bildschirm ausgegeben.

4 Ergebnis

Durch die erarbeitete Lösung wird die Aufgabe der Simulation von Allokation und Deallokation von variabel konfigurierbarem Speicher im Konzept der Buddy-Speicherverwaltung als Bash-Skript vollständig umgesetzt. Hierbei wurden stellvertretend die einzelnen *Memory Array* Elemente verwendet, um Bytes zu repräsentieren. Darüber hinaus wurde eine Visualisierung durch ein Balkendiagramm implementiert, welches den komplexen Sachverhalt in einem High Level Ansatz darstellt.

Die Schwierigkeit der Implementierung liegt vor allem in der Speicherung und Auswahl der tatsächlich benötigten Daten mit einer passenden und effizienten Ablagestruktur, um möglichst einfach auf die abgelegten Werte zugreifen zu können. Das umgesetzte Skript ist eine der speichereffizientesten Lösungen, da der Speicher selbst mit für die Verwaltung verwendet wird und keine weiteren Listen oder umständlichen Implementierungen für multidimensionale Arrays umgesetzt werden mussten.

Das Ergebnis stellt eine von vielen Umsetzungsstrategien dar. Die Algorithmen können durch bestimmte Strategien oder durch die Implementierung in anderen Sprachen noch effizienter gestaltet werden. Zur Erhaltung der Verständlichkeit wurden jedoch einige Schritte wie beispielsweise die Buddyprüfung kleinschrittig abgebildet.

Das Buddy Speicherkonzept, so einfach es im *High Level Design* auch wirkt, ist ein faszinierendes, komplexes und zugleich sehr effektiv umzusetzendes Speicherverwaltungs-system, welches durch Hinzunahme bestimmter Optimierungen bis heute in Systemen zum Einsatz kommt und so für eine effiziente Nutzung des vorhandenen Speichers sorgt.

5 Anhang

5.1 Verwendete Software

Für die Umsetzung des Projektes wurde folgende Software / Umgebung verwendet:

Linux Ubuntu mit Kernel Version 4.15.0-101-generic
- Betriebssystem in einer Virtuellen Maschine (VMware Workstation)

GNU bash, version 4.3.48
- Benutzerschnittstelle unter Unix

sublime text 3
- Texteditor

git version 2.7.4
- Versionsverwaltungssystem

Luciddchart (https://app.lucidchart.com/)
- Diagrammzeichnungstool

5.2 Literaturverzeichnis

Knuth, Donald Ervin (1997). „The art of computer programming - 1. Fundamental algorithms“

Glatz, Eduard (2015). „Betriebssysteme : Grundlagen, Konzepte, Systemprogrammierung“

Tannenbaum, Andrew S. (2009). „Moderne Betriebssysteme“

Ditchen Patrick (2008). „Shell-Skript-Programmierung“