

1 Programmieren unter Linux

In diesem ersten Kapitel werden Sie entdecken, was das Besondere an Linux ist, welchen Einfluss sein Vorbild Unix ausübt und wie der Gedanke von freier Software zu einer Massenbewegung geworden ist. Genauer werden wir dabei folgende Punkte untersuchen:

- Was ist eigentlich Unix?
- Ist Linux ein Unix? (Seite 4)
- Was ist der Unterschied zwischen kommerzieller Software und Open-Source-Software wie Linux? (Seite 5)
- Wie sieht die Programmentwicklung unter Unix aus? (Seite 14)

1.1 Das Unix-Betriebssystem

Für die meisten Leute ist heutzutage der erste Computer, mit dem sie in Kontakt kommen, ein PC, auf dem sich in aller Regel ein vorinstalliertes Windows-Betriebssystem von Microsoft befindet. Diese Geräte sind mittlerweile so weit verbreitet, dass sich manche schon gar nicht mehr vorstellen können, dass es auch noch andere Möglichkeiten gibt, einen Computer zu betreiben.

1.1.1 Die Unix-Familie

Im wissenschaftlich-technischen Bereich, wo immer schon größere Rechenleistung gefragt war, die der PC bis vor kurzem nicht erbringen konnte, setzt man seit langem so genannte *Workstations* ein. Auf diesen mit speziellen Prozessoren ausgestatteten Rechnern laufen Betriebssysteme, die es erlauben, dass mehrere Benutzer gleichzeitig darauf arbeiten und mehrere Programme nebeneinander laufen können. Die meisten dieser Betriebssysteme gehören dabei zur *Unix-Familie*.

Der Ausgangspunkt war eine Entwicklung beim US-Telefonkonzern AT&T in den siebziger Jahren. Die erste Version von Unix war noch in Assembler geschrieben; es wurde aber schon bald in die eigens dafür entworfene Hochsprache C portiert. In der Folgezeit übernahmen

Ursprung bei AT&T

verschiedene Hersteller das Unix-Konzept und entwickelten es für ihre eigene Hardware weiter [SALUS 1994]. Heute sind die Unix-Varianten zwar in ihren Grundkonzepten kompatibel, so dass Programme von einem so genannten Derivat zum anderen relativ leicht zu übertragen sind, sie unterscheiden sich jedoch sowohl in vielerlei tieferen Details als auch (wegen meist eigener Window-Manager) im Erscheinungsbild für den Benutzer.

Das Warenzeichen
UNIX

Genau genommen ist UNIX heute ein eingetragenes Warenzeichen der Open Group (einem Industriekonsortium, dem alle namhaften Firmen angehören, die in diesem Bereich tätig sind). Ein Betriebssystem darf sich demnach »UNIX« nennen, wenn es eine von dieser Organisation genau festgelegte Spezifikation erfüllt. Da deren Verifikation jedoch ein sehr teurer Prozess ist, gibt es nicht besonders viele Systeme, die dieses Kriterium erfüllen – nicht einmal alle kommerziellen. Im allgemeinen Sprachgebrauch hat sich daher eingebürgert, ein Betriebssystem »Unix« zu nennen, wenn es auf der ursprünglichen AT&T-Entwicklung fußt.

Es gibt neben einer Reihe kommerzieller Varianten wie Solaris (Sun Microsystems), AIX (IBM), HP-UX (Hewlett Packard) oder Irix (Silicon Graphics) auch einige frei erhältliche wie FreeBSD, NetBSD oder eben Linux.

1.1.2 Besondere Eigenschaften von Unix

Unix verfügt über einige Eigenschaften, die es auf PCs mit Microsoft-Betriebssystemen überhaupt nicht, nur in vereinfachter Form oder erst seit kurzem gibt. Der Umgang mit Unix erfordert daher sowohl vom Benutzer als auch vom Programmentwickler eine etwas andere Arbeitsweise als mit Windows. Da ich davon ausgehe, dass Sie bereits Linux auf Ihrem Rechner installiert haben, werden Sie vermutlich auch schon ein Gefühl dafür bekommen haben, was ich mit »andere Arbeitsweise« meine.

Multitasking

Zunächst arbeitet Unix mit so genanntem *Multitasking*. Kein Programm kann normalerweise alle Systemressourcen wie Speicherplatz, Rechenleistung oder Netzwerkdurchsatz für sich alleine in Anspruch nehmen. Es laufen stets eine ganze Reihe von Programmen im Hintergrund, denen das System abwechselnd, aber gleichmäßig Zugriff auf die Ressourcen erlaubt. Wenn Sie in einer Shell etwa den Befehl `ps aux` eingeben, werden Sie weit über zwanzig so genannte *Prozesse* sehen, die auf Ihrem Rechner momentan (fast) gleichzeitig ablaufen. Jeder Prozess läuft dabei in einem eigenen Bereich und beeinflusst die anderen kaum.

Multisuser

Ein weiteres wesentliches Merkmal von Unix ist die Fähigkeit, dass mehrere Benutzer gleichzeitig auf dem Computer arbeiten. Man be-

zeichnet dies auch als *Multiuser-Konzept*. (Natürlich darf man sich das nicht so vorstellen, dass vier Leute mit vier Tastaturen vor einem Bildschirm sitzen und jeder in irgendeiner Ecke etwas eingibt; mit »gleichzeitiger Benutzung« ist vielmehr der Zugriff über ein Rechnernetz gemeint.) Diese Fähigkeit hat eine Reihe von Konsequenzen:

- ❑ *Jede Datei gehört einem Benutzer*: Es muss sichergestellt sein, dass ein Benutzer nicht die Dateien eines anderen überschreiben oder löschen kann. (Aus Datenschutzgründen sollte sogar das Lesen verhindert werden können.) Diese Anforderungen erfüllt das Unix-Dateisystem, indem es zu jeder Datei den Namen des Benutzers speichert, der sie angelegt hat. Da zusätzlich Benutzer zu Gruppen zusammengefasst sind, kann man außerdem für jede Datei angeben, ob nur der Benutzer oder seine Gruppe oder jedermann sie lesen, überschreiben beziehungsweise ausführen darf.
- ❑ *Auch Prozesse sind an Benutzer gebunden*: Nicht nur die Dateien, sondern auch jeder einzelne Prozess (beispielsweise jedes laufende Programm) ist eindeutig einem Benutzer zugeordnet, so dass ein Saboteur nicht so einfach Prozesse anderer Benutzer beeinflussen oder beenden kann.
- ❑ *Konfigurationsinformationen sind stets benutzerspezifisch*: Anwendungen dürfen nicht, wie etwa unter Windows zum Teil üblich, ihre Konfigurationsinformationen an einem zentralen Platz abspeichern, denn jeder Benutzer könnte ja unterschiedliche Konfigurationen wünschen. Jeder Anwender verfügt daher über einen eigenen Datenbereich, das so genannte Homeverzeichnis. Applikationen hinterlegen ihre Einstellungen unter Unix somit im Allgemeinen in diesem Verzeichnis (oder sie legen sich dort ein Unterverzeichnis an).

Sie sollten diese Eigenschaften von Unix nicht nur im Hinterkopf behalten, weil wir zusammen gleich damit arbeiten wollen. Später wollen Sie ja auch selbst Programme unter Unix entwickeln, die natürlich auch diesen Randbedingungen gehorchen müssen. Beispielsweise sollten Sie stets vermeiden, dass Ihre Programme durch Leerlaufaktivitäten (etwa beim Warten auf Eingaben) mehr Ressourcen für sich in Anspruch nehmen, als eigentlich notwendig wären.

1.1.3 Die Werkzeug-Philosophie

Wie Sie vielleicht schon gemerkt haben, arbeitet man unter Unix traditionell sehr viel intensiver mit der Kommandozeile (der Shell) als unter DOS oder gar Windows. Daher gibt es unter Unix eine große Zahl klei-

*Tradition der kleinen
Hilfsprogramme*

ner Hilfsprogramme, die eine eng begrenzte Aufgabe effektiv erledigen. Dazu gehören Dateibetrachter wie *more*, Zeileneditoren wie *sed* und *awk*, Suchprogramme wie *find* und *grep* sowie Hilfesysteme wie *man* und *info*. (Diese Liste ließe sich noch einige Zeit fortsetzen ...) Jedes Werkzeug soll nur genau eine Aufgabe übernehmen, diese dann aber so gut wie möglich beherrschen.

Für den Neuling ist die Vielfalt dieser Utilities oft verwirrend, hat doch jedes seine eigenen Parameter, eventuell sogar einige Nebeneffekte. Geübte Unix-Anwender schwören indessen auf ihre Werkzeuge, da sie damit nach eigenen Angaben sehr viel effektiver arbeiten können. Ich nutze zwar auch hin und wieder einige dieser Werkzeuge, bin aber der Ansicht, dass Linux auch für alle verwendbar und nützlich sein sollte, die nicht die Zeit (oder die Lust) haben, sich ein Jahr mit der Shell und allen Tools vertraut zu machen. In diesem Buch werden Sie daher nur ein paar wenige, aber essenzielle Werkzeuge beschrieben finden; ansonsten will ich versuchen, Sie an verschiedene integrierte Lösungen mit grafischen Benutzeroberflächen heranzuführen, die es mittlerweile in respektabler Form auch unter Linux gibt. Wenn diese integrierten Umgebungen Teile ihrer Funktionalität aus dem Aufruf der »kleinen« Werkzeuge beziehen, soll uns das dann auch nicht weiter stören.

1.2 Linux

In welchem Verhältnis steht nun Linux zu Unix? Dazu muss man den Begriff »Linux« erst einmal etwas genauer erklären.

Geschichte von Linux

Ursprünglich bezeichnet Linux nur den Betriebssystemkern (engl. *kernel*), der 1991 vom mittlerweile berühmten Finnen Linus Torvalds und vielen anderen entwickelt wurde und seitdem fortlaufend verbessert und vielfach erweitert wird (zur Entstehungsgeschichte siehe [TORVALDS und DIAMOND 2002]). Dieser Kernel war am Anfang inspiriert von einem Lehrbetriebssystem namens Minix (das seinerseits die Funktionalität von Unix Version 7 bot), wurde aber auf vollkommen eigener Codebasis erstellt und verwendet weder Programmzeilen der AT&T-Entwicklung noch von Minix. Durch den intensiven Austausch der Ideen und Beiträge Hunderter Entwickler über das Internet wurde der Linux-Kernel zu dem eigenständigen, modernen und leistungsfähigen Unix, mit dem Sie heute arbeiten. Eine Verifizierung im Sinne des Unix-Warenzeichens von X/Open ist damit natürlich nicht verbunden, da die Linux-Gemeinde sicher nicht die Gebühren aufbringen könnte (und wollte!) und auch kein kommerzielles Interesse hinter Linux steht.

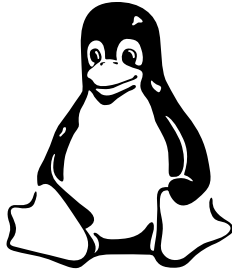


Abbildung 1.1
Das Maskottchen von Linux ist seit Version 2.0 der Pinguin Tux, das Lieblingstier von Linus Torvalds.

Wenn heute jemand davon spricht, er habe Linux auf seinem Computer installiert, so meint er damit im Allgemeinen wesentlich mehr. Ein Linux-System enthält neben dem Kernel noch eine große Zahl verschiedener Hilfs- und Anwendungsprogramme, die ebenfalls kostenlos erhältlich sind und die zumeist aus dem GNU-Projekt der Free Software Foundation stammen (siehe Abschnitt 1.3, Seite 5).

Wenn der Programmierer einige Rahmenbedingungen beachtet, ist es nicht besonders schwierig, Applikationen von Linux auf andere Unix-Versionen zu portieren. Das macht Linux zur idealen Entwicklungsplattform, da es nicht nur selbst sehr preisgünstig ist, sondern sich auch mit preisgünstiger Hardware begnügt – zumindest im Vergleich zu den gängigen Unix-Workstations. Das hat dazu geführt, dass die Weiter- beziehungsweise Neuentwicklung vieler Werkzeuge unmittelbar unter Linux stattfindet und sie erst anschließend auf andere Unix-Plattformen übertragen werden. Sie können also als Linux-Benutzer davon ausgehen, fast immer die aktuellsten Versionen der Applikationen und Tools zur Verfügung zu haben.

Portabilität ist natürlich stets ein relativer Begriff. Fast alle Beispielprogramme, die ich Ihnen im Laufe dieses Buches vorstellen werde, können Sie genauso gut unter Windows oder Mac OS übersetzen und zum Laufen bringen. Die Programmiersprachen C und C++ sind durch ihre Standardisierung hochportabel. Die Einschränkungen, die in den letzten Absätzen angeklungen sind, beziehen sich daher weitgehend auf direkte Systemzugriffe, also beispielsweise Netzwerk- oder Grafikprogrammierung.

Linux-Systeme heute

*Weiter- bzw.
Neuentwicklung oft
unter Linux*

1.3 Kommerzielle und freie Software

Anfangs war Software kein eigenständiges Produkt, sondern eine Beigabe der Hardwarehersteller, damit die Käufer die Computer auch nutzen konnten. Zu dieser Zeit war es auch noch selbstverständlich, dass die Software im Quellcode mitgeliefert wurde, damit die Anwender in der

Lage waren, sie ihren Bedürfnissen anzupassen. Unix konnte sicher nur deshalb so großen Zuspruch und so große Qualität erlangen, weil es aufgrund der kartellrechtlichen Bestimmungen von seinem Hersteller AT&T nicht verkauft werden durfte, sondern im Quelltext an Hochschulen und Forschungseinrichtungen zur Weiterentwicklung verbreitet wurde. Erst in den siebziger Jahren des letzten Jahrhunderts wurde Software zu einem selbstständigen Wirtschaftsgut, das nun in unveränderbarer Form ausgeliefert wurde.

1.3.1 Das GNU-Projekt

Damit hat sich aber die Freiheit des Anwenders ganz erheblich eingeschränkt. Er ist dadurch auf die Funktionalität, die der Hersteller vorgesehen hat, festgelegt und kann die Software nicht nach seinen eigenen Vorstellungen und Anforderungen abändern oder erweitern. Einer der Ersten, die davon überzeugt waren, dass hier existenzielle Freiheiten und Menschenrechte auf dem Spiel stehen, war und ist Richard Stallman, der Autor des Emacs-Editors (siehe Abschnitt 5.2, Seite 379). Er war einer der Initiatoren des GNU-Projektes, das am Massachusetts Institute of Technology (MIT) in Cambridge, Massachusetts, seinen Ausgangspunkt nahm und das sich das ehrgeizige Ziel gesetzt hatte, ein freies Unix zu schaffen. GNU ist dabei eine rekursiv definierte Abkürzung für »GNU's Not Unix«.

Abbildung 1.2

*Das Gnu ist das Symbol
für das gleichnamige
Projekt.*



Die Bestrebungen der GNU-Gemeinde wurden mittlerweile von Linux überholt; damit hat sich im Grunde auch ihr Ziel erfüllt, wenn auch etwas anders als anfangs gedacht. Großen Verdienst an der Nutzbarkeit und damit an der Verbreitung von Linux hat das GNU-Projekt aber trotzdem. In ihm wurden (und werden) viele wesentliche Werkzeuge erstellt, die kennzeichnend und notwendig für ein Unix sind (siehe Seite 3).

Dazu gehören:

GNU-Werkzeuge

- die Shell `bash`,
- die C- und C++-Compiler `gcc` beziehungsweise `g++`,
- der Debugger `gdb`,
- das Utility `GNUmake` zur Organisation der Erzeugung von Programmen aus Quelltext,
- der Editor `emacs`
- und viele andere mehr.

Somit kann man sagen, dass der Kernel von Linus Torvalds und die Tools von GNU die beiden Urzellen sind, aus dem sich das gesamte Linux-System entwickelt hat. Korrekterweise sollte man daher auch von "GNU/Linux" als dem Betriebssystem sprechen. (Der Vorschlag "Linux" konnte sich – zum Glück – nicht durchsetzen.)

Als Symbol für das GNU-Projekt wurde übrigens auch das Bild eines Gnus gewählt (Abbildung 1.2). Wie es im Kommentar auf der GNU-Homepage dazu heißt, hat dieses »den typischen Bart und intelligent aussehende gebogene Hörner. Er oder sie scheint angesichts der bereits vollbrachten Arbeit zufrieden zu lächeln, während der Blick jedoch noch in die Ferne schweift.«

Das Maskottchen

Unter der Adresse www.gnu.org können Sie die Homepage des GNU-Projekts finden. Entsprechend gibt es auch einen FTP-Server (unter [ftp.gnu.org](ftp://ftp.gnu.org)), der die aktuellsten Versionen aller GNU-Software bereitstellt. Aufgrund der Datenmengen und des hohen Nutzeraufkommens möchte ich Ihnen jedoch empfehlen, Downloads nicht direkt vom GNU-Server zu beziehen, sondern auf einen der Mirrors in Ihrer Nähe auszuweichen. Die meisten davon werden täglich aktualisiert und können Ihnen eine höhere Transferrate bieten.

GNU im Internet

1.3.2 Die GNU General Public License

Die Ideale von GNU wurden zu den Leitlinien der gesamten heutigen Open-Source-Bewegung: Software sollte »frei« sein, wobei »frei« im Sinne von »freier Rede« und nicht von »Freibier« zu verstehen ist. Bei freier Software hat der Benutzer das Recht, sie zu kopieren, zu verbreiten, zu analysieren und zu verbessern. Die unbedingte Voraussetzung dafür ist, dass der Anwender Zugang zum Quellcode der Software erhält.

Um die Ideale zu bewahren und zu fördern, gründete Stallman zusammen mit einigen anderen die *Free Software Foundation* (FSF, www.fsf.org). Die FSF arbeitete unter anderem eine Lizenzregelung aus, mit der sich die Freiheit an der Software erhalten lassen und gleichzeitig das Urheberrecht gesichert bleiben sollte. Diese ist heute in

Die GNU General Public License

ihrer Version 2 von 1991 sehr weit verbreitet und unter der Bezeichnung »GNU General Public License« (*GPL*) bekannt. Den genauen Text können Sie unter www.fsf.org/copyleft/gpl.html nachlesen.

Ziel der GPL

Die GPL erlaubt dem Anwender, Software, die darunter lizenziert ist, zu modifizieren, zu kopieren und zu verbreiten – vorausgesetzt, dass die dabei entstehende »abgeleitete Software« ebenfalls wieder unter der GPL veröffentlicht wird. Damit ist immer auch eine Freigabe des Quellcodes verbunden. Sie erhalten also kostenlos GPL-lizenzierten Code als Basis für ein eigenes Programm, müssen als Gegenleistung jedoch die eigenen Ergänzungen auch wieder jedermann zur Verfügung stellen und können damit andere nicht daran hindern, den gleichen Nutzen aus Ihrem Code zu ziehen, wie Sie das bei der vorherigen Version getan haben. Diese Vorschrift bezieht sich allerdings nur auf den Umgang mit dem Code in der Öffentlichkeit; was Sie damit intern, also beispielsweise innerhalb Ihrer Firma machen, steht Ihnen völlig frei. Sie müssen dazu auch niemanden um Erlaubnis fragen. Die Lizenzbestimmungen greifen erst dann, wenn Sie die Software wieder verbreiten, etwa als Produkt oder Teil eines Produktes.

Neue Software wieder unter GPL

Um es noch einmal deutlich zu sagen: Die Verpflichtung, dass abgeleitete Software wieder unter die GPL fällt, bedeutet nichts weniger, als dass jeder, der Teile des Codes eines unter GPL stehenden Programms in einer neuen Software verwenden will, seine Produkte ebenfalls wieder unter die GPL stellen, also frei verfügbar machen muss. Dieser »Virus-Effekt« hat sehr viel zur Verbreitung und zum Erfolg der freien Software beigetragen.

Gebühr für Distribution erlaubt

Wer freie Software verbreitet, darf für diese Dienstleistung auch eine Gebühr verlangen, gewährt seinen Kunden aber auch alle Rechte daran, die er selbst hat. Er muss zudem sicherstellen, dass der Benutzer auf Wunsch auch den Quellcode erhält, damit er diesen nach seinen Bedürfnissen abändern kann. Auch eine originalgetreue Kopie der GPL muss immer mitgeliefert werden, damit alle Empfänger über ihre Rechte im Klaren sind.

Beispiele für GPL-Software

Bekanntestes Beispiel für GPL-lizenzierte Software ist der Linux-Kernel. Aber auch die gesamte GNU-Software, die Benutzeroberflächen KDE und Gnome sowie »Open Office« (von Sun Microsystems) sind unter den Bedingungen der GPL veröffentlicht.

LGPL für Bibliotheken

Steht eine Bibliothek unter der GPL, so fallen alle Programme, die diese Bibliothek dazulinken – gleichgültig ob statisch oder dynamisch –, ebenfalls unter die Veröffentlichungspflicht, da sie als »abgeleitetes Werk« anzusehen sind. Aus diesem Grund hat man für Bibliotheken eine Abschwächung der GPL formuliert, die »Lesser GPL« (*LGPL*, früher auch als »Library GPL« bekannt). So lizenzierte Software (im All-

gemeinen sind dies Bibliotheken) darf ohne weiteres zu proprietären Anwendungen dazugelinkt werden, ohne dass diese wiederum automatisch unter die GPL fallen müssen. Bekanntes Beispiel ist die GNU-C-Bibliothek Glibc, die die Programmierschnittstelle zwischen dem Linux-Kernel und Anwendungsprogrammen bereitstellt. Ohne die LGPL wären »Closed Source«-Applikationen unter Linux überhaupt nicht möglich.

Ein Problem, das viele Leute mit der GPL haben, besteht darin, dass die Grenzen, was unter »abgeleiteter Software« zu verstehen ist, nicht immer leicht zu ziehen sind. Linux-Vater Linus Torvalds hat zu den Lizenzbestimmungen des Linux-Kernels daher noch den Satz hinzugefügt: »Dieses Copyright umfasst keine Anwenderprogramme, die die Kerneldienste durch normale Systemaufrufe verwenden – dies wird lediglich als normale Benutzung des Kernels angesehen und fällt daher nicht unter den Begriff der abgeleiteten Software«. Daher sind Anwenderprogramme, die nur die normalen Kernel-Systemaufrufe verwenden (was ohnehin bereits über die übliche Programmierung über die Glibc hinausgeht), auch als geschlossene Software ohne lizenzrechtliche Probleme realisierbar.

*Klärung der GPL für den
Linux-Kernel*

1.3.3 Andere Open-Source-Ansätze

Die GPL greift sehr weit in den Entstehungs- und Verbreitungsprozess von Software ein. Andere Gruppen und Institutionen, die ebenfalls die positiven Auswirkungen von offenen Quellen erkannten, wollten nicht immer so weit gehen und entwickelten daher eine Reihe anderer Lizenzmodelle (www.fsf.org/philosophy/license-list.html gibt einen Überblick). Allen gemeinsam ist die Freigabe des Quellcodes der Software, so dass jeder Interessierte sich von der Wirkungsweise der Programme überzeugen kann. Die Unterschiede liegen darin, was mit Änderungen passieren soll und zu welchem Zweck die Software genutzt werden darf. Beispielsweise bestimmt die »Qt Public License« QPL, unter der lange Zeit die Qt-Klassenbibliothek des norwegischen Softwarehauses Trolltech stand (siehe auch Seite 478), dass Änderungen nur als Patches und nicht als Gesamtpaket verbreitet werden dürfen und dass die Software nur so lange frei genutzt werden kann, solange das darauf aufbauende Produkt ebenfalls frei ist.

Einige Beispiele für Open-Source-Projekte, die unter anderen Bestimmungen als der GPL veröffentlicht werden, sind Zope von Digital Creations, PHP von Zend Technologies, Eclipse von IBM (siehe Seite 495) oder Mozilla von Netscape/AOL. Auch das oben erwähnte FreeBSD hat eine andere Lizenz als Linux.

*Beispiele für
Open-Source-Projekte*

1.3.4 Vorteile von Open Source

Die GPL trägt ganz wesentlich zum Erfolg und zur raschen und weiten Verbreitung von Linux und seinen Werkzeugen bei. Aber auch allgemein bietet Open-Source-Entwicklung im Gegensatz zu geschlossenem Code eine Reihe von nicht zu unterschätzenden Vorteilen, die wir im Folgenden näher untersuchen wollen (siehe auch [WIELAND 2001] und [WIELAND 2004]).

Einer der renommiertesten Vorreiter der Open-Source-Bewegung ist Eric Raymond, der mit seinen wegweisenden und engagierten Essays »The Cathedral and the Bazaar« [RAYMOND 1999a] und »The Magic Cauldron« [RAYMOND 1999b] viele wertvolle Argumente zur Diskussion beigetragen hat.

Das »Peer review«-Prinzip

Er sieht einen der wichtigsten Vorteile von Open Source im »Peer review«-Prinzip. Das bedeutet, dass Programme anhand ihres Codes von anderen Experten beurteilt werden können. Durch diese kritische Überprüfung können alle Arten von Fehlern relativ rasch aufgedeckt und beseitigt werden. Denn die Wahrscheinlichkeit, dass ein Fehler unbemerkt bleibt, ist bei mehreren Hundert Programmierern um Größenordnungen geringer als bei ein paar wenigen – sofern ein Code-Review bei der geschlossenen Entwicklung überhaupt stattfindet. Aus diesem Grund ist das Open-Source-Modell besonders für Infrastrukturkomponenten wie Betriebssystem-Kernels, Treiber oder Netzwerkdienste sinnvoll.

Sicherheitsüberprüfungen

Nicht nur für die Qualität ist das »Peer review« vorteilhaft, auch die Sicherheit profitiert davon. Denn eine wirklich ernsthafte Sicherheitsüberprüfung muss sich auch auf den Code beziehen; nur Algorithmen und Implementierungen, die von verschiedenen Seiten als sicher eingestuft werden, kann letztendlich vertraut werden. Mit diesen Argumenten warnen Organisationen wie das deutsche Bundesamt für Sicherheit in der Informationstechnik (BSI), die Europäische Kommission oder der amerikanische Geheimdienst NSA vor Betriebssystemen wie Microsoft Windows 2000 oder XP, deren Code nicht öffentlich ist.

Wiederverwendbarkeit für größere Komplexität

Wenn der Code einer Software offen liegt, kann jeder sich die darin verfolgte Problemlösung ansehen und daraus lernen. Eine bereits vorhandene und bewährte Lösung kann dann immer wieder verwendet werden. Der Einzelne muss sich nicht jedes Werkzeug selbst herstellen, sondern kann dieses einsetzen, um darauf eine komplexere Lösung zu erstellen. Dieses allgemeine Innovationsprinzip, das die technische Entwicklung der Menschheit charakterisiert, ist mit geschlossener Software kaum möglich.

Höhere Reife durch Unabhängigkeit

Bei einem Softwareprodukt, für das Lizenzgebühren verlangt werden, ist oft der Zeitpunkt der Markteinführung ausschlaggebend für den Erfolg. Leider ist der Entwicklungszyklus oft nicht synchron da-

mit, das heißt, das Produkt ist bei der Markteinführung eigentlich noch gar nicht reif dafür. Das Ergebnis ist die bekannte »Bananen-Software«, die grün ausgeliefert wird und erst beim Anwender reifen soll. Der ökonomische Schaden, der durch die massenhaften Probleme der Benutzer entsteht, ist kaum zu beziffern. Da die Open-Source-Entwickler ihre Arbeit nur aus Spaß am Programmieren (oder aus Prestige Gründen) verrichten, sehen sie in der Reife und Qualität ihres Produktes einen sehr wichtigen Aspekt. Und da sie keinem Produktmanager und keiner Geschäftsleitung verantwortlich sind, entscheiden sie über die Freigabe einer Version im Allgemeinen nach rein technischen Gesichtspunkten. Ein Beispiel dafür ist der Linux-Kernel, insbesondere die Version 2.4. Seine Freigabe kam über ein Jahr später als ursprünglich angekündigt, da die Entwickler ihn so lange zurückhielten, bis sie von seiner Stabilität überzeugt waren. Bei Mozilla konnte man die Probleme dieser Argumentation mit einer kommerziellen Betreuerfirma erleben. Obwohl die Entwickler von einer Freigabe einer neuen Version des Internetbrowsers abrieten, setzte sich die Muttergesellschaft AOL darüber hinweg und brachte »Netscape 6« auf den Markt. Wie erwartet war das Produkt reichlich instabil und an vielen Stellen schlicht »unfertig«. Das Vertrauen zum Netscape-Browser könnte damit nachhaltigen Schaden genommen haben.

1.3.5 Motivation für Open Source

Welche Gründe sprechen nun aus Sicht eines Programmierers beziehungsweise eines Softwareherstellers dafür, ein Programm als Open Source zu veröffentlichen und damit alle Welt in seine Karten, sprich: Quellen, sehen zu lassen?

Früher entstanden die meisten Open-Source-Projekte dadurch, dass ein Programmierer eine Idee verwirklichen wollte oder ein bestimmtes Tool benötigte, das er sich selber schreiben musste, dann aber mit anderen teilen wollte. Wenn eine Firma heute ein solches Projekt ins Leben ruft, geschieht dies kaum noch aus technischen, dafür meist aus Marketing-Überlegungen. Auf diese Weise kann sich das Unternehmen nicht nur als offen und innovativ hervorheben, sondern auch bestimmte Benutzergruppen an sich binden oder sich in neuen Märkten positionieren.

Marketing und PR

Außerdem können durch Open-Source-Programme eigene Standards mit viel größerer Breitenwirkung und in kürzeren Zeiträumen auf dem Markt etabliert werden, als dies mit kostenpflichtiger Software möglich wäre. Diese Strategie kann man beispielsweise bei Sun mit Java und bei IBM mit Internet- und XML-Software beobachten. Letztlich versichern zwar alle Hersteller, keine proprietären Standards zu

Setzen von Standards

wollen, sondern sich an offene Standards der internationalen Gremien (IETF, W3C etc.) zu halten. Da die Verabschiedung von Standards durch diese jedoch meist recht lange dauert, kann derjenige Hersteller seine eigenen Vorstellungen am Markt und bei der Standardisierung am besten durchsetzen, der über die größte Anwenderbasis verfügt.

*Geschäft mit darauf
aufbauenden
Produkten*

Damit ist nämlich auch gleich das Feld für die nächste Stufe bereitet: Der Hersteller findet bei genügender Akzeptanz des von ihm favorisierten Standards einen lohnenden Markt für kommerzielle Produkte vor, die darauf aufbauen und die Möglichkeiten der Open-Source-Programme ergänzen beziehungsweise erweitern. Das können größere kostspielige Softwarepakete, etwa für die Entwicklung oder Administration, genauso sein wie Hardware. Auch dafür ist IBM mit der WebSphere-Produktreihe, die auf Java- und Internetstandards sowie auf Eclipse setzt, oder der Linux-Portierung auf AS/400 ein gutes Beispiel.

Geschäft mit Services

Für die Anwender ist es mit der kostenlosen Software allein oft nicht getan. Je komplexer die Anwendung, desto mehr Bedarf besteht an Unterstützung bei der Anpassung an die Umgebung des Benutzers und an der Erarbeitung maßgeschneiderter Lösungen. Obwohl dies die klassische Einnahmequelle einer Open-Source-Firma bildet, ist diese Strategie nach wie vor lohnenswert.

*Anwender erweitern
Features*

Die Entwicklungsleistung kann dabei auch teilweise von einzelnen Anwendern kommen. Für manche Kunden kann das Programm so wichtig sein, dass sie eigene Ressourcen für dessen Pflege und Ausbau bereitstellen – Erweiterungen, die sie aufgrund des Lizenzmodells meist wieder an den Hersteller oder die gesamte Öffentlichkeit zurückmelden müssen. Auf diese Weise können also Entwicklungskosten vom ursprünglichen Produzenten auf seine Kunden verlagert werden – in vielen Fällen ein nicht unwesentliches Argument.

*Hersteller entbindet
sich von der laufenden
Pflege*

Aber nicht nur die Entwicklung kostet Geld; oft ist der laufende Unterhalt für die Pflege einer Software sogar noch teuer. Viele Firmen pflegen Softwareprodukte nicht deshalb, weil sie mit dem laufenden Lizenzverkauf noch etwas verdienen, sondern nur, um bestehende Kunden zu bedienen. In solchen Fällen bietet sich die Freigabe der Software als Open Source an, da man damit zumindest teilweise die Pflegeaufwände auf andere verteilen kann. Dazu ist allerdings auch der vorbereitende Aufbau einer entsprechenden Community nötig [WERRY und MOWBRAY 2000]. Ein prominentes Beispiel für diesen Schritt ist AOL/Netscape mit der Freigabe des Mozilla-Browsers – auch wenn dabei in der Folge noch ein paar andere Einflüsse hinzukamen.

Schließlich ist noch zu bedenken, dass auch in einem Unternehmen hinter einer Software nicht immer riesige Teams stehen, sondern oft nur zwei oder drei Mitarbeiter. Und von diesen hängt die Software dann auch ab. Selbst wenn die Programme ausreichend dokumentiert sind, lassen sie sich im Allgemeinen nicht ohne weiteres durch andere übernehmen. Falls also einer der Mitarbeiter (oder gar alle) die Firma verlassen oder eine neue Aufgabe übernehmen, ist die Gefahr groß, dass die Anwendung nicht gepflegt wird, hinter den sonstigen technischen Entwicklungen hinterherhinkt und langsam unbrauchbar wird. Durch die Veröffentlichung der Software als Open Source kann man dieses Problem vermeiden, denn man übergibt das Programm einer – hoffentlich – größeren Gemeinschaft, die sich um den Fortbestand (und somit um die Sicherung der ursprünglichen Investition in die Entwicklung) kümmern wird. So werden vielleicht nicht unmittelbar Kosten eingespart, dafür aber das Risiko der Unbrauchbarkeit auf eine breite Anwenderbasis außerhalb der eigenen Firma verteilt und die Abhängigkeit von einzelnen Entwicklern reduziert.

Unabhängigkeit von einzelnen Entwicklern

1.3.6 Fazit

Die Offenheit, die für Open Source sprichwörtlich ist, stellt für Hersteller wie Anwender eine enorme Chance dar. Hersteller können damit Marketing betreiben und ein positives Image in der Öffentlichkeit erzeugen, den Markt für weitere Produkte bereiten, Standards setzen oder die Pflege laufender Anwendungen an eine Community übergeben. Gerade der Aufbau und die Motivation einer solchen Community sind jedoch die Aspekte, die über Erfolg oder Misserfolg eines Open-Source-Projekts entscheiden. Auch muss man anders mit Produktzyklen und Zeitplanungen umgehen, als das bei proprietären Systemen üblich ist. Die Anwender erhalten, wenn sie bereit sind, sich darauf einzulassen, mit dem Quellcode die detaillierteste Dokumentation, die man sich wünschen kann. Abläufe lassen sich so bis auf die unterste Ebene debuggen, womit sich alle Arten von Fehlern besser aufspüren lassen. Je intensiver der Anwender mit den Quellen arbeitet, desto mehr verschwindet die Grenze zum Herstellerteam. Er erhält also eine neue Rolle, in der er aktiv auf das Produkt Einfluss nehmen kann. Die Frage der Produkthaftung wird dabei jedoch erheblich schwieriger, da einerseits das Entwicklerteam diese aus guten Gründen nicht übernimmt (und selbst nach deutscher Rechtslage nicht übernehmen muss), andererseits von Endkunden oder Management diese oft gefordert wird. Hier ist es am sinnvollsten, entweder diese Haftung in bestimmten Grenzen selbst zu übernehmen oder einen entsprechenden Vertrag mit einem Dienstleister abzuschließen. Auch wenn damit eine andere Denk- und Hand-

lungsweise verbunden ist, stellt Open Source ein höchst interessantes Modell dar. Der Erfolg von Linux, Apache, MySQL, PHP usw. haben gezeigt, dass es am Durchbruch keine Zweifel mehr geben kann, ja dass dieser sich gerade schon vollzieht. Wie so oft bei Trends dürfte auch hier gelten: Es hat der den meisten Erfolg, der sich frühzeitig darauf einstellt.

Die Kommunikation über das Internet hat es möglich gemacht, dass selbst Entwicklungsvorhaben, die ehemals als so groß eingestuft wurden, dass nur ein hierarchisch organisiertes und streng planendes Unternehmen sie bewältigen könnte – wie Betriebssysteme, Benutzeroberflächen oder Büro-Softwarepakete –, von einer Gruppe Freiwilliger, die über die ganze Welt verstreut ist, realisiert werden. Denn auch das ist das Ungewöhnliche an Software unter der GPL: Da sie frei ist, erhalten auch die Entwickler *keinen Lohn* dafür. Alle Beteiligten bringen ihre Arbeitsleistung unentgeltlich ein. Wenn nicht gerade eine Firma dahinter steht, die ihre eigene Motivation einbringt, geht es für die meisten um eine Art Tauschgeschäft, erhalten sie doch eine große Menge qualitativ hochwertiger Programme zum Nulltarif (einmal abgesehen von den Downloadkosten . . .). Im Gegenzug arbeiten sie dann an dem einen oder anderen Projekt mit, um sich erkenntlich zu zeigen und die Idee der freien Software weiter zu fördern.

Ihr Beitrag

Wenn Sie möchten, können auch Sie mit den Kenntnissen, die Sie sich nach Durcharbeiten dieses Buches erworben haben, Ihren Beitrag zur Verbesserung vorhandener oder zur Erstellung neuer Werkzeuge und Anwendungsprogramme leisten.

1.4 Programmentwicklung in Unix

Von DOS und Windows kennen Sie sicherlich den Unterschied zwischen Programmen und Skripten:

- ❑ Ausführbare Programme liegen in maschinenlesbarer Form vor und haben die Endung `.exe`.
- ❑ Skripten sind auch für uns lesbar und haben die Endung `.bat`. Sie bestehen aus einer Folge von Anweisungen, die der Rechner schrittweise abarbeiten soll.

Unter Unix gibt es diese Programmtypen zwar auch, sie sind jedoch nicht so leicht zu erkennen. Denn so etwas wie standardisierte Namens-erweiterungen existieren hier nicht. Im Unix-Dateisystem dient ein entsprechendes Attribut dazu, eine Datei als »ausführbar« zu kennzeichnen, wobei die Shell, aus der das Programm gestartet wird, selbst den Unterschied zwischen »echten« Programmen und Skripten erkennt. Für

den Benutzer ist also am Namen nicht ersichtlich, was er da gerade lostritt.

1.4.1 Wichtige Begriffe

Für das Verständnis des weiteren Textes brauchen wir noch ein paar Begriffe, damit Sie wissen, von was ich rede. Wenn Sie schon Programmiererfahrung haben, sind Ihnen diese sicher geläufig, so dass Sie diesen Abschnitt überspringen können.

Compiler

Bei den Programmiersprachen C, C++, Fortran, Pascal und so weiter handelt es sich um Hochsprachen, die durch ein spezielles Programm in Maschinencode übersetzt werden, aus dem schließlich eine ausführbare Datei entsteht. Ein solches Programm nennt man *Compiler*. Dieser ist also das zentrale und entscheidende Werkzeug, wenn Sie selbst Programme schreiben wollen. Bei der Umwandlung in maschinenlesbare Form, die man als *Übersetzen* oder Kompilieren bezeichnet, überprüft er Ihre Programmtexte auch daraufhin, ob diese im Einklang mit den Regeln der Programmiersprache (der *Syntax*) stehen. Dabei findet der Compiler nicht nur grobe Fehler, sondern weist Sie auch durch Warnungen auf mögliche Inkonsistenzen und Mehrdeutigkeiten hin. Es ist immer ein guter Stil, die Warnungen des Compilers ernst zu nehmen und die Programme nach Möglichkeit so lange zu bearbeiten, bis keine Warnungen mehr auftreten.

Dateiarten

Was Sie bei Ihren Programmen selbst eintippen, ist der so genannte *Quelltext* (*source code*). Quelltextdateien kennzeichnet man mit einer Endung, die auf die Programmiersprache hinweist. Bei C ist dies einfach `.c`; bei C++ verwendet man unter anderem `.cc`, `.cxx`, `.cpp` und `.C`.

Quelltext

Wie Sie vielleicht schon wissen, sind bei C und C++ die Schnittstellen von Unterprogrammen vom übrigen Code getrennt. Sie stehen oft in so genannten *Header-Dateien* (auch *Include-Dateien* genannt), die man üblicherweise mit der Endung `.h` kennzeichnet. (Nach dem neuen C++-Standard haben die Header-Dateien der C++-Standardbibliothek gar keine Endungen mehr.) Bei der objektorientierten Programmierung gilt als Faustregel: Für jede Klasse verwendet man eine Quell- und eine Header-Datei. (Wir werden später auf diese Begriffe noch häufiger und sehr viel genauer eingehen.)

Header-Dateien

Objektcode Aus diesem Quelltext, der aus einer oder mehreren Dateien bestehen kann, erzeugt der Compiler entweder gleich das ausführbare Programm oder eine Zwischenstufe, den Objektcode, aus dem er dann später das Programm aufbauen kann. Diese Objektdateien haben unter Microsoft-Betriebssystemen die Endung `.obj`, unter Unix allerdings schlicht `.o`.

Linker Wenn der Compiler mit dem Übersetzen der Quelltextdateien fertig ist, baut er das ausführbare Programm aus den Objektcodes zusammen und berücksichtigt auch die Aufrufe von Systemfunktionen, etwa zur Ein- und Ausgabe. Diese Aufgabe übergibt er im Allgemeinen einem darauf spezialisierten Programm, dem *Linker*. Demgemäß bezeichnet man diesen Vorgang auch als »Linken«.

Bibliotheken Manchmal möchte man auch mehrere Unterprogramme in übersetzter Form zusammenfassen, die allein zwar noch kein eigenes Programm bilden, die aber für verschiedene Programme nützlich sein könnten. Solche Codesammlungen bezeichnet man als *Bibliotheken* (engl. *libraries*).

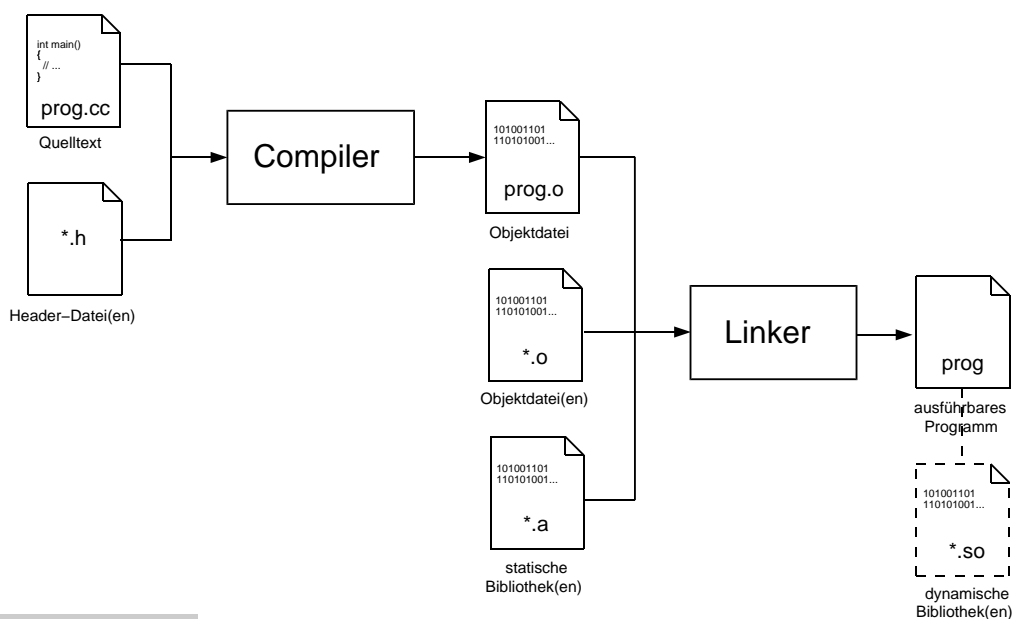


Abbildung 1.3
Compiler und Linker
erstellen die
ausführbare
Programmdatei.

Bibliotheken gibt es übrigens in zwei Varianten:

- als *statische Bibliotheken*, die die Endung `.a` tragen und zur »Compile-Zeit« (wie man sagt) zur ausführbaren Programmdatei dazugelinkt werden, und

- ❑ als *dynamische Bibliotheken*, die erst zur Laufzeit des Programms hinzugezogen werden (entsprechend einer DLL unter Windows). Diese tragen die Endung `.so` (für »shared objects«).

Wenn Sie eigene Bibliotheken verwenden, geben Sie dies dem Linker bekannt, der dann die notwendigen Bibliotheksabschnitte in Ihr ausführbares Programm kopiert. Systembibliotheken werden im Allgemeinen automatisch eingebunden. Wie Sie selbst Bibliotheken erstellen können, erfahren Sie im Abschnitt 3.5 ab Seite 262.

Abbildung 1.3 veranschaulicht nochmals die Zusammenhänge. Beachten Sie dabei, dass für ein einfaches Programm bereits eine einzige Quelldatei genügt. Header-Dateien und weitere Objektdateien benötigen Sie erst bei größeren Projekten, die Sie aus mehreren Dateien zusammensetzen. Dynamische Bibliotheken sind optional und daher gestrichelt gezeichnet.

In Abschnitt 6.1 ab Seite 396 werden wir uns beispielsweise der Frage widmen, wie ein Programm zur Erinnerung an Geburtstage von Freunden gepflegt werden kann und wie dessen Dateien bei Bedarf zu übersetzen sind. Im Spiel sind dabei folgende Dateien:

Beispiel

- ❑ Quelltextdateien: `date.cc` für das Datum, `birthlist.cc` für eine Geburtstagsliste und `birthday.cc` als Hauptprogramm
- ❑ Header-Dateien: `date.h` enthält die Deklarationen zum Datum, `birthlist.h` enthält die der Geburtstagsliste
- ❑ Objektcode: Jede der drei Quelltextdateien wird zu einer Objektdatei kompiliert, also `date.o`, `birthlist.o` und `birthday.o`
- ❑ Bibliotheken: Der Linker bindet selbstständig die nötigen Systembibliotheken hinzu, beispielsweise `libc.a`
- ❑ Ausführbares Programm: `birthcontrol`

1.4.2 Systemdateien zur Entwicklung

Wenn Sie schon mehr Erfahrung mit Linux (oder einem anderen Unix) haben, kennen Sie sicherlich die Konventionen, in welchen Verzeichnissen welche Dateien und Dateitypen abgespeichert und aufbewahrt werden. Für einen Entwickler kann es von Vorteil sein, ein wenig darüber zu wissen, welche Dateien das System bereitstellt und wo er diese finden kann.

Programme

Ausführbare Programme finden Sie üblicherweise in Unterverzeichnissen mit dem Namen `bin`, unter anderem:

- ❑ `/usr/bin` mit den meisten der Programme, die vom System zur Verfügung gestellt werden, darunter oft auch Compiler, Linker und andere Werkzeuge für Entwickler.
- ❑ In `/bin` befindet sich eine kleine Sammlung der elementarsten Systemprogramme (wie Shells, `ls` oder `ps`).
- ❑ `/usr/local/bin` enthält solche Programme, die nicht zum Standard im engeren Sinne gehören beziehungsweise nur für diesen Computer einzeln hinzugefügt wurden. Die Unterscheidung zwischen `/usr/bin` und `/usr/local/bin` wird manchmal bei lokalen Netzen mit mehreren Rechnern gemacht, um eine einheitliche (und damit besser zu pflegende) Konfiguration aller Geräte zu erreichen; Programme, die nicht auf allen, sondern nur auf einem Computer installiert sind, legt man dann unter `/usr/local/bin` ab. Bei Einzelplatzrechnern ist die Trennung meist weit weniger streng, so dass hier oft `/usr/local/bin` gleich ganz leer bleibt.
- ❑ In `usr/X11/bin` finden Sie Programme mit grafischer Benutzeroberfläche, die unter dem X-Window-System laufen. Auch die ausführbaren Dateien zum Starten und Betreiben dieser Oberfläche sind meist hier abgelegt.

Bibliotheken

Die Standardbibliotheken des Systems stehen unter `/lib` und `/usr/lib`, etwa `libm.a` für mathematische Routinen oder `libpthread.a` für die Parallelisierung von Programmen. Ähnlich wie bei den Programmen verwendet man manchmal auch `/usr/local/lib`. Bibliotheken für die X-Window-Programmierung finden sich in `/usr/X11/lib`.

Header-Dateien

Die gerade erwähnten Standardbibliotheken des Systems sind üblicherweise in C geschrieben. Um Funktionen daraus in C- oder C++-Programmen zu nutzen, brauchen Sie auch die zugehörigen Header-Dateien mit den Definitionen der Schnittstellen. Im Verzeichnis `/usr/include` und dessen Unterverzeichnissen finden Sie die meisten davon.

Wie Sie vielleicht wissen, liefert unter MS-Windows im Allgemeinen erst die Entwicklungsumgebung des Compilers die Header-Dateien des Windows-API (der Programmierschnittstelle) mit. Dagegen sind unter Unix die entsprechenden Dateien normalerweise auf jedem System installiert – unabhängig vom gewählten Compiler und auch unabhängig davon, ob Sie Ihren Rechner überhaupt zum Programmieren nutzen wollen oder nicht.

Das genannte Verzeichnis kann auch Dateien für spezielle Bibliotheken in Unterverzeichnissen enthalten, etwa

- ❑ `/usr/include/sys` für betriebssystemnahe Routinen,
- ❑ `/usr/include/X11` für die Bibliotheken zur X-Window-Programmierung oder
- ❑ `/usr/include/g++` für die GNU-Implementierung der C++-Standardbibliothek.

Manche Systemwerkzeuge, die Sie auf Ihrem Rechner installieren, hinterlegen die Header-Dateien zu ihren Bibliotheken auch im Verzeichnis `/usr/local/include`.

Wir werden uns natürlich noch sehr ausführlich mit den Bibliotheken, ihren Funktionen und ihrer Verwendung beschäftigen. Mit diesen Überblicksinformationen fällt es Ihnen indessen sicherlich leichter sich zu orientieren, wo Sie welche Arten von Dateien finden können.

1.5 Übungsfragen

1. Wann darf sich ein Betriebssystem »UNIX« nennen?
2. Nennen und erklären Sie zwei besondere Eigenschaften von Unix?
3. Was ist die Idee der »freien Software«? Nennen Sie drei kennzeichnende Eigenschaften freier Software.
4. Wozu benötigt man Header-Dateien? In welchen Verzeichnissen finden Sie die Header-Dateien für die Systembibliotheken?
5. Sie schreiben ein Programm `fotos.cpp` zur Verwaltung einiger Bilder. Dazu laden Sie sich aus dem Internet eine Programmbibliothek `libjpgcruncher.a` und deren Header-Datei `jpgcruncher.h` herunter, mit deren Hilfe Sie die Bilder in der Auflösung verändern möchten. Die Funktionen, die diese Bibliothek bietet, wollen Sie in Ihrem Programm verwenden. Welche der folgenden Aussagen sind richtig:
 - (a) Die Programmbibliothek `libjpgcruncher.a` ist eine dynamische Bibliothek, die erst zur Laufzeit des Programms eingebunden wird.
 - (b) Sie können Ihre Datei `fotos.cpp` auch zu einer Objektdatei `fotos.o` übersetzen, ohne dass eine der beiden Dateien der Bibliothek verfügbar ist.
 - (c) Das Programm kann erst vollständig getestet werden, wenn `fotos.o` und `libjpgcruncher.a` verfügbar sind.
 - (d) Die Programmbibliothek `libjpgcruncher.a` kann auf allen Computern genutzt werden, auf denen Linux läuft.