Kapitel 1

Einführung

1.1 Über dieses Scriptum

Dieses Scriptum wurde anlässlich der Neugestaltung der bis dato "Unix-Kurs" genannten Veranstaltung im Sommersemester 1999 neu erstellt. Der alte Unix-Kurs wurde zu dieser Zeit in zwei Veranstaltungen aufgeteilt; im Wintersemester werden die Einsteigerthemen, wie Rechnerbenutzung, XWindow, etc. behandelt, im Sommersemester einige speziellere Tools aus der Unix-Umgebung, die im weiteren Verlauf des Studiums von gewisser Bedeutung, z.B. zur Erstellung von Dokumentation bei den FoPras, sind.
In diesem Script, das den Teil des Kurses, welcher im Sommersemester stattfindet, abdeckt, werden zuerst einige allgemeine Tips&Tricks für die Arbeit in der Unix-Umgebung gegeben.
Danach werden in Kapiteln 2 und 3 die Textverarbeitungstools `sed` und `awk` vorgestellt, die eine schnelle Bearbeitung von Daten erlauben, die in Textform vorliegen. Im folgenden Kapitel 4 wird die weit verbreitete Scriptsprache `perl` vorgestellt. Diese bietet auch über `sed` und `awk` hinausgehende Möglichkeiten der Datenmanipulation und hat insbesondere durch den Einsatz im WWW (als Sprache für CGI Scripte) eine große Bedeutung erlangt.
Die Verwaltung verschiedener Files (Programmsourcen, Objektcode, Dokumentationsquellen, etc.) in größeren Projekten schafft durch die untereinander gegebenen Abhängigkeiten der Files oft Probleme; so kann z.B. eine vergessene Neuübersetzung eines geänderten Moduls zu schwer nachzuvollziehenden Fehlern im kompletten Programm führen. In Kapitel 5 wird das GNU `make` Tool vorgestellt, mit dem es möglich ist, diese Abhängigkeiten zu verwalten und nach Änderungen in Teilmodulen alle nötigen Neugenerierungen automatisch auszuführen.
Die Erstellung von Dokumentationen, etwa Hausarbeiten, Seminarausarbeitun-

gen, Diplomarbeiten, FoPra-Dokumentationen, ist eine oft wiederkehrende Aufgabe im Studium. In Kapitel 6 wird daher das LaTeX Textsatzsystem vorgestellt, mit dem Dokumente in professioneller Qualität erzeugt werden können. Zusätzlich behandelt dieses Kapitel auch noch eine Reihe zusätzlicher Tools, mit denen etwa Abbildungen oder Literaturangaben erzeugt werden können.
Bei größeren Programmierprojekten ist eine gründliche Dokumentation unerläßlich. Zusätzlich soll diese auch als Handbuch zu den erstellten Programmen dienen. In Kapitel 7 wird das `texinfo` System vorgestellt, mit dem man aus einer Quelle sowohl ausdruckbare Handbücher, als auch Online-Versionen, in welchen interaktiv herumgewandert werden kann, erstellt werden können.
In Kapitel 8 schließlich wird auf einige Quellen mit zusätzlichen Informationen zu den vorgestellten Tools verwiesen.
Die meisten der Kapitel dieses Scriptes enthalten einen Abschnitt mit einer Reihe von Beispielen, die die dargestellten Konzepte illustrieren sollen.
Als letztes ergeht nun noch die Aufforderung an alle Benutzer dieses Scriptes, Feedback zu geben; sowohl was Fehler im Script betrifft, als auch Themen, die im Rahmen dieses Kurses behandelt werden sollten, oder Beispiele, die in die zugehörigen Abschnitte aufgenommen zu werden verdienen.

1.1.1 Konventionen

In diesem Script werden verschiedene Konventionen benutzt:

- Um auf die Manual-Page (siehe Abschnitt 1.2.3) einer Programmes, eines Files oder einer Funktion zu verweisen, geben wir den Namen der Manual-Page, zusammen mit dem Manual-Abschnitt an, z.B.: *ls(1)*. Wir geben die Manual-Page ebenfalls an, wenn wir eigentlich über das Programm (die Funktion,...) reden.

  Man beachte, daß einige Manual-Pages mehrmals (in verschiedenen Abschnitten) vorkommen, etwa *intro(1)*, *intro(2)* (die Einführungen zu den jeweiligen Manual-Abschnitten).

- Ausgaben auf dem Bildschirm werden in Typewriter dargestellt, etwa `uid=10016(thesing)` als Ausgabe des *id(1)* Kommandos.

- Eingaben des Benutzers werden fett gedruckt dargestellt, etwa **id thesing**.

- Teile, die variabel sind, etwa ein Benutzername, werden kursiv dargestellt, etwa wie in `ls ~`*`username`*`/.netscape`.

1.2 Einige Tips & Tricks

In diesem Kapitel werden einige nützliche Tips & Tricks vorgestellt, mit denen man einige Probleme beheben kann, die bei der Arbeit unter UNIX immer wieder auftauchen. Zuerst wird das wichtige Thema der *Libraries* besprochen, die Unterschiede zwischen statischen und dynamischen Libraries erklärt.
Im folgenden Abschnitt werden einige nützliche Umgebungsvariablen vorgestellt und gezeigt, wie man das Verhalten des Systems mit ihnen beinflußen kann.
Der letzte Abschnitt zeigt auf, wo man Hilfe bei auftretenden Fragen bekommen kann.

1.2.1 Libraries

Libraries sind eine Sammlung von Objectcode-Files, die eine Reihe häufiger benutzter Funktionen enthalten. So enthält etwa die Standard-C Library (`/usr/lib/libc.a`) u.a. die Funktion *printf(3)*, mittels derer man Text ausgeben kann. Abbildung 1.1 illustriert den Weg von einem Sourcecode-File zu einer Library, wobei *gcc(1)* zum Kompileren benutzt wird und *ar(1)* zum Zusammenfügen der Objectcode-Files zu einer Library[1].
Wenn man eigene Libraries generieren will, dann muß man die Objectcode-Files mittels *ar(1)* zusammenstellen und danach einen Index dieses Archivs (Library) erzeugen, indem man *ranlib(1)* aufruft, z.B.:

ar cr libfoo.a file1.o . . . fileN.o
ranlib libfoo.a

Die Optionen `rc` besagen, daß die Files hinzugefügt werden sollen und die Library ggf. neu kreiert wird, wenn sie noch nicht existiert.
NB: Wenn man eine Library neu übersetzt, sollte man die alte Version mittels **rm libfoo.a** vorher löschen, da sonst das Ersetzen (`r` = replace) der alten Objectcode-Files lange dauern kann.
Diese Library kann man dann mittels **-lfoo** beim Linken angeben[2].
Probleme können allerdings auftreten, wenn die Objectcode-Files voneinander abhängen, etwa in `file1.o` Funktionen aus `file2.o` aufgerufen werden. Denn beim Linken wird eine Library nur von vorne nach hinten durchsucht. Taucht in einem Objectcode-File in einer Library eine Referenz auf, muß diese in einem weiter hinten liegenden File aufgelöst werden.

[1] Gibt man *gcc* die Option **-v**, kann man die Aufrufe der einzelnen Stufen der Compilierung verfolgen

[2] Zusätzlich kann es nötig sein, mit **-L*dir*** noch das Directory anzugeben, in welchem die Library liegt.

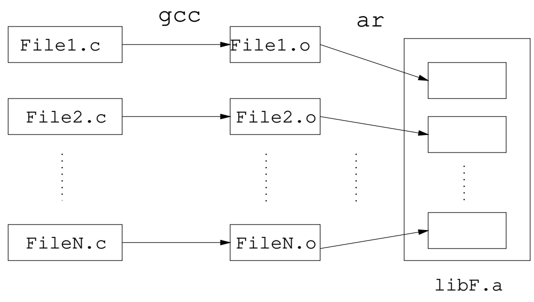

Abbildung 1.1: Vom Sourcecode-File zur Library.

Nun fügt *ar(1)* aber die Objectcode-Files in der Reihenfolge der Library hinzu, in welcher sie auf der Kommandozeile angegeben wurden. Um eine optimale Reihenfolge zu finden, benutzt man am besten die beiden Tools *lorder(1)* und *tsort(1)*. Ersteres erzeugt eine Abhängigkeitsliste, die angibt, welches File Referenzen auf ein anderes File enthält. *tsort(1)* sortiert diese Liste dann so, daß Objectcode-Files, welche referenziert werden, nach den Referenzen stehen. Man benutzt also etwa folgende Befehlsfolge:

ar rc libfoo.a ‘lorder file1.o … fileN.o | tsort‘
ranlib libfoo.a

In obigem Beispiel werden `lorder` und `tsort` in einer Subshell aufgerufen (wegen der einfachen Rückstriche '‘') und deren Ausgabe dann als Fileliste für `ar` eingesetzt.
Bei der letzten Stufe des Übersetzens eines Programmes, dem *Linken*, werden die einzelnen Objectcode-Files des Programmes mit denen aus den Libraries zusammengestellt, welche Funktionen enthalten, die das Programm benutzt. Alle diese Objectcode-Files werden aneinandergehängt und ergeben letztendlich das ausführbare Programm, siehe Abbildung 1.2.
Neben dieser Methode des Linkens (siehe übrigens *ld(1)* und *link(5)* dazu), welche man *statisches Linken* nennt, gibt es noch eine andere Methode, das *dynamische Linken*: Beim statischen Linken wird jedesmal eine Kopie der aus einer Library benötigten Objctcode-Files in das Programm übernommen, verbraucht also Platz auf der Platte und (bei der Ausführung des Programmes) auch im Speicher.
Vor allem bei sehr häufig genutzten Libraries, etwa `libc.a`, `libX11.a`, hat man also u.U. eine große Anzahl von Kopien. Außerdem muß man bei einer Änderung der Library (etwa einem Update des XWindow-Systems) alle Programme neu übersetzen, damit sie die neue Library verwenden können. Beim dynamischen Linken hat man hingegen nur (höchstens) eine Kopie jeder Library im Speicher. Alle Programme, die einen Teil dieser Library benutzen, verwenden

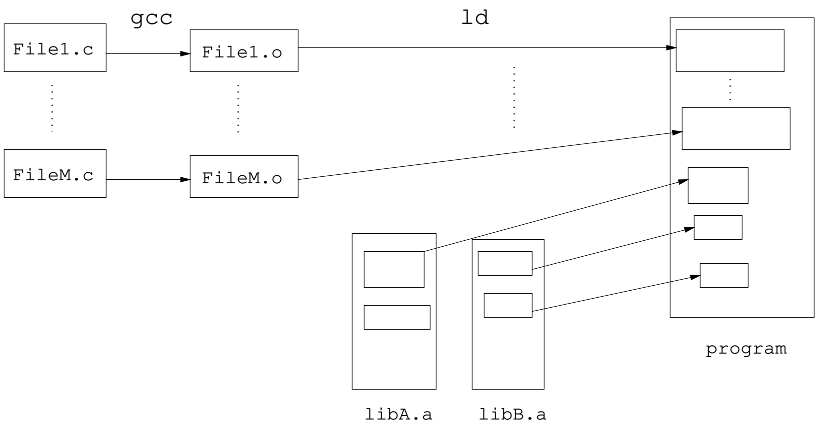

Abbildung 1.2: Vom Sourcecode zum Programm (statisch)

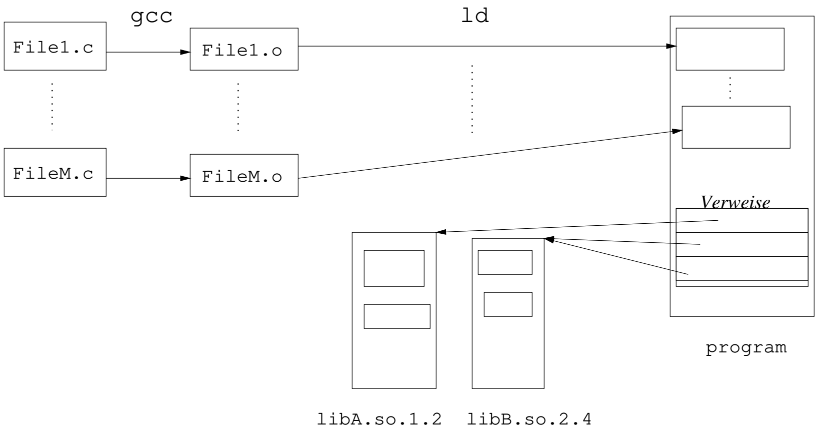

Abbildung 1.3: Dynamisches Linken

dieselbe Kopie im Speicher. Beim Linken werden nun nicht mehr die einzelnen Objectcode-Files aus der Library in das Programm übernommen, sondern nur Verweise auf die Library im Programm-File eingetragen. In Abbildung 1.3 ist dieser Sachverhalt noch einmal dargestellt.
Beim Laden des Programmes löst das Betriebssystem, mittels des *ld.so(1)* dynamischen Linkers[3], diese Verweise auf:

- Wenn die referenzierte Library noch nicht im Speicher ist (also dieses Programm das erste ist, welches einen Teil dieser Library braucht), dann wird die Library gesucht und in den Speicher geladen.

[3]Dieser wird vom Startupcode des Programmes geladen und ausgeführt.

- Ist die Library im Speicher, werden alle Referenzen im Programm auf die Library an die tatsächlichen Speicheradressen der Library angepaßt.

Dabei muß allerdings die Library auf eine spezielle Weise erzeugt worden sein, damit nämlich alle Referenzen in ihr unabhängig zu ihrer Startadresse im Speicher (die für verschiedene Programme variieren kann) auftreten[4]. Bei der Kompilierung von C-Programmen muß man z.B. die Option `-fPIC` (oder `-fpic`) angeben, siehe *gcc(1)*. Auch das Zusammenstellen der einzelnen Objectcode-Files zur Library muß speziell erfolgen, so daß man nicht die selben Library-Files für statisches und dynamisches Linken verwenden kann.
An dieser Stelle sei auf das GNU `libtool` Programm verwiesen[5], das sich um all diese speziellen Kommandos kümmert und vor allem auf verschiedenen Plattformen verwendet werden kann.
Allgemein haben statische Libraries immer die Endung `.a`, dynamische Libraries immer etwas in der Form von `.so.`*`Major`*`.`*`Minor`*. Hierbei gibt *Major* die Hauptversion und *Minor* die Unterversion der Library an. Damit werden mehrer Versionen derselben Library unterstützbar.
Die dynamischen Libraries werden beim Laden eines Programmes gesucht, wenn sie noch nicht im Speicher vorhanden sind. Damit *ld.so(1)* sie finden kann gibt es mehrere Mechanismen. Zum einen wird ein Cache mit Library-Namen und ihren Positionen auf der Platte angelegt, der das Suchen dynamischer Libraries beschleunigt. Dieser wird (meist beim Booten des Systems) mittels *ldconfig(8)* angelegt und kann i.d.R. nur vom Superuser geändert werden. Zum anderen sucht *ld.so(1)* in Verzeichnissen, welche in der Umgebungsvariablen `LD_LIBRARY_PATH` (durch Doppelpunkte getrennt) angegeben werden; etwa wird nach **setenv LD_LIBRARY_PATH .:$HOME/lib** im aktuellen Verzeichnis und im Unterverzeichnis `lib` im Home-Verzeichniss nach Libraries gesucht. Als letztes wird auch noch in den Directories gesucht, welche mit `-L`*`dir`* beim Linken des Programmes angegeben wurden.
Findet `ld.so` eine dynamische Library `libfoo.so.`*`Major`*`.`*`Minor`*, dann

- Falls die *Major* Nummer der gefundenen Library sich von der Nummer der gesuchten Library unterscheidet, wird sie ignoriert.

- Es wird immer die Library mit der höchsten Minor Nummer (und gleichen Major Nummern) genommen. Dadurch kann immer die aktuellste Version einer Library verwendet werden. Ist die höchste gefundene Minor Nummer *kleiner* als die Minor Nummer, die beim Linken des Programmes gefunden

[4] Es darf also nur relative Adressierung benutzt werden.
[5] Siehe etwa unter `http://www.gnu.org`

wurde (kann z.B. passieren, wenn man ein Programm auf einer SPARC-10 linkt und dann auf einer SPARC-1 ausführt), dann wird eine Warnung ausgegeben, die Library aber trotzdem akzeptiert.

Wenn man wissen will, welche dynamischen Libraries von einem Programm benutzt werden, gibt es zwei Wege:

- Man setzt die Umgebungsvariable `LD_TRACE_LOADED_OBJECTS` in der Shell auf irgendeinen Wert (etwa mittels des Kommandos **setenv LD_TRACE_LOADED_OBJECTS**) und ruft dann das zu untersuchende Programm einfach auf. Statt das Programm auszuführen, werden nun alle dynamischen Libraries, die das Programm benötigt, ausgegeben (und ihre Filenamen, sollten sie gefunden werden). Natürlich muß man, wenn man danach wieder Programme ausführen will, die Umgebungsvariable wieder löschen (**unsetenv LD_TRACE_LOADED_OBJECTS**).

- Alternativ kann man das *ldd(1)* Programm aufrufen: **ldd *program***, was zur selben Ausgabe führt. (Bem.: Unter NetBSD oder Linux ist *ldd(1)* nichts anderes, als ein Wrapper, der die obige Umgebungsvariable setzt und dann das Programm ausführt! Ob dies auch für SunOS zutrifft kann ich mangels Sourcecode nicht sagen, vermute es aber.)

Beim Kompilieren und Linken eigener Programme ist das dynamische Linken immer als Default eingestellt. Gibt man also eine Library mittels **-lfoo** zum Linken an, dann wird zuerst nach derjenigen dynamischen Library `libfoo.so.`x`.`y gesucht, die die höchste Major Nummer x hat. Bei mehreren dynamischen Libraries mit derselben Major Nummer wird diejenige mit der höchsten Minor Nummer y genommen.
Nur wenn keine dynamische Library gefunden wird, wird (ebenfalls unter Verwendung von `LD_LIBRARY_PATH`) nach einer statischen Library `libfoo.a` gesucht. Manchmal ist es aber wünschenswert, das statische Linken zu erzwingen. Dies kann man *gcc(1)* mittels der **-static** Option mitteilen. Mit dieser Option werden dann keine dynamischen Libraries gesucht.
Man kann auch gegen einige Libraries dynamisch, gegen andere statisch Linken. Siehe hierzu `ld(1)`.

1.2.2 Environment Variablen

Environment Variablen beeinflussen das Verhalten vieler Programme. So verwenden z.B. fast alle Programme den mittels der Variablen `EDITOR` angegeben Editor, wenn der Benutzer aus ihnen heraus Text editieren muss (z.B. *mail(1)*).
Eine Liste der gängisten Variablen ist in *environ(5)* zu finden.
Noch zu erwähnen sind:

- `MANPATH`: Diese Variable enthält, durch Doppelpunkte getrennte, Directories, in denen nach Manual-Pages gesucht wird. In jedem so angegebenem Directory sollten Subdirectories mit Namen `cat`*`section`* oder `man`*`section`* existieren, in welchen die formatierten (`cat`) oder rohen Manual-Pages (`man`) stehen. Der Default für den Suchpfad nach Manual-Pages ist `/usr/share/man`. Nützlich ist dies, wenn man Software selbst installiert hat und auch deren Manual-Pages angezeigt bekommen möchte. Legt man die Seiten etwa unterhalb des `$HOME/man` Directories an, so werden die lokalen Seiten nach einem `setenv MANUALPATH $HOME/man:/usr/share/man` ebenfalls von *man(1)* angezeigt.

- `INFOPATH`: Das `info` Kommando zeigt ebenfalls Informationen zu Programmen an (siehe Abschnitt 1.2.3). `INFOPATH` enthält eine Liste (durch Doppelpunkte getrennt) von Directories, in denen nach Info-Seiten gesucht wird, entspricht in seiner Funktion also `MANPATH`. Der Default für die Suche ist i.A. `/usr/local/info`.

Im folgenden werden noch einige Environment Variablen vorgestellt, die von der *tcsh(1)* verwendet werden, um deren Verhalten zu beeinflußen.

tcsh Variablen

Die *tcsh(1)* ist eine oft benutzte interaktive Shell, die wohl die meisten Benutzer als ihre Login-Shell benutzen. Ihr Verhalten kann durch das Setzen von lokalen Variablen vielfältig beeinflußt werden.
Lokale Variablen werden in dieser Shell mittels **set** ***var=value*** gesetzt, im Gegensatz zu globalen Variablen, die mittels **setenv** ***var value*** gesetzt werden. Der Unterschied zwischen beiden besteht darin, daß nur globale Variablen an Programme weitergegeben werden.

- `autolist`: Nach einer versuchten Completion (mittels TAB oder CTRL-D), werden alle Möglichkeiten der Vervollständigung (alle in Frage kommenden Filenamen, etc.) automatisch angezeigt.

- `cdpath`: Diese Variable enthält eine Liste von Directories und wird mittels **set cdpath=(*path1 path2* . . .)** gesetzt. Will man mittels **cd** in ein Directory wechseln, das nicht im akuellen Directory existiert, dann wird dieses Directory in den Directories von `cdpath` gesucht und dahin gegangen.

- `fignore`: Diese Variable enthält eine Liste von Endungen, die bei der Completion ignoriert werden sollen. Will man z.B. keine `.dvi` Files oder Objectcode Files angezeigt bekommen, kann man dies mittels **set fignore=(.dvi .o)** erreichen.

- `histdup`: In der `tcsh` sind alle Kommandos in der History-Liste gesammelt, so daß man durch einfaches Blättern mittels der Cursortasten vorherige Kommandos wieder erreichen kann. Die Variable `histdup` steuert, was mit doppelten Einträgen in der Histroy-Liste passiert (etwa, wenn man dasselbe Kommando mehrmals hintereinander eingibt). Setzt man sie auf `prev` (**set histdup=prev**), so wird ein Kommando nur dann in die History-Liste aufgenommen, wenn es vom vorherigen Kommando verschieden ist. Setzt man sie auf den Wert `all`, dann wird das Kommando nur in die Liste aufgenommen, wenn es noch gar nicht in ihr vorkommt. Ein Setzen auf den Wert `erase` bewirkt, daß das Kommando am Ende der Liste aufgenommen wird und alle anderen Vorkommen desselben Kommandos aus der Liste entfernt werden.

- `implicitcd`: Wird diese Variable gesetzt, dann kann man in ein Unterverzeichnis wechseln, indem man einfach dessen Namen eingibt (und das sonst übliche `cd` davor wegläßt).

- `notify`: Ist diese Variable gesetzt, wird die Beendigung einer Hintergrundjobs (mittels **&** gestartete Kommandos) sofort bei der Beendigung des Jobs auf dem Bildschirm gemeldet. Ansonsten wird bis zum nächsten Drücken der Return-Taste gewartet.

- `rmstar`: Ist diese Variable gesetzt, dann wird bei der Eingabe von **rm *** nachgefragt, bevor alle Files gelöscht werden.

Als weiteres nützliches Schmankerl kann man in der tcsh unter XWindow den Rechnernamen und das aktuelle Directory[6] in der Titelleiste anzeigen lassen. Dazu muß man einen besonderen Alias namens `cwdcmd` besetzen, etwa folgendermassen[7]:

```
alias cwdcmd
 'echo -n "^[]2;${HOST}:$cwd^G"'
```

Hierbei steht `^[` für den ASCII Code 27 (Escape) und `^G` für Control-G. Eingeben kann man dies z.B. im `vi` mittels '**CTRL-VESC**' bzw. **CTRL-VCTRL-G**. Auch auf der Kommandozeile der Shell kann man mit CTRL-V das folgende Zeichen quoten, es also direkt übernehmen.
Die Shell verfügt über eine ganze Reihe zusätzlicher Variablen, ein Studium von *tcsh(1)* ist daher oft nützlich, wenn auch langwierig.

[6]Das nebenbei bemerkt immer in der Variablen `$cwd` steht.

[7]Die folgenden Zeilen sind nur der Lebarkeit halber im Script umgebrochen, eingeben sollte man sie als eine Zeile.

1.2.3 Hilfe!

Auf UNIX Systemen gibt es mehrere Wege, um an Information zu Programmen zu gelangen:

- Meistens gibt das Programm selbst, wenn z.B. ohne Optionen oder mit **--help** [8] oder **-h** aufgerufen eine kurze Meldung mit den verfügbaren Optionen aus.
- Mit ziemlicher Sicherheit existiert für das Programm eine Manual-Page. Diese kann mit dem `man` Programm aufgerufen werden. Z.B. liefert **man man** Hilfe über das `man` Programm selbst.

  Die Manual-Pages sind nach Abschnitten geordnet, etwa Programme, Systemaufrufe, Funktionen, etc.pp, dabei sind die Abschnitte durchnumeriert (1,...,8,l,n):

  - In Abschnitt 1 werden Benutzerkommandos beschrieben.
  - Abschnitt 2 widmet sich den Systemaufrufen.
  - Die Library-Funktionen werden in Abschnitt 3 behandelt.
  - Abschnitt 4 beschreibt Devices (z.B. Tastatur oder `/dev/null` (*null(4)*), das "Schwarzes-Loch-Gerät".)
  - In Abschnitt 5 findet man Informationen zu Fileformaten, etwa von `/etc/passwd`: *passwd(5)*.
  - Die verfügbaren Spiele werden in Abschnitt 6 beschrieben.
  - Einige nützliche Sachen, die sonst nirgendwohin passen, sind in Abschnitt 7 zusammengestellt.
  - Abschnitt 8 gibt Informationen zu Kommandos, die i.A. nur für den Systemverwalter interessant sind.
  - In Abschnitt l werden lokale Kommandos beschrieben.

  Ein Aufruf von **man 1 printf** gibt z.B. die Erläuterungen zu dem `printf` Programm, während **man 3 printf** die Erläuterungen zu der Funktion `printf` liefert. Eine Einführung in die verschiedenen Abschnitte der Manual-Pages erhält man mittels **man *section* intro**.

  `man` hilft nur weiter, wenn man den Namen des Programmes (der Funktion,...) bereits kennt. Sucht man nach Informationen zu einem bestimmten Schlagwort, hilft *apropos(1)* weiter, das alle Manual-Pages, in denen das

[8] Fast alle Programme aus dem GNU Projekt unterstützen diese Option. SunOS Programme dagegen nicht.

Schlagwort vorkommt, auflistet. Ebenso sollte man sich *whatis(1)* einmal zu Gemüte führen.

- Viele GNU Programme, etwa `make`, `texinfo` kommen neben rudimentären Manual-Pages auch noch mit einer anderen Form der Dokumentation, den *info*-Einträgen. Diese stellen eine verzeigerte Sammlung von Seiten dar (ähnlich einem HTML Dokument), in denen man aus Menues Punkte auswählen und auch Querverweisen folgen kann. Das `info` Programm dient dabei zur Navigation innerhalb dieser Einträge. In Kapitel 7 werden wir das `texinfo` System kennenlernen, mit dem man genau solche Info-Seiten erzeugen kann (und auch noch normale, druckbare Dokumentation!). Anfangs ist die Navigation innerhalb der Info-Seiten etwas gewöhnungsbedürftig (vor allem für die GUI, ein-Knopf-für-alles verwöhnten Benutzer von heute), allerdings leicht zu erlernen. Eine Einführung in `info` erhält man (wen verwundert's) mittels **info info**.

- Für alle die, die ohne das Web nicht mehr leben können gibt es unter der URL `http://www.delorie.com/gnu/docs` Dokumentationen zu den meisten GNU Programmen.