X.systems.press

X.systems.press ist eine praxisorientierte Reihe zur Entwicklung und Administration von Betriebssystemen, Netzwerken und Datenbanken.

Markus Zahn

Unix-Netzwerkprogrammierung mit Threads, Sockets und SSL

Mit 44 Abbildungen und 19 Tabellen

Springer

Markus Zahn
unp@bit-oase.de
http://unp.bit-oase.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1611-8618
ISBN-10 3-540-00299-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-00299-4 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: Druckfertige Daten des Autors
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3100 YL – 5 4 3 2 1 0

Vorwort

Vernetzte Rechnersysteme und insbesondere das weltumspannende *Internet* haben unsere Welt verändert. Mit Hilfe der dabei entstandenen Technologien ist es heute nicht nur möglich, sondern sogar äußerst einfach, mit dem eigenen PC selbst Teil dieses riesigen Computernetzwerks zu werden. Natürlich ist allein ein Verbund vernetzter Rechnersysteme für den Normalverbraucher noch nicht sonderlich interessant. Erst die Fülle von Anwendungen, von *Online-Enzyklopädien* über *Online-Banking* und *Online-Shopping* bis hin zu *File-Sharing* und *Online-Spielen,* die seit den Anfängen des Internets entstanden sind, gestaltet dieses Netz anziehend für seine Nutzer.

Die Anziehungskraft vernetzter Rechnersysteme steht und fällt also mit der Attraktivität und Zuverlässigkeit der darüber verfügbaren Anwendungen. Das vorliegende Buch beschäftigt sich deshalb mit der Programmierung vernetzter Computersysteme, genauer gesagt mit der Entwicklung netzwerkfähiger Client-/Server-Programme für Unix-Systeme (oder Unix-ähnliche Computersysteme). Es hat zum Ziel, dem Leser einen fundierten Einstieg in die Welt der Unix-Netzwerkprogrammierung zu vermitteln, klammert aber auch fortgeschrittene Themen nicht aus. Die notwendigen Grundlagen der Unix-Systemprogrammierung werden demnach ebenso berücksichtigt wie die Absicherung des Datenverkehrs mittels *SSL (Secure Socket Layer).* Zahlreiche Programmbeispiele mit typischen Implementierungsmustern stellen dem Leser darüber hinaus eine solide Codebasis für die Entwicklung zuverlässiger, leistungsfähiger und sicherer Netzwerkprogramme zur Verfügung.

Die Einschränkung auf Unix und Unix-ähnliche Systeme geht auf die gemeinsame Entwicklungsgeschichte des Unix-Betriebssystems und des Internets zurück. So fand z. B. in den 70'er Jahren die Implementierung von TCP/IP und der zugehörigen Socket-API zunächst auf Unix-Systemen statt. Unix-Systeme bilden aufgrund ihrer hohen Betriebsstabilität sowie ihrer seit langem etablierten Multiuser-, Multiprozeß- und Multithreading-Fähigkeiten auch heute die Plattform für die wichtigsten Netzwerkdienste im Internet. Selbst auf der Seite der Arbeitsplatzsysteme gewinnt momentan mit Linux wieder

ein Unix-ähnliches Betriebssystem mehr und mehr an Bedeutung. Nachdem andere Betriebssysteme ebenfalls die in diesem Buch beschriebene Socket-API für die Netzwerkprogrammierung adaptiert haben und auch das vorgestellte OpenSSL auf vielen Systemen verfügbar ist, sind die hier diskutierten Konzepte der Netzwerkprogrammierung mitsamt der Beispiele meist ohne größere Probleme auf nicht Unix-basierte Betriebssysteme übertragbar.

Sowohl durch meine eigenen praktischen Erfahrungen als auch durch meine Tätigkeit als Dozent mußte ich lernen, daß es für die Entwicklung effizienter und stabiler Netzwerkanwendungen keinesfalls ausreichend ist, allein die Grundlagen der klassischen Netzwerkprogrammierung zu beherrschen. Zum einen entstammen viele Problemstellungen weniger der Netzwerkprogrammierung als vielmehr der Unix-Systemprogrammierung. Auch typische Fehler beruhen oftmals auf grundlegenden Mißverständnissen (oder Informationsdefiziten) aus dem Bereich der Systemprogrammierung. Zum anderen steigt aber auch durch die wachsende Nutzung des Internets für private und geschäftliche Transaktionen wie Online-Banking und Online-Shopping der Bedarf an einer gesicherten Übertragung der transportierten Daten.

Insofern gliedert sich das Buch in drei Teile: der erste Teil widmet sich der Unix-Systemprogrammierung, der zweite Teil beschäftigt sich mit der klassischen Netzwerkprogrammierung und der dritte Teil beleuchtet die Absicherung des Datenverkehrs.

Nach einer allgemeinen Einführung in Kapitel 1 bereitet Kapitel 2, *Programmieren mit Unix-Prozessen,* die wichtigsten Unix-Grundlagen auf. Das Kapitel konzentriert sich ausschließlich auf die Ausschnitte der Systemprogrammierung, auf die wir später im Rahmen der Netzwerkprogrammierung zurückgreifen werden. Im Wesentlichen sind dies die Themen Ein-/Ausgabe, Signalbehandlung und Nebenläufigkeit auf Prozeßebene. Kapitel 3, *Programmieren mit POSIX-Threads,* zeigt, wie nebenläufige Handlungen auch innerhalb eines Prozesses implementiert werden können.

Die Kapitel 4, *Grundlagen der Socket-Programmierung,* und 5, *Netzwerkprogrammierung in der Praxis,* beschäftigen sich dann mit der Netzwerkprogrammierung auf Basis der Socket-API und besprechen fünf verschiedene Implementierungsmuster für typische Server-Programme.

Die letzten beiden Kapitel widmen sich schließlich der Absicherung des Datenverkehrs über das SSL-Protokoll. Nachdem in Kapitel 6, *Netzwerkprogrammierung mit SSL,* sowohl die SSL-Grundlagen als auch die Basisfunktionalität der freien SSL-Implementierung OpenSSL erläutert wurden, widmet sich Kapitel 7, *Client-/Server-Programmierung mit OpenSSL,* v. a. dem sachgemäßen, sicheren Umgang mit SSL-Zertifikaten und damit der Entwicklung sicherer SSL-fähiger Anwendungen.

Bei der Erstellung des Manuskripts sind mir zahlreiche Helfer mit ihren kritischen Fragen, inhaltlichen Anregungen und durch ihr fleißiges Korrekturlesen zur Seite gestanden. Mein Dank gilt den Freunden und Kollegen Dr. Lars

Freund, Thomas Morper, Hella Seebach, Dr. Michael Westerburg und ganz besonders Dr. Harald Görl für zahlreiche fruchtbare Diskussionen und viele hilfreiche Anregungen. Dem Springer-Verlag danke ich in Person von Frau Jutta Maria Fleschutz und Herrn Frank Schmidt für die angenehme, problemlose und notwendigerweise auch geduldige Zusammenarbeit.

Sämtliche Beispielprogramme aus dem vorliegenden Buch stehen unter der Adresse http://unp.bit-oase.de/ zum Download bereit. Konstruktive Kritik und Hinweise auf Fehler, die sich trotz sorgsamer Ausarbeitung des Manuskripts eingeschlichen haben könnten, nehme ich jederzeit gerne entgegen.

Augsburg, im Juni 2006

Dr. Markus Zahn
unp@bit-oase.de

Inhaltsverzeichnis

Beispielprogramme

1

Einführung

Das *Internet* ist in seiner heutigen Form ein weltumspannendes Rechnernetz, das sich selbst aus einer Menge einzelner, voneinander unabhängiger Netzwerke zusammensetzt. Der Begriff Internet ist dabei als Kurzschreibweise für *Interconnected Networks,* also miteinander verbundene Netzwerke, entstanden. Angefangen bei einzelnen Rechnersystemen, die zu einem *Local Area Network (LAN)* zusammengeschlossen sind, werden mehrere LANs über größere Distanzen hinweg zu einem *Wide Area Network (WAN)* verknüpft. Der verschachtelte Verbund vieler derartiger WANs ergibt schließlich das Internet.

Natürlich hatte zu den Anfangszeiten des Internets niemand die kühne Vision, ein Rechnernetz bestehend aus knapp 400 Millionen[1] miteinander vernetzter Computersysteme zu initiieren. Vielmehr sollte gegen Ende der 60'er Jahre im Auftrag der *Advanced Research Projects Agency (ARPA),* die damals für das US-amerikanische Verteidigungsministerium Forschungsprojekte förderte, ein Netzwerk entstehen, über das die damals knappen Rechenkapazitäten der Hochschulen durch den Austausch von Daten besser ausgenutzt werden sollten. Das resultierende Netzwerk hieß zu dieser Zeit auch noch nicht Internet, sondern *ARPANET,* und wurde Ende 1969 von der *University of California, Los Angeles,* der *University of California, Santa Barbara,* der *University of Utah* und dem *Stanford Research Institute* in Betrieb genommen.[2] Das Netz verband zunächst genau vier Rechner der vier beteiligten Universitäten.

Um eine möglichst hohe Ausfallsicherheit zu erreichen, ersannen die beteiligten Forscher ein *paketvermitteltes Netzwerk.* In paketvermittelten Netzen werden die zu übertragenden Daten vom Absender in einzelne Pakete zerlegt und vom Empfänger nach dem Eintreffen wieder zusammengesetzt. In einem komplexen Netzwerk können die einzelnen Pakete bei der Übertragung

[1] Stand: Januar 2006, siehe http://www.isc.org/ds/

[2] Der erste Datenaustausch soll laut [ZEI01] am 29. Oktober 1969 stattgefunden haben, um die Buchstabenkombination *LOG* zu übermitteln. Von den gleichzeitig telefonierenden Technikern werden die Worte „Hast du das L?" – „Ja!" – „Hast du das O?" – „Ja!" – „Hast du das G?" überliefert, dann sei der Rechner abgestürzt.

durchaus unterschiedliche Wege gehen. Absender und Empfänger sind also, im Gegensatz zu *leitungsvermittelten Netzwerken* mit fester Verbindung, lediglich lose miteinander verbunden.

Weitere Forschungsarbeiten führten bis 1974 zur Entwicklung von TCP/IP, einer Familie von Netzwerkprotokollen, die bis heute die Basis für den Datenaustausch über das Internet bilden. Um die Akzeptanz von TCP/IP zu forcieren, wurde die *University of California at Berkeley* damit beauftragt, TCP/IP in *Berkeley Unix* zu integrieren. Mit der zunehmenden Verbreitung von TCP/IP wuchs gleichzeitig die Anzahl der Rechner im ARPANET rapide an. Der große Verbund von Netzen, den das ARPANET nun langsam aber sicher darstellte, wurde schließlich als *das Internet* bekannt. TCP/IP wird seitdem auch als Internet-Protokoll-Familie bezeichnet.

1.1 TCP/IP-Grundlagen

Um die komplexen Zusammenhänge, die in einem Rechnernetz bestehen, besser strukturieren und beschreiben zu können, wurden sogenannte Referenzmodelle eingeführt. Das bekannteste dieser Modelle ist das von der *International Organization for Standardization (ISO)* standardisierte *OSI-Referenzmodell (Open Systems Interconnection Reference Model)*. Nachdem die Internet-Protokoll-Familie bereits vor dem OSI-Referenzmodell entwickelt wurde, liegt der TCP/IP-Architektur ein anderes, vereinfachtes Referenzmodell zugrunde. Die Erfahrungen mit dem TCP/IP-Referenzmodell sind dann bei der Ausarbeitung des OSI-Referenzmodells eingeflossen.

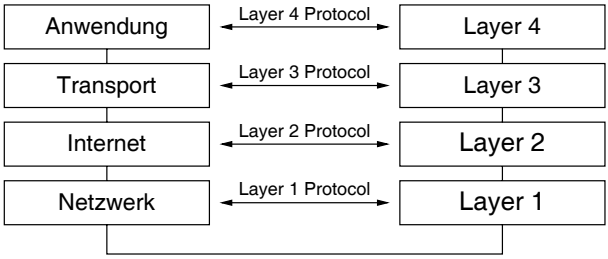

Abb. 1.1. Kommunikation im TCP/IP-Referenzmodell

Das TCP/IP-Referenzmodell unterteilt die einzelnen Aufgabenstellungen der Netzwerkkommunikation, angefangen bei der physikalischen Übertragung der Datensignale über die Vermittlung von Datenpaketen bis hin zu speziellen, anwendungsspezifischen Aufgaben, in einen Stapel von vier Schichten. Die einzelnen Schichten des Modells sind in Abb. 1.1 zu sehen. Jede der Schichten im Referenzmodell definiert bestimmte Funktionen, die in Form von Diensten und Protokollen implementiert werden:

- Jede Schicht kann die Dienste der darunterliegenden Schicht nutzen, ohne dabei konkrete Kenntnisse darüber besitzen zu müssen, wie die in Anspruch genommenen Dienstleistungen genau erbracht werden bzw. implementiert sind. Sämtliche Daten, die von einer Anwendung über das Netzwerk verschickt werden, durchlaufen von der Anwendungs- bis zur Netzwerkschicht alle Schichten des Modells (und auf dem empfangenden System wieder in der umgekehrten Reihenfolge).
- Die gleichnamigen Schichten zweier kommunizierender Systeme kooperieren miteinander jeweils über spezielle, schichtspezifische Protokolle. Dazu werden den von einer Anwendung verschickten Daten in jeder Schicht protokollspezifische Informationen hinzugefügt (und von der gleichnamigen Schicht auf dem empfangenden System wieder entnommen).

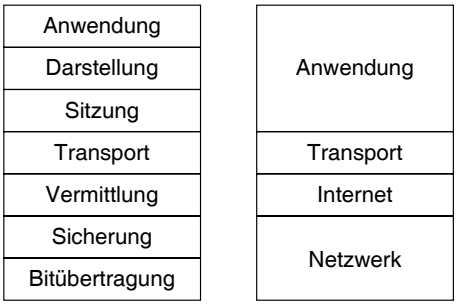

Abb. 1.2. OSI- und TCP/IP-Referenzmodell im Vergleich

Im direkten Vergleich zwischen OSI- und TCP/IP-Referenzmodell (vgl. dazu Abb. 1.2) zeigt sich die etwas feinere Untergliederung des OSI-Modells in einen Stapel von insgesamt sieben Schichten zusammen mit der korrespondierenden Aufteilung des TCP/IP-Modells.

1.1.1 Netzwerkschicht

Die beiden untersten Schichten des OSI-Referenzmodells repräsentieren grob gesprochen die Netzwerk-Hardware und die zugehörigen Gerätetreiber. Sie kapseln die eingesetzten Netzwerktechnologien wie z. B. *Ethernet, Token Ring* oder *FDDI* und liefern den übergeordneten Schichten eine davon unabhängige Schnittstelle zur Kommunikation. Diese beiden Schichten sind in der *Netzwerkschicht* des TCP/IP-Referenzmodells zusammengefaßt und spielen bei der Entwicklung netzwerkfähiger Client-/Server-Programme im Allgemeinen keine Rolle.

1.1.2 Internet-Schicht

Der Vermittlungsschicht des OSI-Modells entspricht im TCP/IP-Modell die *Internet-Schicht.* Diese Schicht kümmert sich in paketvermittelten Netzwerken wie dem Internet um die Weitervermittlung der einzelnen Datenpakete. Die Internet-Schicht etabliert damit eine Rechner-zu-Rechner-Verbindung, unabhängig von der zugrundeliegenden Netzwerkstruktur (bestehend aus zahlreichen LANs und WANs). Dadurch befreit die Schicht die übergeordneten Schichten von den Details der Datenübertragung.

Internet Protocol (IP)

Die Internet-Schicht wird bei TCP/IP vom *Internet Protocol (IP)* besetzt. Die wesentliche Aufgabe dieses in RFC 791 [IP81] spezifizierten Protokolls ist die Adressierung von Netzteilnehmern über *IP-Adressen* und die Datenübertragung von und zu anderen Teilnehmern. Die Daten werden dabei in einem vom Internet Protocol definierten Paketformat als sogenannte *Datagramme* an die tieferliegenden Schichten übergeben und über das Netzwerk übertragen. Bei Bedarf werden die Datagramme vom Internet Protocol fragmentiert, d. h. vor dem Versand in kleinere Fragmente zerlegt und nach dem Empfang wieder zusammengesetzt.

Beim Internet Protocol handelt es sich um ein *verbindungsloses Protokoll*, d. h. es existiert auf der IP-Ebene keine Ende-zu-Ende-Verbindung zwischen den kommunizierenden Anwendungen. Die einzelnen IP-Datagramme werden unabhängig von den jeweils anderen Datagrammen zugestellt, insbesondere kann sich dabei die Empfangsreihenfolge der Datagramme von der Reihenfolge beim Versand unterscheiden. Darüber hinaus wird der Empfang der einzelnen IP-Datagramme nicht bestätigt. Das Internet Protocol bietet also bezüglich der Datenübertragung keine Zuverlässigkeitsgarantie, weshalb das Protokoll auch oft als *nicht zuverlässig* oder sogar als *unzuverlässig* bezeichnet wird.[3] Das Internet Protocol kann demnach die tatsächliche Zustellung der verschickten Datagramme nicht garantieren, die einzelnen IP-Datagramme werden lediglich bestmöglich verschickt.

Internet Control Message Protocol (ICMP)

Das in RFC 792 [Pos81] spezifizierte *Internet Control Message Protocol (ICMP)* ist ebenfalls auf der Internet-Schicht angesiedelt. Das Protokoll ist

[3] Das Attribut *unzuverlässig* bedeutet in diesem Fall nicht, daß beim Internet Protocol ständig etwas schief läuft und man sich deshalb generell nicht darauf verlassen kann. Die Bezeichnung weist vielmehr darauf hin, daß das Protokoll selbst keine Mechanismen zur Erkennung und Behebung von Paketverlusten besitzt, so daß diese Aufgaben bei Bedarf von höheren Schichten übernommen werden müssen.

integraler Bestandteil jeder IP-Implementierung und transportiert Status-, Fehler- und Diagnoseinformationen für das Internet Protocol. Die ICMP-Pakete werden dazu als IP-Datagramme über das Netzwerk übertragen.

1.1.3 Transportschicht

Die *Transportschicht* ist die erste (niedrigste) Schicht, die den übergeordneten anwendungsorientierten Schichten, also den Schichten 5–7 des OSI-Modells bzw. Schicht 4 des TCP/IP-Modells, eine vollständige Ende-zu-Ende-Kommunikation zwischen zwei Anwendungen zur Verfügung stellt. Die Transportschicht leitet den Datenfluß von der Anwendung zur Internet-Schicht und umgekehrt. Die Adressierung der Anwendung erfolgt dabei über einen sogenannten *Port*, an den die Anwendung zuvor gebunden wurde.

User Datagram Protocol (UDP)

Genau wie das zugrundeliegende Internet Protocol ist auch das *User Datagram Protocol (UDP)* ein verbindungsloses Transportprotokoll ohne Zuverlässigkeitsgarantie.[4] Die Spezifikation des Protokolls findet sich in RFC 768 [Pos80]. Das User Datagram Protocol erweitert die bereits vom Internet Protocol erbrachten Leistungen lediglich um die Portnummern der sendenden und empfangenden Anwendung. Die von UDP transportierten Pakete werden deshalb in Anlehnung an IP auch *UDP-Datagramme* genannt. UDP hat demzufolge auch nur einen minimalen *Protokoll-Overhead,* was das Protokoll v. a. für Anwendungen, die nur wenige Daten übertragen müssen, interessanter als das nachfolgend beschriebene TCP macht.

Soll auf Basis von UDP eine zuverlässige, reihenfolgetreue Datenübertragung erfolgen, so muß sich eine der übergeordneten Schichten (sprich: die Anwendung selbst) um diese Aufgabe kümmern.

Transmission Control Protocol (TCP)

Das ursprünglich in RFC 793 [TCP81] spezifizierte *Transmission Control Protocol (TCP)* ist, im Gegensatz zu UDP, ein verbindungsorientiertes und zuverlässiges Transportprotokoll. Das Protokoll stellt je zwei Kommunikationspartnern eine virtuelle Ende-zu-Ende-Verbindung (im Sinne einer festen Punkt-zu-Punkt-Verbindung in einem leitungsvermittelten Netzwerk) zur Verfügung. Die Datenübertragung erfolgt bei TCP deshalb in drei Phasen: dem Verbindungsaufbau, der eigentlichen Datenübertragung und dem

[4] Bezüglich der (fehlenden) Zuverlässigkeitsgarantie gelten die selben Anmerkungen wie beim Internet Protocol (vgl. dazu Abschnitt 1.1.2).

abschließenden Verbindungsabbau. Die von TCP übermittelten Datenpakete werden *TCP-Segmente* genannt. Die Reihenfolge der über eine solche virtuelle Ende-zu-Ende-Verbindung verschickten TCP-Segmente bleibt im Rahmen der Übertragung erhalten. Nachdem das Protokoll zudem den Empfang eines TCP-Segments gegenüber dem Sender quittiert und der Sender beim Ausbleiben einer solchen Empfangsbestätigung die Übertragung des betreffenden Segments wiederholt, wird das Transmission Control Protocol als zuverlässiges Transportprotokoll bezeichnet.

TCP nimmt Daten von den übergeordneten Schichten als *Datenstrom* an und teilt diesen Datenstrom auf einzelne TCP-Segmente auf. Jedes dieser TCP-Segmente wird dann über die Internet-Schicht als IP-Datagramm verschickt. Umgekehrt werden die einer TCP-Verbindung zugeordneten IP-Datagramme bzw. TCP-Segmente auf der Empfängerseite wieder zum ursprünglichen Datenstrom zusammengesetzt. Nachdem das zugrundeliegende Internet Protocol bezüglich der Datenübertragung keine Zuverlässigkeitsgarantie übernimmt, muß sich das Transmission Control Protocol selbst um diese Aufgabe (wiederholtes Senden verlorengegangener Pakete, Einhalten der Reihenfolge) kümmern. Die TCP-Segmente erhalten dazu sogenannte *Sequenz-* und *Bestätigungsnummern,* mit deren Hilfe die Segmente quittiert, ggf. neu übertragen und in der korrekten Reihenfolge wieder zu einem Datenstrom zusammengesetzt werden können.

1.1.4 Anwendungsschicht

Die oberste Schicht des TCP/IP-Referenzmodells ist die *Anwendungsschicht.* Sie faßt die Schichten 5–7 des OSI-Modells in einer Schicht zusammen. Für diese Schicht ist inzwischen eine Fülle von Anwendungsprotokollen spezifiziert, über welche heute die gängigen Internetdienste miteinander kommunizieren. Stellvertretend seien an dieser Stelle die weithin bekannten Anwendungsprotokolle *Hypertext Transfer Protocol (HTTP)* für das *World Wide Web (WWW)* oder das *Simple Mail Transfer Protocol (SMTP)* zum Austausch *elektronischer Post (E-Mail)* genannt.

Die Anwendungsprotokolle bzw. die Anwendungen, die diese Anwendungsprotokolle implementieren, greifen ihrerseits über die *Socket-API* auf die Dienste der Transportschicht zu. Im weiteren Verlauf dieses Buchs kümmern wir uns nun im Wesentlichen um die Funktionalität der Socket-API und ihre Anwendung im Rahmen eigener Client-/Server-Programme.

1.2 Internet-Standards

Die sogenannten *RFCs (Request for Comments)*, von denen wir in diesem Kapitel bereits mehrere zitiert haben, spielten und spielen bei der Entstehung

und Weiterentwicklung des Internets eine gewichtige Rolle. Dabei handelt es sich um eine fortlaufend ergänzte Reihe von technischen und organisatorischen Dokumenten, über welche die Standards für das Internet festgelegt werden. Sie beschreiben die Dienste und Protokolle des Internets und legen Regeln und Grundsätze für dieses Netzwerk fest. Die Sammlung aller RFCs wird an zahlreichen Stellen im Internet publiziert, u. a. auf der Homepage des *RFC-Editors.*[5]

Neue RFC-Dokumente werden von einer Arbeitsgruppe bzw. dem RFC-Editor geprüft und durchlaufen dabei bestimmte Reifestufen. Ein einmal veröffentlichter RFC wird nie wieder verändert oder aktualisiert. Stattdessen wird er bei Bedarf durch einen neuen RFC ersetzt und erhält den Zusatz, daß er durch den neuen RFC abgelöst wurde. Der neue RFC enthält seinerseits einen Hinweis auf den RFC, den er abgelöst hat.

Neben ihrer verantwortungsvollen Aufgabe als eine Art Standardisierungsgremium beweisen RFC-Autoren mitunter auch sehr feinen Humor, bevorzugt am 1. April jeden Jahres. Stellvertretend für viele amüsante Exkursionen sei an dieser Stelle auf den lesenswerten und durchaus zu diesem Buch passenden RFC 1925 [Cal96], *The Twelve Networking Truths,* hingewiesen.

1.3 Unix-Standards

Die Entstehungsgeschichte von Unix reicht wie die Geschichte des Internets ins Jahr 1969 zurück. Ken Thompson begann damals bei den Bell Laboratories mit der Entwicklung des neuen Betriebssystems *UNICS,* einer in Assembler geschriebenen, abgespeckten Version des Betriebssystems *MULTICS.*[6] Der Name UNICS wandelte sich im weiteren Projektverlauf zu Unix.[7] Das Unix-System wurde dann Anfang der 70'er Jahre in der, im Rahmen der Unix-Entwicklung entstandenen, Programmiersprache C neu implementiert und wenig später zusammen mit einem C-Compiler kostenfrei an verschiedene Universitäten verteilt. In der Folgezeit entstanden viele variierende Systemlinien von Unix mit jeweils unterschiedlichen Kommandos, Kommandooptionen und Systemschnittstellen. So wurde z. B. von der *University of California at Berkeley* neben anderen Anpassungen auch TCP/IP in das dort entstandene *Berkeley Unix* integriert.

[5] http://www.rfc-editor.org/

[6] Übrigens: Der Name UNICS enthält gleich ein zweifaches Wortspiel: Zum einen dokumentiert *UNI* im Gegensatz zu *MULTI*, daß es sich bei UNICS um ein abgespecktes MULTICS handelt. Zum anderen wird UNICS in etwa so ausgesprochen wie das Wort *Eunuchs,* was UNICS als kastriertes MULTICS darstellt.

[7] Streng genommen steht die Schreibweise *UNIX* für Unix-Systeme, deren Implementierung auf den ursprünglichen Unix-Quellen der Bell Laboratories basiert, während durch die Schreibweise *Unix* zusätzlich auch Unix-ähnliche Systeme, wie z. B. Linux, mit eingeschlossen werden.

Die vielen verschiedenen Unix-Derivate mit ihren gegenseitigen Inkompatibilitäten führten schließlich zu den ersten Standardisierungsbemühungen und es entstanden die sogenannten *POSIX-Standards*.[8] Die aktuelle Version [SUS02], auf die sich die Ausführungen des vorliegenden Buchs beziehen, ist unter den drei Namen *IEEE Std 1003.1-2001, ISO/IEC 9945:2002* und *Single UNIX Specification, Version 3* bekannt. Wir werden im weiteren Verlauf den Namen IEEE Std 1003.1-2001 oder einfach nur kurz POSIX-Standard verwenden.

[8] Das Akronym *POSIX* steht für *Portable Operating System Interface,* ergänzt um die Unix-typische Endung *IX*.

2

Programmieren mit Unix-Prozessen

Prozesse bilden die Grundlage der Datenverarbeitung auf Unix-Systemen. Es ist also nicht weiter überraschend, daß vertiefte Programmierkenntnisse mit Prozessen die beste Basis für die Implementierung solider Netzwerkanwendungen sind. Aus diesem Grund vermittelt dieses Kapitel zunächst die notwendigen Grundkenntnisse der Programmierung von Unix-Prozessen, auf die im weiteren Verlauf dieses Buchs aufgebaut wird. In [Ste92, Her96] finden Sie bei Bedarf ausführlichere Informationen zur Unix-Systemprogrammierung. Die Grundlage für dieses Kapitel sowie für den weiteren Verlauf dieses Buchs bilden die in IEEE Std 1003.1-2001 [SUS02] vereinbarten Unix-Standards.

2.1 Unix-Prozesse

Unter einem *Prozeß* wird in einer Unix-Umgebung ein *in Bearbeitung befindliches Programm* bezeichnet. Wird also ein ausführbares Programm, z. B. ein zuvor mit einem C-Compiler übersetztes C-Programm, gestartet, so nennt man die gerade laufende Instanz dieses Programms einen Prozeß. Formal definiert sich ein Prozeß als eine Einheit, die aus

- einem (zeitlich invarianten) Programm,
- einem Satz von Daten, mit denen der Prozeß initialisiert wird, und
- einem (zeitlich varianten) Zustand besteht.

Im klassischen Kontext liegt jedem Prozeß ein sequentielles Programm zugrunde. Dies bedeutet, daß die einzelnen Anweisungen des Programms und damit die einzelnen Schritte des implementierten Algorithmus' immer in einer bestimmten Reihenfolge nacheinander durchlaufen werden.

In allen modernen Unix-Systemen ist diese Annahme in der Zwischenzeit überholt. Die bislang beschriebenen Prozesse werden dort als *schwergewichtige Prozesse* bezeichnet, welche sich dann ihrerseits aus einem oder mehreren *leichtgewichtigen Prozessen,* sogenannten Threads, zusammensetzen. Der Standard IEEE Std 1003.1-2001 definiert deshalb wie folgt:

Eine Prozeß ist eine Einheit bestehend aus

- einem Adreßraum,
- mit einem oder mehreren in diesem Adreßraum ablaufenden Threads und
- den für die Threads benötigten Systemressourcen.

In Kapitel 3 werden wir ausführlich auf die Programmierung mit Threads eingehen. Für den weiteren Verlauf dieses Kapitels können wir allerdings unter einem Prozeß meist problemlos die klassische, sequentielle Version verstehen.

2.1.1 Prozeßgruppen und Sessions

Jedem Unix-Prozeß wird bei seinem Start vom Betriebssystem eine eindeutige Prozeßnummer (Prozeß-ID, kurz: PID) zugewiesen. Neue Prozesse können ausschließlich von bereits vorhandenen Prozessen erzeugt werden. Der erzeugende Prozeß wird als Elternprozeß, der neue Prozeß als Kindprozeß bezeichnet. Beim Systemstart wird unter Unix ein ausgezeichneter Prozeß, der sogenannte `init`-Prozeß, gestartet. Diesem Prozeß wird *immer* die PID 1 zugewiesen. Der `init`-Prozeß bildet den Ursprung aller Prozesse, alle weiteren Prozesse stammen mehr oder weniger direkt von diesem Prozeß ab.

Die verschiedenen Prozesse werden in Unix zu *Prozeßgruppen* zusammengefaßt und jede Prozeßgruppe gehört wiederum zu einer *Session.* Auch Prozeßgruppen und Sessions bekommen vom Betriebssystem eindeutige Identifikationsnummern (Prozeßgruppen-ID und Session-ID bzw. PGID und SID) zugewiesen.

Eine neue Session wird z. B. dann gestartet, wenn sich ein Benutzer über ein Terminal erfolgreich am System anmeldet. Für den Benutzer wird bei der Anmeldung eine Login-Shell, etwa die Korn-Shell (ksh), gestartet. Für diese Shell wird gleichzeitig eine neue Prozeßgruppe erzeugt. Werden nun von der Shell aus weitere Programme gestartet, gehören alle diese Prozesse inklusive der Login-Shell standardmäßig zur gleichen Session. Der erste Prozeß in einer neuen Session, in unserem Beispiel die Korn-Shell, wird als *Anführer der Session (Session Leader)* bezeichnet.

Um das beschriebene Szenario zu veranschaulichen, melden wir uns an einem Unix-System an und geben auf der Kommandozeile die beiden folgenden, an sich nicht besonders sinnvollen, Befehlsfolgen ein:

```
$ sleep 60 | tail &
$ ps | cat | head
```

Die beiden Kommandos der ersten Zeile werden von der Shell im Hintergrund gestartet. Während diese beiden Prozesse noch laufen, führt die Shell die nächsten drei Prozesse im Vordergrund aus. Abbildung 2.1 zeigt den Zusammenhang zwischen Session, Prozeßgruppen und Prozessen.

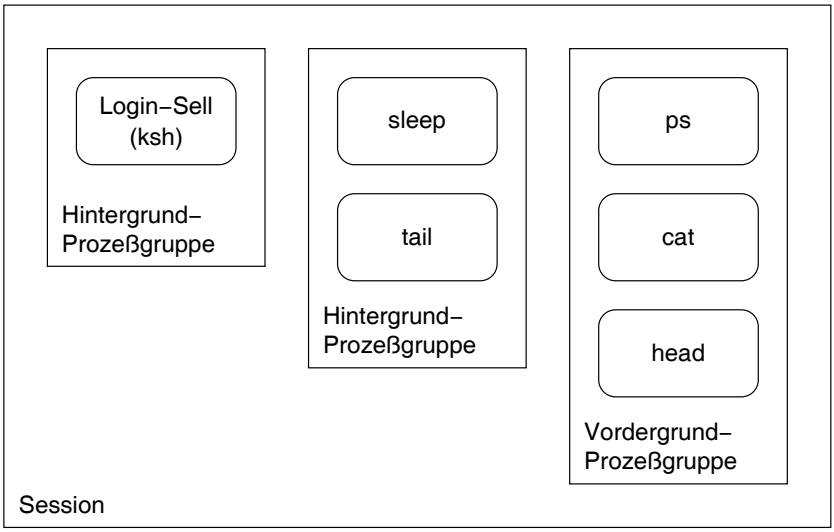

Abb. 2.1. Zusammenhang zwischen Session, Prozeßgruppen und Prozessen

Wie beschrieben gehören alle Prozesse zur gleichen Session. Neben der Login-Shell bilden die zuerst gestarteten Prozesse *sleep* und *tail* sowie die danach gestarteten Prozesse *ps, cat* und *head* zusammen je eine eigenständige Prozeßgruppe. Die Prozeßgruppe, die im Vordergrund ausgeführt wird – davon kann es pro Session maximal eine geben – wird als Vordergrund-Prozeßgruppe bezeichnet. Analog heißen alle anderen Prozeßgruppen der Session Hintergrund-Prozeßgruppen. Der nachfolgende Auszug aus der Prozeßtabelle zeigt die Details:

```
 PID  PPID  PGID   SID TTY   TPGID COMMAND
2701  2698  2701  2701 tty0   3271 -ksh
3269  2701  3269  2701 tty0   3271 sleep
3270  2701  3269  2701 tty0   3271 tail
3271  2701  3271  2701 tty0   3271 ps
3272  2701  3271  2701 tty0   3271 cat
3273  2701  3271  2701 tty0   3271 head
```

Die sechs Prozesse (PID 2701 bis 3273) bilden drei verschiedene Prozeßgruppen (PGID 2701, 3269 und 3271). Die Elternprozeß-ID (PPID) zeigt an, daß die Login-Shell (PID 2701) die anderen Prozesse (PID 3269 bis 3273) gestartet hat. Die Login-Shell selbst wurde wieder von einem anderen Prozeß initiiert.

Die Vordergrund-Prozeßgruppe erkennt man in der Prozeßtabelle an der Übereinstimmung zwischen der Prozeßgruppen-ID (PGID) und der Terminal-Prozeßgruppen-ID (TPGID). Die Prozesse *ps, cat* und *head* bilden im vorangehenden Beispiel die Vordergrund-Prozeßgruppe. Alle Prozesse gehören zur selben Session (SID 2701). Die Prozesse, bei denen die Prozeß-ID mit der Prozeßgruppen-ID übereinstimmt, werden als Anführer der Prozeßgruppe (process group leader) bezeichnet. Im obigen Beispiel sind das die Login-Shell *ksh* sowie die beiden Prozesse *sleep* und *ps.*

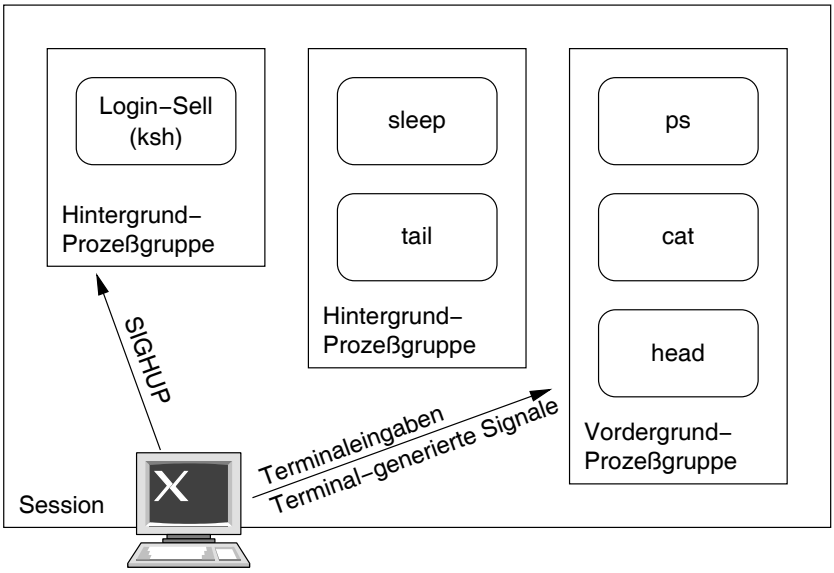

Abb. 2.2. Session und kontrollierendes Terminal

2.1.2 Kontrollierendes Terminal

Einer Session kann genau ein *kontrollierendes Terminal* zugeordnet sein. Dies ist z. B. immer dann der Fall, wenn die Session bei einem erfolgreichen Login erzeugt wurde. Über das kontrollierende Terminal können Daten an die Prozesse der Vordergrund-Prozeßgruppe geschickt werden. In unserem Beispiel bedeutet dies, daß alle Tastatureingaben am Login-Terminal an die Prozesse mit der PGID 3271 gesendet werden. Außerdem werden die über dieses Terminal generierten Signale (etwa `SIGSTOP` zum Stoppen oder `SIGTERM` zum Ab-

bruch eines Programms) an alle Prozesse der Vordergrund-Prozeßgruppe geschickt. In unserem Fall würden durch das Abbruchsignal die drei Kommandos *ps, cat* und *head* beendet. Danach existiert die Prozeßgruppe 3271 nicht mehr und die Prozeßgruppe der Login-Shell wird zur Vordergrund-Prozeßgruppe, an die dann bis auf weiteres alle Tastatureingaben gehen.

Der Anführer einer Session, der die Verbindung zum kontrollierenden Terminal herstellt, wird als *kontrollierender Prozeß (Controlling Process)* bezeichnet. Der kontrollierende Prozeß erhält das `SIGHUP`-Signal *(Hang Up)*, sobald das kontrollierende Terminal die Verbindung trennt. Abbildung 2.2 veranschaulicht diesen Zusammenhang zwischen einer Session und dem assoziierten kontrollierenden Terminal.

2.1.3 Verwaiste Prozesse und verwaiste Prozeßgruppen

Beendet sich ein Prozeß, von dem noch Kindprozesse aktiv sind, so verlieren diese Kindprozesse natürlich ihren Elternprozeß. In Analogie zum richtigen Leben werden diese Kindprozesse dann als *verwaiste Prozesse* bezeichnet. Die Elternprozeß-ID solcher Waisen wäre damit nicht mehr gültig, sie referenziert keinen aktiven Prozeß mehr. Der Standard sieht deshalb vor, daß die Elternprozeß-ID in diesem Fall automatisch auf einen (von der Unix-Implementierung vorgegebenen) Systemprozeß gesetzt wird. In der Praxis ist dies der `init`-Prozeß.

Auch Prozeßgruppen können unter Unix verwaisen. Dieser Fall tritt genau dann ein, wenn für alle Elternprozesse aller Mitglieder einer Prozeßgruppe folgendes zutrifft:

- Der Elternprozeß ist entweder selbst Mitglied dieser Prozeßgruppe oder
- der Elternprozeß gehört nicht zur Session dieser Prozeßgruppe.

Mit anderen Worten ist eine Prozeßgruppe also solange *nicht* verwaist, solange ein Prozeß aus der Gruppe einen Elternprozeß besitzt, der zu einer anderen Prozeßgruppe innerhalb der gleichen Session gehört.

Würde sich im Beispiel aus Abb. 2.1 die Login-Shell beenden, bevor sich die Prozesse der beiden anderen Prozeßgruppen beendet haben, so wären alle Prozesse dieser Gruppen verwaist. Ihre Elternprozeß-ID würde in diesem Fall jeweils automatisch auf die Prozeß-ID von `init` gesetzt. Der `init`-Prozeß wäre damit für alle der neue Elternprozeß. Da für die Login-Shell eine eigene Session erzeugt wurde, hätten nun alle Prozesse aus den beiden Prozeßgruppen mit `init` einen Elternprozeß, der zu einer anderen Session (und damit selbstverständlich auch zu einer anderen Prozeßgruppe) gehört. Die beiden Prozeßgruppen wären damit verwaist.

2.1.4 Prozeßumgebung

Intern bestehen Prozesse im wesentlichen aus ihren Code- und Datenbereichen. Abbildung 2.3 zeigt die verschiedenen Segmente eines Prozesses:

- In das *Text-Segment* wird der Maschinencode des Programms geladen,
- im *Data-Segment* werden die explizit initialisierten globalen Variablen gespeichert,
- das *BSS-Segment* enthält alle uninitialisierten globalen Variablen (welche dann beim Programmstart mit 0 initialisiert werden),
- im nach oben wachsenden *Heap-Segment* (der Halde) werden die dynamisch allozierten Speicherbereiche angelegt und
- das nach unten wachsende *Stack-Segment* (der Stapel), enthält die automatischen und temporären Variablen, die Rücksprungadressen bei Funktionsaufrufen, zwischengespeicherte Registerwerte und einiges mehr.

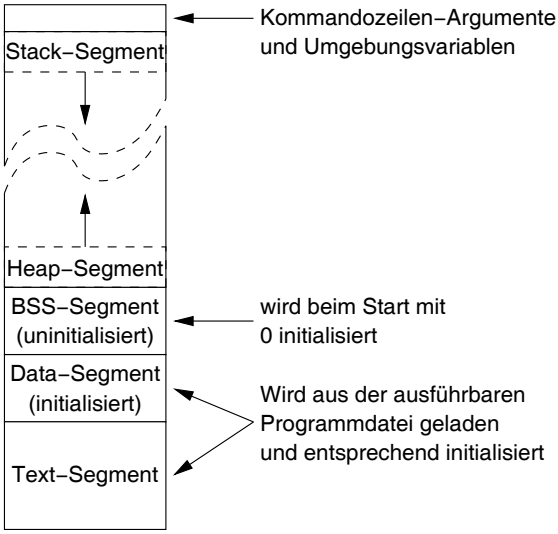

Abb. 2.3. Speicher-Layout von Prozessen

Der Systemkern hält in einer *Benutzerstruktur* weitere charakteristische Informationen für jeden Prozeß, u. a.:

- Registerinhalte die bei einem Systemaufruf dort abgelegt werden,
- Parameter und Ergebnisse des aktuellen Systemaufrufs,
- die Tabelle der Dateideskriptoren (siehe Abschnitt 2.2) sowie

- Abrechnungsdaten (verbrauchte CPU-Zeit im Benutzer und Systemmodus, Systembeschränkungen, wie z. B. maximale CPU-Zeit, die maximale Stackgröße, etc.)

Bei der *Umgebung* eines Prozesses, also bei den genannten Attributen der Benutzerstruktur sowie den Code- und Datenbereichen, handelt es sich um geschützte Ressourcen. D. h. das Betriebssystem schottet diese Bereiche vor dem lesenden und schreibenden Zugriff durch andere Prozesse ab.

Unter dem *Kontext* eines Prozesses versteht man die aktuellen Registerwerte des ausführenden Prozessors, dazu gehören insbesondere der Befehlszähler und der Zeiger auf den Stack.

Ein Prozeßwechsel auf einem Unix-System bedingt demzufolge immer einen Umgebungswechsel *und* einen Kontextwechsel. In Kapitel 3 kommen wir nochmals auf die Bedeutung von Umgebungs- und Kontextwechsel zurück.

2.1.5 Lebenszyklus

Sobald ein neuer Prozeß gestartet wird, übergibt der Systemkern die weitere Ausführung des Programms über eine Startup-Routine der Laufzeitumgebung an die Hauptfunktion des Programms. Der sogenannten `main()`-Funktion werden dabei die Kommandozeilen-Argumente übergeben:

```
int main( int argc, char *argv[] );
```

Der neue Prozeß hat nun – nachdem er seine Aufgaben hoffentlich erfolgreich bewältigen konnte – drei verschiedene Möglichkeiten, sich selbst mit Anstand zu beenden:

1. Er verläßt mit `return` die `main()`-Funktion,
2. er ruft die Funktion `exit()` auf oder
3. er beendet sich durch Aufruf der Funktion `_exit()`.

Verläßt ein Prozeß die `main()`-Funktion durch `return`, so liegt die Kontrolle wieder bei der Startup-Routine, die nun ihrerseits die `exit()`-Funktion anspringt. Abbildung 2.4 veranschaulicht diesen Lebenszyklus. Darüber hinaus kann ein Prozeß von außerhalb – und daher meist unfreiwillig – durch ein Signal beendet werden. Eine letzte Möglichkeit: Der Prozeß beendet sich selbst durch ein Signal, das er durch Aufruf von `abort()` selbst auslöst. Die letzten beiden Varianten sind in Abb. 2.4 nicht dargestellt.

Die Funktionen `exit()` und `_exit()` sind durch ANSI/ISO C bzw. IEEE Std 1003.1-2001 spezifiziert. Der Standard für ANSI/ISO C legt fest, daß

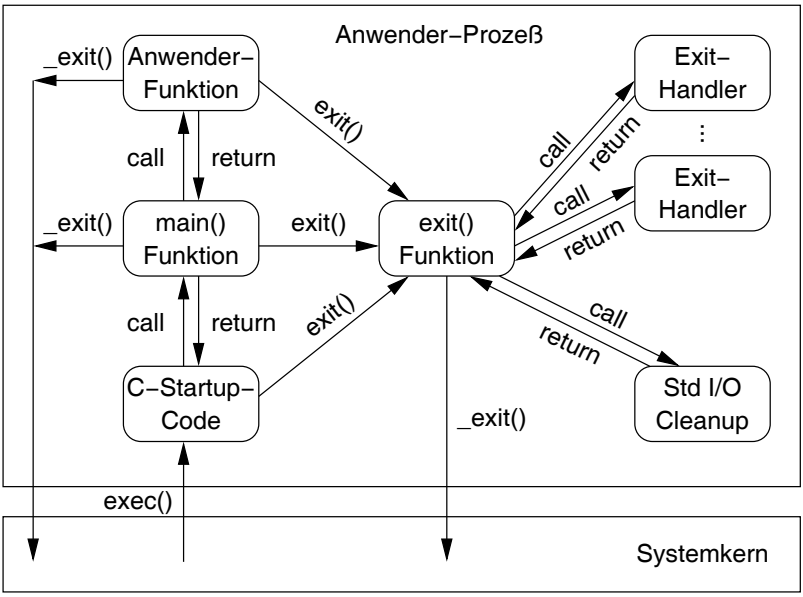

Abb. 2.4. Lebenszyklus von Prozessen

`exit()` die Puffer aller noch offenen Dateien schreibt, offene Datenströme mit `fclose()` abschließt, alle durch `tmpfile()` erzeugten temporären Dateien löscht, und die Kontrolle an das System zurück gibt. Ganz zu Beginn der `exit()`-Funktion müssen zudem noch alle mit `atexit()` hinterlegten *Exit-Handler* aufgerufen werden. Dies erfolgt in der umgekehrten Reihenfolge ihrer Registrierung mit `atexit()`.

```
#include <stdlib.h>

void exit( int status );
```

Nach ANSI/ISO C bedeutet der Statuswert Null bzw. `EXIT_SUCCESS` ein erfolgreiches Programmende. Hat `status` den Wert `EXIT_FAILURE`, wird dadurch ein nicht erfolgreiches Programmende angezeigt. Wie genau der aufrufenden Umgebung Erfolg oder Mißerfolg mitzuteilen ist und wie mit anderen Werten zu verfahren ist, überläßt ANSI/ISO C der jeweiligen Implementierung. Für Unix wird deshalb durch IEEE Std 1003.1-2001 festgelegt, daß der Status in Form der niederwertigen acht Bits des Statuswerts (d. h. der Wert `status & 0377`) zurück geliefert wird.

Genau genommen gibt die `exit()`-Funktion unter Unix den Rückgabewert und die Kontrolle nicht direkt an das System zurück, sondern ruft zu diesem

Zweck am Ende die Funktion `_exit()` auf, welche dann die Terminierung des Prozesses ordnungsgemäß abschließt.

```
#include <unistd.h>

void _exit( int status );
```

Die durch IEEE Std 1003.1-2001 festgelegte Funktion `_exit()` schließt u. a. alle noch offenen Dateideskriptoren (vgl. dazu Abschnitt 2.2) des aktuellen Prozesses und informiert gegebenenfalls den Elternprozeß über das „Ableben“ und den Rückgabe- bzw. Statuswert seines Kindes. Wir kommen in den Abschnitten 2.5 und 2.6 noch ausführlicher auf die Effekte und die z. T. damit verbundenen Probleme zu sprechen, die ein Prozeßende auf andere Prozesse wie Eltern- und Kindprozesse haben kann.

Handelt es sich bei dem terminierenden Prozeß um einen kontrollierenden Prozeß, also um den Anführer seiner Session, der die Verbindung zum kontrollierenden Terminal hergestellt hat, so wird an *alle* Prozesse in der Vordergrund-Prozeßgruppe (vgl. Abschnitt 2.1.2) des kontrollierenden Terminals das `SIGHUP`-Signal ausgeliefert. Außerdem wird die Assoziation zwischen der Session und dem kontrollierenden Terminal aufgelöst.

Verwaist durch den terminierenden Prozeß eine Prozeßgruppe und ist einer der Prozesse dieser Gruppe gestoppt, dann wird an alle Prozesse dieser Prozeßgruppe das `SIGHUP`-Signal, gefolgt vom `SIGCONT`-Signal ausgeliefert. Auch dazu später noch mehr.

```
#include <stdlib.h>

int atexit( void (*func)( void ) );
```

Die Funktion `atexit()` hinterlegt die benutzerdefinierte Funktion `func()`, die dann ausgeführt wird, wenn das Programm normal endet. Ein Rückgabewert ungleich Null signalisiert, daß der angegebene Exit-Handler nicht registriert werden konnte.

Exit-Handler dienen im Allgemeinen dazu, unmittelbar vor Programmende noch spezielle Aufräumarbeiten durchzuführen, die durch die `exit()`-Funktion nicht abgedeckt werden. Exit-Handler können sogar neue Exit-Handler hinterlegen. Es ist jedoch streng darauf zu achten, daß alle registrierten Funktionen auch zurückkehren.

Das nachfolgende Programm zeigt die Funktionalität und die Unterschiede der beiden Exit-Funktionen auf.

1–4 Zunächst binden wir die notwendigen Headerfiles in unser Programm ein. Das Headerfile `<stdlib.h>` enthält unter anderem die Prototypen für die in