Inhaltsverzeichnis

1
Plan 9
from Outer Space

1.1 Einführung

HERRSCHER: Welchem Plan wirst du nun folgen?

EROS: Plan 9 — es war unmöglich, mit diesen Erdlingen zu arbeiten — ihre Seelen zu kontrollieren.

HERRSCHER: Plan 9 (nimmt den Bericht zur Hand). Ah ja ... Plan 9 befaßt sich mit der Wiederauferweckung der Toten. Langstreckenelektroden werden in das Rückenmark kürzlich Verstorbener geschossen. Habt ihr schon versucht, diesen Plan in die Tat umzusetzen?

EROS: Ja ...

HERRSCHER: Wie erfolgreich wart ihr?

EROS: Bis jetzt haben wir zwei auferweckt — aber wir werden noch erfolgreicher sein ...

HERRSCHER: Die Lebenden — sind eurem Tun noch nicht auf die Schliche gekommen?

EROS: Wir sahen uns gezwungen, uns eines Polizisten zu entledigen. Aber keiner der Wiederauferstandenen ist bis jetzt gesehen worden. Zumindest von niemand, der noch lebt ...

Aus dem Drehbuch von *Plan 9 from Outer Space* von Edward D. Wood Jr.

Gedreht wurde der Film innerhalb von vier Tagen in der letzten Novemberwoche 1956 in den Merle Cornells Quality Studios unter dem Titel *Grave Robbers From Outer Space.* Die Preview fand am fünften März 1957 im Carlon Theater in L.A. statt. Der Kinostart für die USA verzögerte sich bis zum Juli 1959.

Der Film ist in aller Regel nur unter Cineasten bekannt und gilt nicht als ein Meisterwerk seiner Zunft. Auf der Liste der schlechtesten Filme* findet man *Plan 9* auf Platz 40. Schlechter gewertet wurden nur Filme wie *Police Academy 5* und *Over the Top*.

Trotzdem wählte die Gruppe um Dennis Ritchie, Ken Thompson und Rob Pike *Plan 9* als Namen für ein neues, verteiltes Betriebssystem, das für verschiedene Hardware-Architekturen in den letzten Jahren in den Bell Laboratorien entwickelt wurde und seit Anfang 1995 Jahres verfügbar ist. Plan 9 wurde fast von derselben

---

* http://www.msstate.edu/Movies/search.html

Gruppe entwickelt, die uns Unix, C und C++ beschert hat: Sean Dorward, Tom Duff, Bob Flandrena, Tom Killian, Jim McKie, Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom und Dennis Ritchie als Leiter der Gruppe.

Plan 9 ist zwar ein experimentelles System, aber es enthält einige faszinierende neue Ideen. Wie bei UNIX sind Dateinamen noch immer Pfade, deren Komponenten durch Schrägstriche getrennt sind, und liebgewordene Programme von *awk* über *ed* bis *yacc* existieren nach wie vor, aber sonst sind alle weiteren Ähnlichkeiten zu UNIX unbeabsichtigt und eher mißverständlich. Vom Super-User über harte und symbolische Links bis zum *vi* wurden Altlasten gründlich entfernt und durch elegante, kohärente neue Mechanismen ersetzt.

Das System wurde unter anderem nach folgenden Prinzipien entworfen:

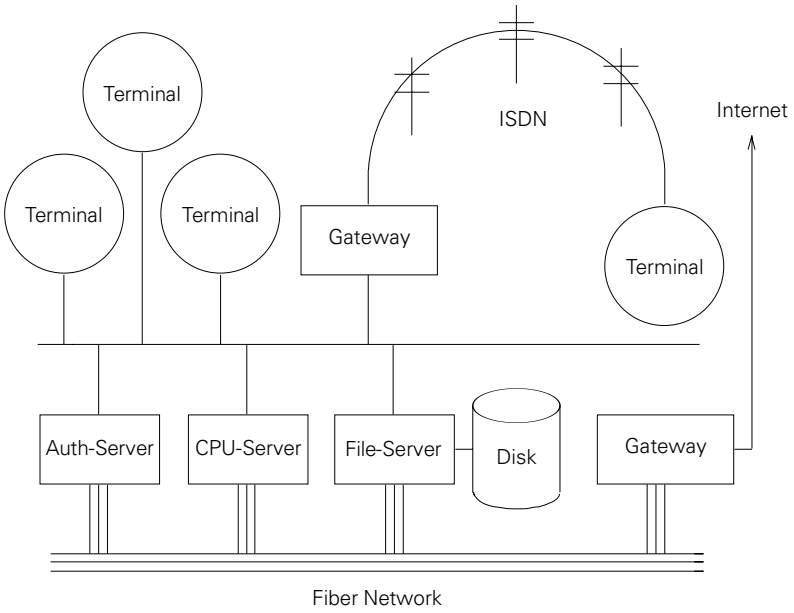
- Alle Ressourcen wie Netzverbindungen, Geräte usw. sind benannt und auf sie kann wie auf Dateien von einem hierarchischen Dateibaum aus zugegriffen werden.
- Es gibt ein Standard-Protokoll namens 9P, unter dessen Verwendung auf alle Ressourcen zugegriffen wird.
- Die von verschiedenen Servern angebotenen, ursprünglich nichtzusammenhängenden Hierarchien werden innerhalb von prozeßeigenen Namensräumen zusammengeführt.

Normalerweise besteht Plan 9 aus einem Netz von Computern, in dem verschiedene Hardware mit verschiedener Software verschiedene Aufgaben übernimmt: Man sitzt an einem *Terminal*, das ein Window-System und lokale Prozesse wie etwa den Editor *sam* bearbeitet [`unix/mail` 3/93], und kontrolliert Prozesse auf einem *CPU-Server*, die zum Beispiel C-Quellen von einem *File-Server* holen und übersetzen. Die Authentifizierung von Terminals und Nutzern erfolgt unter Inanspruchnahme eines Authentifizierungsservers, welcher im Netz bekannt ist. Alle drei Computer sowie das Ziel der Übersetzung können verschiedene Hardware-Architekturen sein; alle benützen zwar verwandte, aber genau für ihre speziellen Aufgaben zugeschnittene Betriebssysteme und ein einziges, sehr elegantes Kommunikationsprotokoll *9P*, das die Architektur des Gesamtsystems entscheidend geprägt hat.

Computer sind billig, und die immer kürzeren Innovationszyklen zwingen dazu, Systeme mit vollkommen austauschbaren Hardware-Komponenten zu konzipieren. Typische große Netze sind deshalb so schwer zu verwalten, weil jede Workstation mit lokaler Platte letztlich als eigenständiges System funktioniert und wohl auch als solches explizit verwaltet werden muß. Andrew Tanenbaum verglich Workstations mit Zahnbürsten (''keiner kriegt meine'') — dank Wasserleitung und Kanalisation sind wir heute von den öffentlichen Badehäusern zwar unabhängig, aber wir nehmen noch immer ganz gern die Leistungen eines Frei- oder zur Not auch Hallenbads in Anspruch.

Terminal mit Systemanschluß heißt deshalb die Plan-9-Devise, und das (relativ) zentrale System besteht im Idealfall aus dedizierten File-Servern und CPU-Servern mit schnellen lokalen Verbindungen untereinander und relativ langsamen Leitungen zu den Terminals. Nur der File-Server enthält einen persistenten Zustand, denn er

verwaltet Platten. Alle anderen Systeme können jederzeit ersetzt und neu gestartet werden. Das Verwaltungsproblem reduziert sich damit wie früher auf ein zentrales System, den File-Server, und in den ist Backup in Form einer Hierarchie aus Hauptspeicher, Magnetplatte und optischen WORM-Laufwerken *a priori* eingebaut. Der Aufbau eines typischen Plan 9-Netzes sieht somit wie folgt aus:



1.2 Vom Dateisystem zum Namensraum

Selbst *diskless* entwickelt eine UNIX-Workstation immer ein Eigenleben. Man lädt einen Kern, verbindet ein Dateisystem als Wurzel und montiert den Rest vom Netz. *automount*() [`unix/mail` 5/92] reduziert die Wartezeiten beim Systemstart und hält trotzdem alles Nötige für den Fall der Fälle vor. Jede Workstation hat nach kurzer Zeit ihre eigene Sicht der Datei-Welt — der nachbarliche Arbeitsplatz dürfte für die eigene Arbeit vollkommen unbrauchbar sein.

Natürlich kann man dies auch mit UNIX besser organisieren: Man richtet einfach alle Workstations gleich ein und montiert alle Dateisysteme aller Anlagen überall. Jede Rechnerbetriebsgruppe kann aber ein Lied davon singen, wie wenig interoperabel trotz aller anderslautenden Schwüre die Systeme verschiedener Hersteller sind und wie zäh zum Beispiel ein Universitätsnetz wird, in dem sich der *backbone* hauptsächlich mit dem NFS-*broadcast*-Verkehr der verschiedenen Fachbereiche beschäftigt. Aus reiner Verzweiflung kauft man eine kleine Platte und hält seine Daten lokal vor, damit die Workstation überhaupt noch etwas tut. Das Rechenzentrum betreibt einen schnelleren (und vor allem teureren) Zentralrechner, aber wenn dieser mit den lokalen Daten einer Workstation arbeiten soll, ist man der Netzkapazität wieder hoffnungslos ausgeliefert.

Spätestens im zentralen CPU-Server treffen sich die verschiedensten Benutzer, und wenn dort eine maschinenweite Sicht der Datei-Welt vorherrscht, ist jeder mit allen verbunden, und aus Effizienzgründen muß sich die Individualität auf ein absolutes Minimum beschränken.

Plan 9 geht korrekt und konsequent davon aus, daß in einem Netz grundsätzlich ein Unterschied besteht zwischen einem physikalisch irgendwo gespeicherten Dateisystem und der logischen Sicht, die irgendein System dafür präsentiert. Nur — während UNIX diese Sicht maschinenweit aufbaut, ordnet sie Plan 9 den Prozessen einzeln zu: Bei UNIX ändert *mount*() die Sicht der Dateienwelt für alle Prozesse im gesamten laufenden System, bei Plan 9 beziehen sich *mount*(1,2) und sein Partner *bind*(1,2) nur auf eine Gruppe von Prozessen. Der *Namensraum* als Sicht der Datei-Welt und anderer Ressourcen ist Eigenschaft eines Prozesses und wird bei *rfork*(2) wahlweise für den neuen Prozeß kopiert oder mit dem neuen Prozeß gemeinsam benutzt.

Die Arbeit am Terminal beginnt bei Plan 9 damit, daß man einen Kern lädt — in der Regel vom Netz — und einen Namensraum aufbaut, der auch ein Dateisystem enthält. Das Terminal beschäftigt sich dann vor allem damit, ein Window-System zu betreiben und die Namensräume für die verschiedenen Windows zu unterhalten. Verbindet man ein Window mit einem CPU-Server, so wird für die eigenen Prozesse dort der gleiche Namensraum aufgebaut, und ein Prozeß hat auf beiden Systemen die gleiche Sicht der Datei-Welt.

Gleiche Sicht muß jedoch nicht die gleiche Abwicklung bedeuten. Der CPU-Server verwendet zwar die gleichen Pfade zu Dateien wie das Terminal, aber er hat in der Regel eine wesentlich schnellere Verbindung zum gleichen File-Server. Im Terminal führt */dev/cons* ziemlich direkt durch den Kern zur Terminal-Emulation in einem Window; beim CPU-Server ist */dev/cons* so im Namensraum gebunden, daß es über das Netz und den richtigen Terminal-Kern ebenfalls zur Terminal-Emulation im zuständigen Window führt. Terminal oder CPU-Server, Window-System oder nicht: Dank dem Unterschied zwischen Namensraum und Abwicklung kann sich ein Prozeß immer darauf verlassen, daß */dev/cons* zu einer Terminal-Emulation führt, über die man Ein- und Ausgaben tätigen kann. Das Window-System *8½* ist ein File-Server-Prozeß, der die Datei */dev/cons* für den jeweiligen Namensraum eines Windows individuell zur Verfügung stellt.

1.3 Lokaler Umbau des Namensraums

Sichten auf Dateisysteme oder deren komplette Unterbäume kontrolliert zwar auch UNIX-*mount*(), aber dessen Fähigkeiten erstrecken sich nicht auf verschiedene Interpretationen eines Geräts wie */dev/cons*. Plan 9 hat zwei Systemaufrufe zur Manipulation von Namensräumen: *bind*(1,2) für Pfade im Namensraum und *mount*(1,2) für Verbindungen nach außerhalb.

```
bind(neu, alt, how);
```

verknüpft die existenten Pfade *alt* und *neu*. Je nach Wert von *how* verdeckt der Inhalt von *neu* den Inhalt von *alt* vollständig, oder es entsteht ein kombinierter Katalog mit Namen *alt*, in dem der Inhalt von *neu* vor oder nach dem Inhalt von *alt* gefunden

wird. Außerdem kann man kontrollieren, ob *neu* unter diesen Umständen schreibbar sein soll. Näheres siehe Kapitel 4. *bind*(1,2) existiert auch als Kommando. Beispielsweise sorgt

```
% bind -bc /usr/axel/bin /bin
```

dafür, daß die privaten Kommandos von *axel* in */usr/axel/bin* vor (–*b*) den öffentlichen Kommandos in */bin* gefunden werden und daß der lokale *bin*-Katalog geändert werden kann (–*c*), das heißt, daß Operationen wie

```
% echo '#!/bin/rc' > /bin/now
% echo date >> /bin/now
% chmod +x /bin/now
```

ein Kommando *now* im lokalen Katalog */usr/axel/bin* anlegt, denn er ist dann der erste schreibbare Teilkatalog von */bin*.

Der Systemaufruf *bind*(2) ersetzt durchaus die Fähigkeiten, die sonst durch *PATH* in der Shell realisiert werden:

```
% bind /386/bin /bin
```

sorgt dafür, daß sich nur die für die *386* Architektur vorgesehenen Kommandos in */bin* befinden — dabei bleibt allerdings noch die Frage offen, woher das Kommando *bind*(1) kommen soll, das dies bewerkstelligt...

1.4 Server im Kern

Im Kern selbst existiert ein kleiner Namensraum, der einige wenige leere Kataloge wie */bin* oder */dev* enthält. Außerdem bedienen die Gerätetreiber jeweils eigene Dateisysteme, deren Namen mit # und einem Buchstaben beginnen, zum Beispiel:

`#b`	Bitmap-Gerät
`#c`	Konsole
`#e`	Environment
`#f`	Floppy
`#I`	Internet-Protokolle
`#l`	Ethernet
`#s`	Server-Registratur
`#w`	Festplatte

Wie üblich wird ein Programm *init*(8) als erster Prozeß zur Ausführung gebracht. Dieses Programm interpretiert eine Datei */lib/namespace*, in der eine Reihe von *bind*-Anweisungen stehen, zum Beispiel:

```
bind #c /dev
```

Damit enthält der Namensraum dann Pfade wie die Konsole */dev/cons*, die Uhrzeit */dev/time* usw.

1.5 Erweiterung des Namensraums

Der Systemaufruf *mount*(2) beschäftigt sich mit Verbindungen zur Außenwelt. In einem UNIX-System benötigte man dazu ursprünglich ein Gerät (Platte) mit einem Dateisystem und einen Katalog, bei Plan 9 hat dies inzwischen folgende Form:

```
mount(fd, old, how, name);
```

fd ist eine Kommunikationsverbindung für einen sicheren Message-Strom, zum Beispiel auf der Basis einer lokalen Pipe oder von TCP oder dem neuen IL-Protokoll, einem sicheren Datagramm-Service, den Plan 9 im Internet bevorzugt verwendet. *old* ist ein existenter Katalog im Namensraum, und *how* kontrolliert wie bei *bind*(2) die möglichen Operationen. *name* kann dazu verwendet werden, einen bestimmten Dateibaum eines Servers zu spezifizieren. Dieses Argument ist optional. *mount*(2) sorgt nun dafür, daß ein *mount*-Treiber Systemaufrufe, die sich auf Pfade beziehen, die mit *old* beginnen, in *9P*-Messages übersetzt und über die Verbindung *fd* schickt. Aus den Antworten werden dann die Resultate der Systemaufrufe gebildet.

Auch für *mount*(2) existiert ein Kommando. Startet man einen lokalen Server, so hinterlegt er i.a.R. nach Konvention seine Verbindung in einem Katalog */srv*, und man kann etwa folgenden Dialog führen:

```
% tmpfs
% mount -c /srv/tmpfs /tmp
```

Angenommen, *tmpfs* ist ein Server, der seine Verbindung als */srv/tmpfs* hinterlegt, dann ist jetzt dieser Server für den kompletten Katalog */tmp* zuständig.

Einen derartigen Server gibt es tatsächlich: */tmp* ist in der Regel schreibgeschützt. Der Terminal-Benutzer verwendet normalerweise *bind*(1) und überdeckt */tmp* mit seinem eigenen Katalog für temporäre Dateien. Bootet man ein Terminal aber mit einer CD als Wurzel, so liefert *ramfs*(1) ein *in-core* Dateisystem für */tmp*.

1.6 Server

Es gibt eine Vielzahl nützlicher Server, die man mit *mount*(1) verwenden kann, zum Beispiel:

9660srv	ISO-9660-Dateisystem
dossrv	DOS-Dateisystem
ftpfs	*ftp*-Verbindung
u9fs	UNIX-Dateisystem

Da Plan 9 keine Links besitzt, können ziemlich primitive Dateisysteme ganz gut abgebildet werden. Die Technik eignet sich besser zum Import als für Produktionszwecke, denn die Namenskonventionen von DOS oder ISO-9660 müssen natürlich eingehalten werden, und der UNIX-ähnliche Zugriffsschutz von Plan 9 ist aufgehoben.

ftpfs(4) richtet eine FTP-Verbindung mit einem beliebigen System ein, die dann per *mount*(1) im Namensraum als Dateisystem für Import und Export von Dateien verwendet werden kann.

u9fs(1) ist ein Programm, das man auf einem UNIX-System übersetzen und dann mit dem *inetd*-Dämon als Service anbieten kann. Ein Plan 9-System erreicht über diesen Service einen UNIX-Dateibaum und kann ihn per *mount*(1) in Namensräume einbinden.

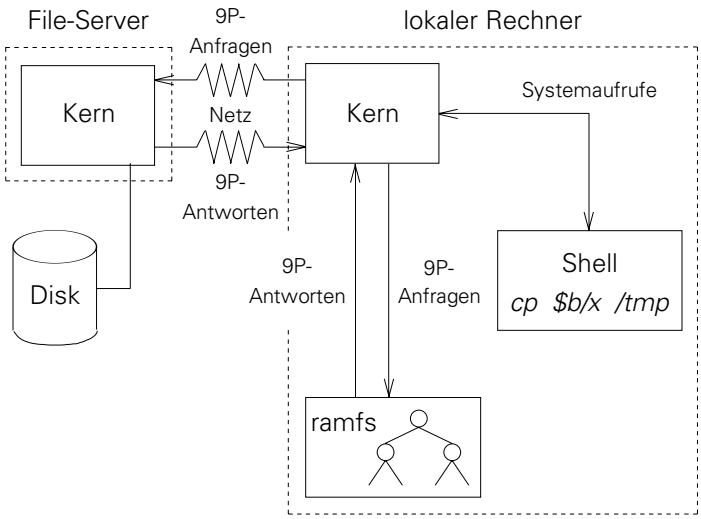
1.7 Alles 9P

Im Lauf der Jahre oder Jahrzehnte hat UNIX eine Vielzahl von Protokollen adoptiert. Für Dateisysteme gibt es NFS und RFS, für Terminals *telnet* und *rlogin*, zum Kopieren von Dateien *ftp*, *rcp*, *kermit*, *tip* und viele, viele andere. Rechnet man die Gateways zu globalen Netzen hinzu, so ist die Vorstellung von Babel und seinen Folgen naheliegend.

Plan 9 macht mit all dem Schluß. Zur Kommunikation mit Dateisystemen — nah und fern — gibt es ein einziges Protokoll, 9P, unter anderem mit folgenden Nachrichten:

`attach`	Verbindung einrichten
`clone`	Verbindung kopieren
`clunk`	Verbindung aufgeben
`walk`	Pfadkomponente suchen
`stat`	Attribute eines Objekts
`wstat`	Attribute ändern
`create`	Objekt erzeugen
`remove`	Objekt löschen
`open`	Zugriffsart prüfen
`read`	lesen
`write`	schreiben

Der Betriebssystem-Kern und alle Server verhandeln untereinander mit Hilfe des zustandsbehafteten Netzwerkprotokolls 9P. Die einzelnen Nachrichten von 9P dienen zur Manipulation von Dateien. Man kann mit ihnen durch die Dateibäume der Server wandern, Dateien lesen und schreiben sowie Dateien kreieren und löschen. 9P ist als ein Satz von Transaktionen strukturiert. Eine Transaktion besteht aus einer Anfragenachricht des Klienten an den lokalen oder entfernten Server und der zugehörigen Antwortnachricht des Servers an den Klienten. Die nachfogende Abbildung zeigt an einem Beispiel den Fluß der 9P-Nachrichten zwischen dem File-Server und der lokalen Maschine. In einer Shell soll durch den Befehl *cp $b/x /tmp* die Datei *x* von dem Verzeichnis *$b* in das Verzeichnis */tmp* kopiert werden. Das Verzeichnis *$b* und damit die Datei *x* wird vom File-Server verwaltet und das Verzeichnis *tmp* von einem Server namens *ramfs*. Ramfs arbeitet als RAM-Disk und stellt seinen Service normalerweise im Verzeichnis */tmp* bereit. Dies geschieht durch einen *mount*(2)-Aufruf (Näheres dazu später). Alle Dateioperationen, die nach dem Start von ramfs unterhalb von */tmp* stattfinden, werden nun von ramfs im Speicher des Rechners durchgeführt statt auf der Platte des File-Servers.



Die Pointe liegt nicht so sehr in der Gestaltung der Nachrichten, sondern in der Art, wie darauf aufbauend ein Großteil des Systems strukturiert wird. Lokal sind die Nachrichten Systemaufrufe im Kern, das heißt, sie werden praktisch als Funktionsaufrufe abgewickelt. *mount* sorgt dafür, daß diese Aufrufe als Nachrichten über eine sichere Verbindung aus dem Kern zu einem lokalen oder fremden Benutzerprozeß geschickt und die Antwort-Nachrichten als Funktionsresultate zurückgegeben werden.

Sehr entscheidend ist bei dieser Lösung, daß ein lokaler Prozeß mit Systemaufrufen über den *mount*-Treiber direkt mit einem fremden Server kommuniziert. Der fremde Server sieht exakt die Systemaufrufe, die sein Client verwendet, und kann entsprechend reagieren. Ist eine Verbindung aufgebaut, kann man mit *9P* auf Dateisysteme, Geräte und vieles andere in einem fremden System zugreifen, als ob es lokale Ressourcen wären. Dieser Mechanismus ersetzt Dinge wie *rlogin*, wo auf dem fremden System eine neue Datei-Sicht entsteht, oder NFS, wo fremde Dateien lokal verwendet werden müssen.

1.8 Spezielle Dateisysteme

Daß ein File-Server seine Dateien als Dateisystem anbietet, ist ja nicht überraschend, aber bei Plan 9 gibt es darüber hinaus eine Vielzahl von konventionellen Ressourcen eines Betriebssystems, die sich als Dateisysteme verkleiden und folglich lokal wie im Netz mit *9P* angesprochen werden können. Eine serielle Schnittstelle besteht zum Beispiel aus zwei Dateien *data* und *ctl*. An *data* schickt man *read* und *write*, um Daten zu transferieren, an *ctl* schickt man zum Beispiel den Text *b9600* per *write*, wenn man die Baud-Rate einstellen will. Der heißgeliebte UNIX-Aufruf *ioctl* entfällt.

Es gibt viele weitere Beispiele für unkonventionelle Dateisysteme, die komplizierte Portabilitätsprobleme überraschend einfach lösen. Ein Prozeß ist als Katalog

repräsentiert, dessen Name die Prozeßnummer ist. Der Kern enthält einen Server für das Dateisystem aller Prozesse, das normalerweise unter */proc* im Namensraum zu finden ist. Jeder Prozeß ist dort mit seiner Prozeßnummer als Katalog repräsentiert. In dem Katalog gibt es eine Reihe von Dateinamen, über die nicht nur Debugger auf interessante Aspekte zugreifen können. *text* repräsentiert die Datei, aus der der Prozeß gestartet wurde. Ein Debugger sucht dort die Symboltabelle und liest oder schreibt *mem*, den virtuellen Adreßraum. Man kann aber auch *text* als Kommando aufrufen, um eine Kopie des Prozesses zu starten. Liest man *stat*, so erhält man den Prozeßzustand als eine Folge von Texten mit konstanter Länge — eine vollständig portable Version von *ps* kann sich darauf beschränken, diese Texte je nach Optionen zu formatieren. *kill* ist ein Skript für die neue Shell *rc*:

```
% kill 8½ | rc
```

kill sucht in den *stat*-Dateien nach dem gewünschten Kommandonamen und erzeugt dann eine Folge von *echo*-Aufrufen, mit denen der Text *halt* in die *note*-Datei der betroffenen Prozesse geschrieben wird und auf den Prozeß dann als Signal wirkt. Man kann sich die Ausgabe von *kill* im Window ansehen und von dort per *cut-and-paste* ausführen lassen, oder man schickt sie direkt per Pipe an *rc*.

Das Beispiel illustriert noch einen kleinen, aber sehr signifikanten Aspekt: Plan 9 arbeitet mit *unicode*, einem neuen 16-Bit Zeichensatz, und kann deshalb nahezu beliebige Zeichen, zum Beispiel in Dateinamen, verkraften. Bearbeitet man UNIX- oder gar DOS-Dateisysteme unter Plan 9, so kann das zu interessanten Resultaten im Bereich Dateibenennung, Tiefe des Dateisystems etc. führen.

Auch das Environment wird als Dateisystem angeboten. Die Anweisung

```
% bind -c '#e' /env
```

bewirkt, daß dieses Dateisystem im Namensraum unterhalb von */env* angeboten wird. Jede Variable ist dort als Datei vertreten, Funktionen haben besondere Namen. Da verwandte Prozesse einen Namensraum gemeinsam benutzen können, kann ein Prozeß das Environment eines anderen modifizieren.

1.9 Heterogenität

Plan 9 ist als heterogenes Netz konzipiert, in dem Architekturen wie MIPS, SPARC, i386, 68020 und andere betrieben werden. Für jede Zielarchitektur gibt es einen C-Compiler und einen Lader, und diese Programme können auf allen Architekturen betrieben werden.

Heterogenität wirft normalerweise eine Reihe von Problemen auf, die in Plan 9 sehr elegant vermieden werden. Binäre Schnittstellen wie *ioctl* werden in Plan 9 durch die Idee der Dateisysteme als Geräte und durch die Übergabe von Text an *ctl*-Dateien nahezu vollständig ersetzt. Bei einem Gerät wie *bitblt* ist binäre Information aus Effizienzgründen notwendig, aber sie wird in einer portablen Grafik-Bibliothek verborgen.

Beim Systemstart ist eine Environment-Variable *$cputype* vordefiniert, mit deren Hilfe das *init*-Programm in einem Katalog wie */386* gefunden wird. In */lib/namespace* steht die Anweisung

```
bind /$cputype/bin /bin
```

und damit enthält der Namensraum die zuständigen binären Programme im üblichen Katalog. Der Benutzer kann den Namensraum dann analog um seine eigenen Programme für die richtige Architektur erweitern.

Übersetzungen werden mit *mk* abgewickelt, einer Neuauflage von *make*. Hierbei spielt die Variable *$objtype* die entscheidende Rolle, denn *mk* liest seine impliziten Regeln aus einer Datei */$objtype/mkfile*. Die Regeln enthalten dann die richtigen Compiler- und Lader-Namen für die durch *$objtype* ausgewählte Architektur. Normalerweise ist das *$cputype*, aber man kann den Wert jederzeit ändern und damit cross-compilieren.

Damit man wirklich kreuz und quer übersetzen kann, gibt es für jede Architektur ein charakteristisches Zeichen, mit dem die verschiedenen Namen und die Kennungen der Objekte gebildet werden: *8c* und *8l* sind Compiler und Lader für i386, ein Montageobjekt heißt *hello.8*, und das übersetzte Programm ist *8.out*. Wenn sich Plan 9 vollständig durchsetzt, dürften dank *unicode* die möglichen Zeichen auch noch lange nicht ausgehen!

1.10 Zusammenfassung

Neue Betriebssysteme setzen sich nur sehr langsam durch. In den Manuals aus dem Jahre 1973 zur *Third Edition* von Unix wurde geschrieben: *Finally, the number of Unix installations has grown to 16, with more expected.*. 22 Jahre später kann man feststellen, daß die Voraussage sich als richtig erwiesen hat.

Man muß kein großer Prophet sein, um festzustellen, daß sich die aktuelle Plan 9 Edition genauso wenig durchsetzen wird, wie sich die dritte Edition von Unix durchgesetzt hat. Betrachtet man allerdings die grundlegenden Konzepte und Ideen, auf denen die Realisierung von Plan 9 beruht, ist es schon schwieriger zu beurteilen, was die Zukunft bringen wird.

2
Flinke Winke

Die Nutzung von Plan 9 steht im Mittelpunkt dieses Kapitels. Es zeigt in Zeitraffer die Funktionalität und die Nutzung von Plan 9. Spätere Kapitel beleuchten einige der vorgestellten Punkte noch wesentlich genauer. Zur Beschreibung des Systems werden »normale« Kenntnisse und Begriffe aus der Unix-Welt verwendet.

Plan 9 wurde von Entwicklern entworfen und realisiert, denen Funktionalität und Eleganz alles, graphischer Luxus nichts bedeutet. Die Konsequenz in puncto Graphik offenbart sich direkt nach dem Start von 8½, dem Fenstersystem von Plan 9. Das Notwendigste, was ein Fenstersystem bieten muß, ist realisiert — Nützliches wie z.B. *drag & drop* von »hier nach dort« sucht man allerdings vergebens, auch kann man die Graphikumgebung nicht als »graphisch ansprechend« bezeichnen, wenn man im Vergleich hierzu NeXTStep oder *Windows* heranzieht. Allerdings besitzen einige Tools, wie *acme*, überraschende Fähigkeiten, was sich letztlich in schnellerer Produktion von Software auszahlt.

Die Eleganz, mit der man arbeiten kann, offenbart sich nicht ganz so schnell. Auf den ersten Blick erweckt Plan 9 glaubhaft den Eindruck, es sei eine Spielart von Unix. Es gilt immer noch:

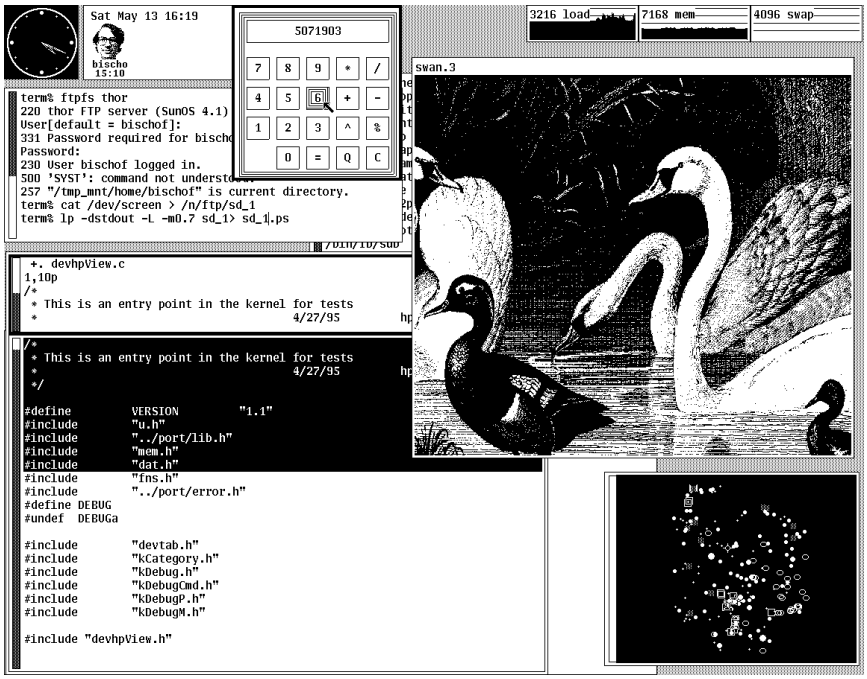
- Dateinamen sind Pfade,
- Komponenten sind durch »/« voneinander getrennt,
- altbekannte Kommandos wie z.B. *tar*, *cp* und *mv* sind noch nutzbar, aber mit zum Teil leicht modifizierter Funktionalität,
- nützliche Werkzeuge wie *yacc, lex, C-Compiler, awk ...* sind erhalten geblieben,
- die Dokumentation teilt sich auf in 564 Seiten Manuals und 462 Seiten Dokumentation. Die Dokumentation ist ohne Schnörkel — kurz und knapp. Man hat also durchaus die Möglichkeit, alles zu lesen. Andere Dokumentationen zu Werkzeugen oder Sprachen lassen dies nicht immer zu; z.B. wiegt die Java-Serie von Sun ca. 2kg.

Auch andere, sehr erfreuliche Dinge wurden beibehalten. Wie unter Unix kann man ebenso unter Plan 9 davon ausgehen, daß es für jedes Problem eine adäquate Lösung gibt. Auf eine gute Lösung kommt man unter Unix i.a.R. nur dann, wenn man mit dem Konzept und nicht gegen das Konzept des Systems arbeitet. Die gleiche Aussage gilt natürlich für das Arbeiten mit Plan 9. Ein Beispiel: Hat man das Problem, daß viele Dateien den gleichen Inhalt haben, der sich bei einer Modifikation gleichzeitig überall ändert, würde man wahrscheinlich ein Original und viele symbolische Links verwenden, die alle auf das Original zeigen. Hat man sich diese Lösung auch für Plan 9 ausgedacht, dann wird man feststellen, daß symbolische Links nicht existent sind.

An Kleinigkeiten offenbart sich, daß das System grundlegend von Unix verschieden ist. Unter anderem wurden folgende Ideen bzw. Kommandos eliminiert:

- die Idee des *super user*,
- *find* gibt es nicht mehr, weil das logische Dateisystem anders zusammengesetzt wird,
- *ln* wurde durch eine neuere Technik im Bereich Dateisystem-Umbau ersetzt
- das Verschieben von Katalogen mittels *mv*(1) ist nicht mehr möglich, weil die Struktur des Dateisystems dieses nicht mehr zuläßt,
- *termcap* und somit auch der von vielen geliebte oder gehaßte *vi*,
- die Idee der Terminalgruppen, Sessions und Prozeßgruppen — somit wurde eines der am schwierigsten zu verstehenden Komplexe aus der Unix-Welt eliminiert,
- *ioctl(2)* — das dunkle Kapitel aus alten Unix-Zeiten,
- ...

Eine typische Sitzung ist in der nachfolgenden Abbildung dargestellt. Normalerweise wird Plan 9 mit dem Window-System 8½ benutzt. Verschiedene Applikationen erhalten dort ihre eigenen Fenster.



Das Skript *$home/lib/profile* steuert, welche Programme aufgerufen werden. In unserem Fall stellt sich der Bildschirm nach dem Anmelden wie folgt dar: Links oben befindet sich eine Uhr ebenso eine Applikation, die neu ankommende Post in Verbindung mit dem Gesicht des Absenders anzeigt. Die Gesamtbelastung des Rechners, die Auslastung des Hauptspeichers und die *swap*-Aktivitäten werden rechts

oben in der Ecke angezeigt. Im linken oberen Fenster wurde eine FTP-Verbindung aufgebaut, der Bildschirmabzug gezogen und das Resultat in PostScript umgewandelt. Der Standard-Editor *sam(1)* ist in der linken unteren Ecke zu sehen. Eine Zeichnung mit drei Schwänen zeigt das mittlere Fenster auf der rechten Seite. Im rechten unteren Fenster sind die mit dem bloßen Auge zu sehenden Sterne des Sternbilds Orion dargestellt.

Der Bildschirmabzug zeigt Nützliches, Skuriles und Interessantes in einer, vom grapischen Standpunkt aus betrachtet, durchaus noch zu verbessernden Qualität — die Funktionalität entspricht dem heutigen Standard.

2.1 Kommandointerpreter

Tom Duff's *rc* ist der Standard-Kommandointerpreter für Plan 9, welcher ähnliche Fähigkeiten wie Steven Bourne's *sh* hat. Allerdings liest *rc* im Gegensatz zu *sh* die Kommandozeile genau einmal, womit die Frage nach der Anzahl »\« doch etwas einfacher zu beantworten ist. Nutzern von Unix-Kommandointerpretern wird der Umstieg nicht schwerfallen. Die interaktive Nutzung funktioniert wie erwartet (das Prompt ist durch ein %-Zeichen dargestellt):

`% date`	Ausgabe des Datums
`% cat dat.1 dat.2 >> dat`	Inhalt von *dat.1* und *dat.2* an *dat* anfügen
`% echo *.[ch] | wc -l`	Anzahl C-Quellen und Include-Dateien im aktuellen Katalog zählen
`% rm -rf ex || echo rm failed`	*ex* löschen; falls dies nicht erfolgreich durchgeführt werden kann, wird eine Meldung ausgegeben
`% p file`	Datei mit Plan 9s *more* betrachten
`% stop rm | rc`	*rm(1)* anhalten (^Z)
`% start rm | rc`	*rm(1)* weiter laufen lassen
`% kill 8½ | rc`	das Fenstersystem mit *kill* abbrechen

Auch die gewohnten Kontrollstrukturen finden sich in *rc* wieder. Etwas ungewöhnlicher ist dann schon die Idee, daß Variablen Listen aus Zeichenketten enthalten, die komponentenweise oder distributiv mit dem Operator ^ verkettet werden können. Beispiele für distributive Verkettungen:

```
% src=(main io jobs controll)   # die Quellen
% echo $src ^ .c                # uebersetzen (echo == cc)
main.c io.c jobs.c controll.c
% echo $src ^ .k                # binden (echo == ld)
main.k io.k jobs.k controll.k
% rm $src ^ .k                  # Objekte loeschen
```

Tom Duff hat sich auch einiges im Bereich Umlenkung von Datenströmen, oder exakter formuliert von Filedeskriptoren, ausgedacht. Ein Beispiel: Der Ablauf der Pipeline-Kette

```
% cmd1 |[5=19] cmd2
```

erzeugt eine Pipeline, in der der Filedeskriptor 5 von *cmd1* mit dem Filedeskriptor 19 von *cmd2* verbunden wird. *Here-Is*-Dokumente können ohne Schnörkel an beliebigen Filedeskriptoren eingelesen werden:

```
#!/bin/rc
 ...
 cmd <<[4]END
        Hello World!
        Hello World!
        END
 ...
```

In diesem Fall erhält *cmd* seine Eingabe vom Filedeskriptor 4, wobei sich die Eingabe auch im Skript befindet.

2.2 Enge Kontakte

Mit Plan 9 wurde das durch die Entwicklung von Unix propagierte Prinzip *kiss* (*keep it simple, stupid*) sehr erfolgreich in die Tat umgesetzt. Alle Operationen, die mit dem Betriebssystem interagieren, sind letzlich Dateisystemoperationen. Diese Art der Funktionalität setzt natürlich voraus, daß die benötigten Einstiegspunkte im Dateisystem verankert sind. Ein Beispiel: Die Prozeßtabelle ist im Katalog */proc* als zweistufiger Baum abgebildet. *proc* enthält für jeden Prozeß einen Katalog, der als Namen die Prozeß-Id trägt:

```
acid
acid
acid lc /proc
1    1135 1305 1578 1584 2    3    42   690  8
10   12   151  1579 1585 23   30   5    7    910
11   1291 1577 1583 17   25   35   51   72
acid lc /proc/25
ctl     note    notepg  segment text
mem     noteid  proc    status  wait
acid cat /proc/25/status
8½                          bischof                          Read
                 240        3760      992568           0
     0           0          244 acid
```

cntl ist die Datei, durch die ein Prozeß kontrolliert wird. *echo kill > /proc/pid/cntl* ist im wesentlichen äquivalent zu *kill pid* unter Unix. Der Kanal, der zur Prozeßkommunikation verwendet werden kann, ist sehr flexibel und einfach geworden, weil an einen Prozeß durch *echo 'text' > /proc/pid/note* zu jedem Zeitpunkt beliebige Information geschickt werden kann. Bei dem Prozeß kommt die Nachricht ähnlich an wie ein Signal unter Unix.

Das Ausstülpen der Prozesse in Form eines Dateisystems ist auch unter anderen Unix-Betriebssystemen realisiert wie z.B. unter *Linux*. Zwar kann in diesen Umgebungen keine Prozeßkontrolle stattfinden, ermöglicht aber immerhin ein portables *ps*(1).

Unter Plan 9 wird alles, durch entsprechende Server, im Dateisystem in Form von Dateien zur Verfügung gestellt. Ein Plan 9-Gerätetreiber, auch Kernel-Server genannt, stellt seine Daten auch in Form eines Dateibaums zur Verfügung. Die Namen der Kernel-Server, unter denen sie angesprochen werden, setzt sich aus »#« und einem Buchstaben zusammen. So ist zum Beispiel der Name für den IP-Gerätetreiber »#I«. Mit

```
% bind '#I' /tmp/IP
```

wird der Namensraum in */tmp/IP* um das Dateisystem des IP-Gerätetreibers erweitert.

Aktive Netzwerkverbindungen werden im Standardfall in Unterkatalogen von */net* eingebunden. In der Datei *status* findet sich eine textuelle Beschreibung des Zustands der Verbindung. Der Buchstabe »I«, welcher in der *ls -l* Ausgabe zu finden ist, verweist auf den Server, der die Information zur Verfügung stellt. »I« ist der Verweis auf den IP-Kernel-Server. Die Dateien können mit *lc* mehrspaltig angeordnet werden.

```
% cd /net/tcp; lc
0     1     2     3     4     5     clone
% lc 0
ctl    data   listen local  remote status
tcp ls —l 0
——rw—rw——— I 0 bischof bischof 0 Apr  5  1995 0/ctl
——rw—rw——— I 0 bischof bischof 0 Apr  5  1995 0/data
——rw—rw——— I 0 bischof bischof 0 Apr  5  1995 0/listen
——r——r——r—— I 0 bischof bischof 0 Apr  5  1995 0/local
——r——r——r—— I 0 bischof bischof 0 Apr  5  1995 0/remote
——r——r——r—— I 0 bischof bischof 0 Apr  5  1995 0/status
% cat 0/status
%/0 1 Established connect ftpfs 0+0
```

Eine Verbindung ist ein Katalog, der eine Reihe von Dateien zur Kontrolle und Darstellung erhält.

Environment-Variablen sind in */env* abgelegt. Die Inhalte der Dateien entsprechen ihrem Wert. Im Falle von Shellfunktionen, deren Namen sich aus *fn#* und dem Namen der Shellfunktion zusammensetzt, ist der Inhalt der Datei die Funktionsdefinition.

```
% lc /env
*           aa        cputype     fn#sigexit  objtype     service
0           apid      dir         fn#%        path        site
BK_ROOT     ba        fn#cd       home        prompt      status
NPROC       c         fn#do_cd    ifs         ps1         suffixes
O9_ROOT     cflag     fn#ll       ipaddr      ps2         swap
a           cpu       fn#pbd      names       rcname      sysname
% cat /env/'fn#ll'
fn ll {ls —l $*}
```

Kernel- wie auch User-Server stellen ihre Daten bereit, die dann mittels *mount(1,2)* in das Verzeichnis montiert werden können. Es ist offensichtlich, daß damit sehr viele Programme, wie z.B. *pstat, dmesg* - oder *ps* portabel, implementiert werden können, weil man als Entwickler nur mit ASCII-Daten zu tun hat.

2.3 Entwicklungsumgebung

Unter Plan 9 stehen die von Unix bekannten Entwicklungswerkzeuge *yacc, lex, mk* — ein *makefile*-Ersatz —, *alef* — geeignet zum Schreiben verteilter Programme —, *acid* — ein programmierbarer Quellcode-Debugger —, C++, ... und natürlich C zur Verfügung. Allerdings unterscheidet sich der C-Dialekt von ANSI C im Detail. Plan 9s C enthält den Kern von ANSI C garniert mit wenigen marginalen Erweiterungen — z.B. namenlose Strukturkomponenten —, einen sehr stark vereinfachten Präprozessor und eine kleine Bibliothek, die Systemaufrufe und nützliche Funktionen enthält. Die Struktur und der Mechanismus, wie *include*-Dateien verwendet werden, sind allerdings komplett modifiziert worden.

Der C++-Compiler übersetzt C++-Quellen unter Nutzung des ANSI C-Compilers und des C++ zu C Übersetzers *cfront* 3.0.1. Dies wurde allerdings nur für die SPARC- und MIPS-Architektur realisiert.

Phil Winterbottoms *alef* ist eine Programmiersprache für verteilte Anwendungen. Als Kommunikations- und Sychronisationsmechanismen werden *channels* verwendet. *Alef* ermöglicht eine Umsetzung der objektorientierten Programmierparadigma. Die Syntax und Semantik von *alef*-Programmen sind der von C-Programmen sehr ähnlich, so daß der Lernaufwand sehr gering ist.

Wenn ein Programm in einer ANSI/POSIX-Umgebung übersetzt wird, gibt's hierfür ein Paket von Howard Trickey namens *ape. Ape* bietet eine Anzahl von *header*-Dateien und Bibliotheken an, die dem ANSI-Standard (ANSIX3.159-1989) sowie der POSIX-Definition IEEE1003.1-1990 , ISO9945-1 entsprechen. Des weiteren findet sich unter *ape/sh* ein Kommandointerpreter, dessen Aufruf dafür sorgt, daß man »Standardwerkzeuge« wie *dirname, make* etc. auf dem Suchpfad zur Verfügung hat. In Plan 9 nicht nachbildbare Funktionalitäten wie *ln*(2) sind als Platzhalter zwar existent, liefern aber immer einen Fehler als Resultat.

Rob Pikes *acme* ist das Werkzeug in Plan 9. Es ist vergleichbar mit dem *vi* unter Unix. Es ist *das* Werkzeug, das verwendet werden muß, damit man effizient Programme schreiben kann.

2.4 Fenstersystem

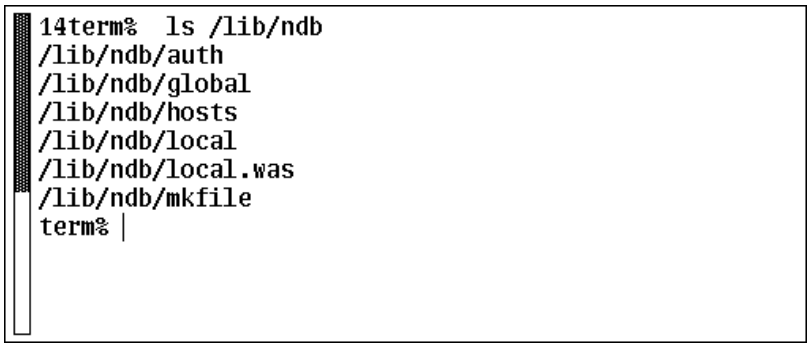
Rob Pikes 8½ ist das Fenstersystem von Plan 9. Es stellt Text-Ein- und Ausgabe und Bitmapgraphik für das lokale System und für fremde Systeme zur Verfügung. Aus Sicht des Nutzers ist 8½ im Vergleich zu NeXTStep oder X als sehr spartanisch zu bezeichnen. Aus Sicht des Entwicklers ist 8½ äußerst faszinierend.

Im Gegensatz zu UNIX-Systemen gibt es in den Fenstern von 8½ keine Cursor-Adressierung. Mit Hilfe von *cut and paste* ist es möglich, Text innerhalb eines Fensters zu bearbeiten, aber auch Informationen zwischen den Fenstern auszutauschen.

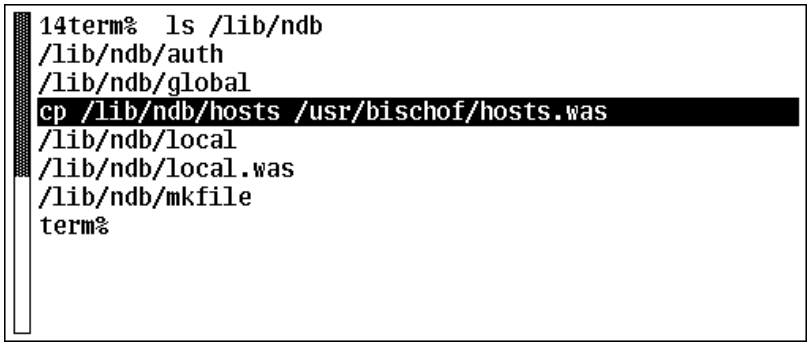
Wird im UNIX-Bereich eine neue Applikation gestartet, so ist es üblich, daß dies in einem neuen Fenster geschieht. Gewöhnlich benutzt bei 8½ eine neue Applikation das Fenster, in dem es gestartet worden ist, was aber nicht zwingend ist. Ein Prozeß kann durchaus eigene, neue Fenster erzeugen.

Der *window manager* ist ein fester Bestandteil von 8½. Die Nutzung von 8½ kann nur mit einer Maus mit drei Knöpfen erfolgen. Mit dem linken Mausknopf werden Text-Bereiche selektiert, die man ausschneiden oder an anderer Stelle einfügen kann. Der mittlere Mausknopf dient zur Manipulation des selektierten Bereichs — *cut, paste, snarf, send* — und zum kontinuierlichen Verschieben des Schiebereglers. Wird die linke bzw. rechte Taste auf dem Schieberegler gedrückt, bewirkt dies ein Springen des Fensterinhalts nach oben bzw. unten. Die gedrückte rechte Maustaste offeriert Fenstermanipulationsmöglichkeiten wie *New, Reshape, Move, Hide, UnHide*.

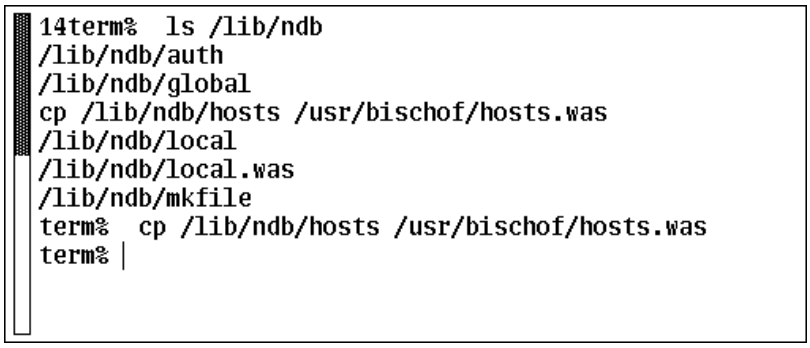
Im Gegensatz zu anderen Fenstersystemen ist der komplette Text eines Fensters, in dem *rc* oder ein anderes Programm läuft, via *cut & paste* modifizierbar. Dies ändert den Arbeitsstil ziemlich, wenn man an einen Kommandoprozessor mit History-Mechanismus gewöhnt ist, den *rc* nicht besitzt. Normalerweise werden die Resultate von Kommandos mittels Maus und *cut & paste* aufbereitet und wieder als Kommandos verwendet:



Die Ausgabe wird nun durch Einfügen von Text durch »klicken & tippen« leicht modifiziert. Doppelklick der linken Maustaste am Zeilenende selektiert die komplette Zeile.



Der mittlere Mausknopf läßt den Menüpunkt *send* selektieren, dessen Aktion den selektierten Text an die aktuelle Cursorposition zur Ausgabe schickt und somit in *rc* zur Ausführung bringt.



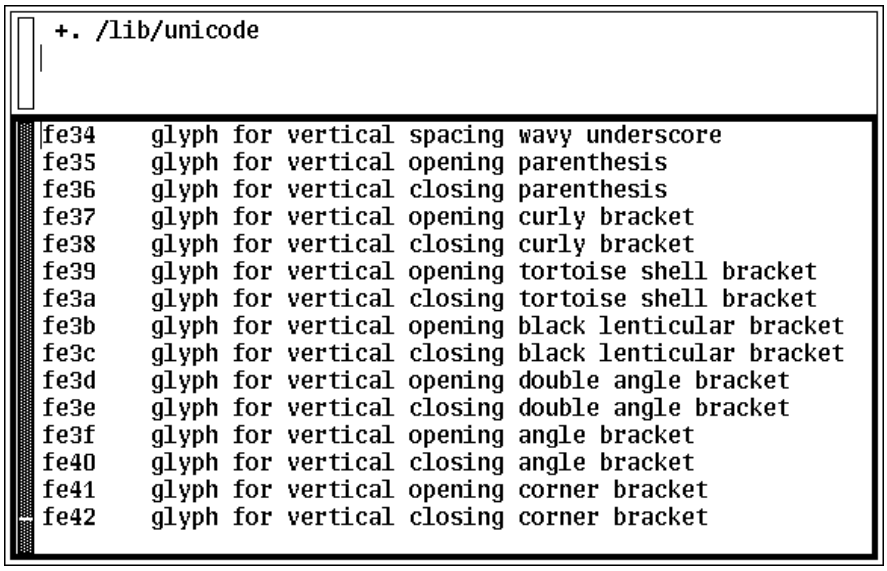
2.5 Unicode

8 Bit für ein Zeichen sind nicht genug, wenn ein Rechner nicht nur in Englisch sondern auch in der Muttersprache anderer Kulturkreise einsetzbar sein soll. Besteht ein Zeichen aus 16 Bits, kann man 65535 Zeichen ($2^{16} - 1$) ansprechen. Diese Anzahl dürfte für die meisten realen Sprachen ausreichen.

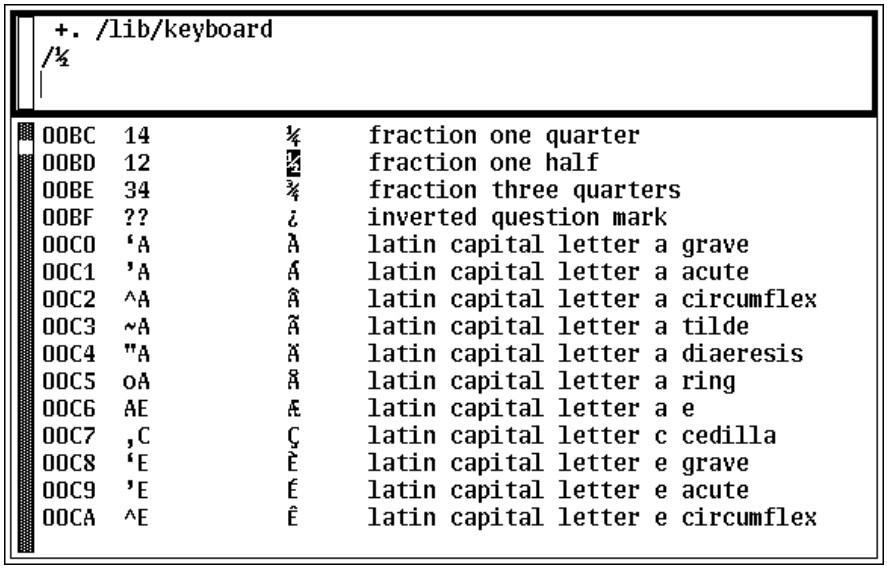
Es ist klar, daß eine Umstellung von 8 auf 16 pro Zeichen, sprich ASCII auf Unicode, siehe auch [Unicode 1991], nicht trivial ist. Es gibt kein uns bekanntes Betriebssystem, das komplett umgestellt worden ist. Nehmen wir für einen kurzen Moment an, Plan 9s Zeichendarstellung sei 16-Bit breit. Die Konsequenz wäre, daß ein Import von Daten anderer Systeme ohne Konvertierung nicht möglich ist, was sich letztlich nicht als sehr praktikabel erweisen würde. Eine Antwort auf dieses Problem lag in der Nutzung von UTF. Dieses *Universal Character Set Transformation Format* stellt Unicode durch ein bis drei Bytes pro Unicode-Zeichen so dar, daß eine ASCII-Datei auch eine UTF-Datei ist. Das Auswerten eines Unicode-Textes ist rückwärts kompatibel, d.h., ASCII-Texte können von Fremdsystemen direkt übernommen werden. Die Ver- bzw. Entschlüsselungvorgehensweise ist in der nachfolgenden Abbildung dargestellt.

1. *c* aus [00000000.0bbbbbbb] → 0bbbbbbb
2. *c* aus [00000bbb.bbbbbbbb] → 110bbbbb, 10bbbbbb
3. *c* aus [bbbbbbbb.bbbbbbbb] → 1110bbbb, 10bbbbbb, 10bbbbbb

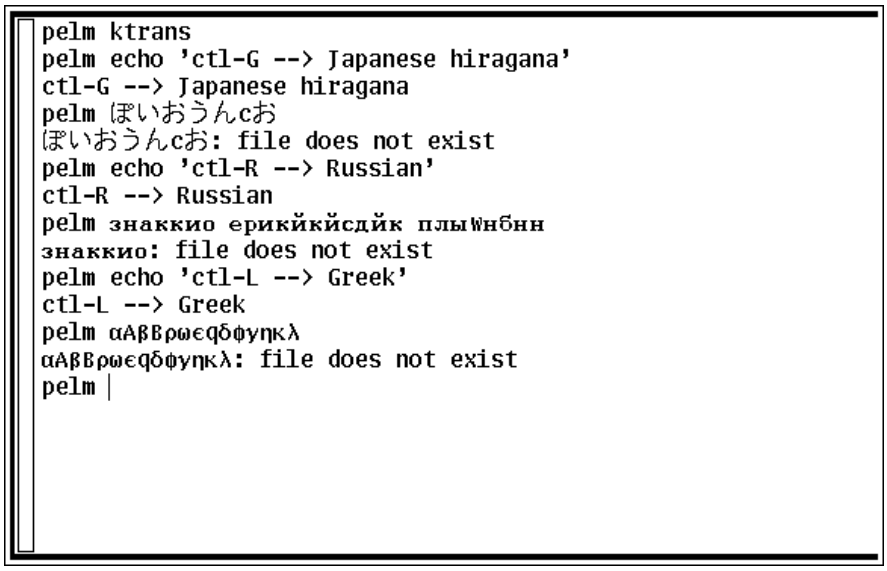
Die Umwandlung 1. garantiert die Kompatibilität zu 8-Bit-basierten Systemen. Die Datei */lib/unicode* enthält die Beschreibung der Zeichen; hier ist ein Ausschnitt:



Normalerweise stellt sich für den Nutzer von Plan 9 allerdings die Frage, was man eintippen muß, um das Fenstersystem 8½ aufrufen zu können. In diesem Fall: *8 COMPOSE 1 2*. Die Datei */lib/keyboard* enthält die Kombination für alle möglichen Zeichen.



Ist man daran interessiert, auf einfache Art und Weise im griechischen Alphabet zu schreiben, ist *ktrans(1)* sehr hilfreich. *ktrans(1)* installiert sich selbst zwischen Tastatur und Konsole und übersetzt die getippten Zeichen in das entsprechende Alphabet.



ktrans(1) benötigt einen entsprechenden Zeichensatz, um diese auch darstellen zu können. In diesem Fall wurde 8½ in dem Fenster der vorigen Abbildung mit dem Font */lib/font/bit/pelm/unicode.8.font* gestartet.

2.6 Heterogenität

Plan 9 ist als heterogenes Netz konzipiert, in dem Architekturen wie MIPS, SPARC, i386, 68020 und andere betrieben werden. Für jede Zielarchitektur gibt es einen C-Compiler und einen Lader, und diese Programme können auf allen Architekturen betrieben werden.

Heterogenität wirft normalerweise eine Reihe von Problemen auf, die in Plan 9 sehr elegant vermieden werden. Binäre Schnittstellen wie *ioctl* werden in Plan 9 durch die Idee der Dateisysteme als Geräte und durch die Übergabe von Text an *ctl*-Dateien nahezu vollständig ersetzt. Bei einem Gerät wie *bitblt* ist binäre Information aus Effizienzgründen notwendig, aber sie wird in einer portablen Grafik-Bibliothek verborgen.

Beim Systemstart ist eine Environment-Variable *$cputype* vordefiniert, mit deren Hilfe das *init*-Programm in einem Katalog wie */386* gefunden wird.

Übersetzungen werden mit *mk* abgewickelt, einer Neuauflage von *make*. Hierbei spielt die Variable *$objtype* die entscheidende Rolle, denn *mk* liest seine impliziten Regeln aus einer Datei */$objtype/mkfile*. Die Regeln enthalten dann die richtigen Compiler- und Lader-Namen für die durch *$objtype* ausgewählte Architektur. Normalerweise ist das *$objtype*, aber man kann den Wert jederzeit ändern und damit cross-compilieren. Damit man wirklich kreuz und quer übersetzen kann, gibt es für jede Architektur ein charakteristisches Zeichen, mit dem die verschiedenen Namen und die Kennungen der Objekte gebildet werden: z.B. *8c* und *8l* sind Compiler und

Lader für i386, ein Montageobjekt heißt *it.8*, und das übersetzte Programm heißt im Standardfall *8.out*.

Mit dem *mkfile*

```
</$objtype/mkfile

t:V:
    for(i in sparc mips 386 68020 ) @{
        objtype=$i mk it
    }

it: it.$O
    $LD —o $O.it it.$O
it.$O:  it.c
    $CC it.c
nuke:
    rm —f [2kv8x].*  *.[2kv8x]
```

kann *it.c* für die Architekturen in *sparc, mips, 386* und *68020* übersetzt werden. Die nachfolgende Abbildung zeigt einen Ablauf.

```
% lc
it.c   mkfile
% mk
for(i in sparc mips 386 68020 ) @{
        objtype=$i mk it
}
kc it.c
kl -o k.it it.k
vc it.c
vl -o v.it it.v
8c it.c
8l -o 8.it it.8
2c it.c
2l -o 2.it it.2
% lc
2.it   it.2   it.c   it.v   mkfile
8.it   it.8   it.k   k.it   v.it
%
```

Natürlich könnte man das Cross-compilieren auch ohne Modifikationen im *mkfile* durchführen.

```
% for ( i in sparc mips 386 68020 )
>       {    objtype=$i mk it }
```

2.7 Textverarbeitung

Werkzeuge zur Produktion von Satztext sind neben den Standardwerkzeugen wie Compiler, Compilergeneratoren etc. heutzutage unabdingbare Werkzeuge, die bereitgestellt werden müssen, um ein Betriebssystem für Entwickler akzeptabel zu

machen. Unter Plan 9 stehen dafür die Textcompiler der *troff*-Familie sowie *tex* in seinen nutzbaren Ausprägungen zur Verfügung.

Alles, was an Dokumentation von den Plan 9 Entwicklern existiert — Manualseiten, *early papers,* ... — ist mit *troff* und co. gesetzt worden. Es gibt *troff*- und *tex*-Betrachter, aber keine WYSIWYG-Werkzeuge zur Dokumentenentwicklung und wie üblich produzieren die Werkzeuge PostScript.

2.8 Plan 9 im Internet

Zu Plan 9 gibt es eine kleine, aber feine *mailing list.* Aufgenommen wird man in diese Liste, indem man an *majordomo@cse.psu.edu* eine *mail* schickt, die folgender Syntax genügen muß: *subscribe [<list>] [<address>]*. Die gewünschte Liste ist in diesem Fall *9fans.* Falls der ganze Vorgang nicht zufriedenstellend funktioniert, kann man mit der Angabe von *help* im Rumpf der *mail* eine Beschreibung der möglichen Kommandos anfordern, die dieser Mail-Server versteht.

Plan 9 ist auch im WWW präsent:

Distribution:	`http://plan9.bell-labs.com/plan9/distrib.html`
Einführung:	`http://plan9.bell-labs.com/plan9/index.html`
FAQ:	`http://plan9.bell-labs.com/plan9/faq.html`

Auf den angebenen Seiten finden sich die wesentlichsten Querverweise zu anderen Servern. Viel Spaß beim »Surfen«.

Die Manual-Seiten, *early papers,* Unix Utilities und *updates* sind im *ftp*-Bereich von AT&T zu finden:

```
ftp plan9.att.com
Connected to plan9.att.com.
220 Plan 9 FTP server ready
Name (plan9.att.com:bischof): ftp
331 Send email address as password
Password:
...
ftp> cd /plan9
250 directory changed to /plan9
ftp> ls -C
200 Data port is tcp!131.173.161.22!2857
150 Opened data connection (tcp!131.173.161.22!2857)
cd.gif              cd.html             cd.listing
copyright.html      distrib.html        doc
errata              errata.html         faq.html
index.html          lucent-b.gif        lucent.gif
man                 man1.html           man2.html
man3.html           man4.html           man5.html
man6.html           man7.html           man8.html
man9.html           mansearch.html      mothra.gif
mothracompat.gif    mothraenhanced.gif  pcdist
readme              shrink.html         sky.jpg
skysmall.gif        unixsrc             update
vol1.gif            vol1.html           vol2.gif
vol2.html
```

2.9 Verschiedenes

Dieser Abschnitt beschäftigt sich mit einem Sammelsurium von Nützlichem oder zumindest von Interessantem. Ein Reihenfolge oder funktionale Zuordnung zu Problembereichen ist in diesem Abschnitt nicht zu finden.

½char(1), *char*(1) und *rchar* ermöglichen ein interaktives Betrachten des Unicode-Standards. Es ist erstaunlich festzustellen, wie viele Zeichen es gibt und wie schwierig manche Programme zu nutzen sind.

cpu(1) ermöglicht die Nutzung der Hardware-Ressourcen eines CPU-Servers. *ftpfs*(1) sorgt dafür, daß ein Dateisystem, das von einem *ftp*-Server zur Verfügung gestellt wird, im lokalen Namensraum sichtbar wird und somit mit den Standardkommandos modifiziert werden kann.

```
% ftpfs thor
220 thor FTP server (SunOS 4.1) ready.
User[default = bischof]:
331 Password required for bischof.
Password:
230 User bischof logged in.
500 'SYST': command not understood.
257 "/tmp_mnt/home/bischof" is current directory.
% ls -l /n/ftp/.bashrc
--rw-r--r-- M 32 bischof inf 224 Jun 27 13:38 /n/ftp/.bashrc
```

get und *put* werden zu *cp*(1) wobei allerdings die Daten über */tmp* kopiert werden. Da in aller Regel der Hauptspeicher zum Arbeiten nicht ausreicht, sollte man eine *swap*-Datei mit *swap*(1) anlegen. Dies erfolgt normalerweise in der eigenen *lib/profile*.

doctype(1) versucht zu erraten, mit welchen Werkzeugen ein Dokument gesetzt werden muß. Die typische Verwendung sieht wie folgt aus:

```
% doctype c.01
grap c.01 | pic  | tbl  | eqn   | troff -ms
% eval `{ doctype c.01 } | lp
```

Das ist wieder eines der kleinen, aber nützlichen Werkzeuge. Eine Formatierung aller Manualseiten ohne Aktivierung unnötiger Präprozessoren wird somit zur Fingerübung:

```
#!/bin/rc

MAN_PS=$home/man
MAN_ORIG=/sys/man

for ( dirs in $MAN_ORIG/? ) {   # ueber alle Manual-Kataloge
        cd $dirs                # von diesem Kapitel
        dir=`{ basename $dirs}  # Kapitelnummer beschaffen
        for  ( f in * ) {       # alle Dateien des Kapitels
                echo $f         # Debug Information
                eval `{ doctype $f } > $MAN_PS/$dir/$f
                }
        cd ..
        }
```

page(1) und *proof*(1) sind die Betrachter für ankommende Faxe, Bitmap- und PostScript-Dateien. Auch Plan 9 kommt ohne das Web nicht aus. *mothra*(1) ist der Plan 9 Web Browser. Das Editieren von *subfont,* Bitmap- oder *face*-Dateien erledigt man mit *tweak.*

3
Die ersten Schritte

Plan 9 wird in Buchform mit oder ohne Software vertrieben: *Plan 9 The Manuals* und *Plan 9 The Documents* im Verlag Harcourt Brace & Company; ISBN 0-03-017143-1 für das komplette System und 0-03-017142-3 für die Handbücher allein. Vom Internet kann man sich neuere Installationsdisketten mit einem kleineren, auf PC lauffähigen System holen: *http://plan9.bell-labs.com/plan9/distrib.html*. Wie man dort nachlesen kann, befinden sich auf der CDROM auch die Quellen des Systems sowie Binärdateien für alle unterstützten Architekturen wie Sparc, MIPS und andere. Die Installationsdisketten enthalten immerhin viele der üblichen Kommandos von *bind* über *ls* bis *unmount* und insbesondere für Reparaturen den Editor *ed* und die Shell *rc* sowie zur mehrspaltigen Ausgabe *mc*. Außerdem sind das Window-System *8½*, der Editor *sam* und das C-Entwicklungssystem für die PC-Architektur inklusive *mk* vorhanden.

Dieses Kapitel beschreibt im Detail, wie man mit den Installationsdisketten zu einem lauffähigen Plan 9-System auf einem PC kommt und dabei mit typischen Fehlern fertig wird. Nebenbei sieht man schon, was beim Start von Plan 9 passiert; dies wird im nächsten Kapitel vertieft.

3.1 Die Vorbereitungen

Für eine erste Installation von Plan 9 beschafft man sich am besten einen einfachen, mittelgroßen PC, möglichst mit einer IDE-Platte und einer Logitech-Maus. Über Alternativen kann man bei der obigen Adresse (*The Various Ports* und *Installing the Plan 9 Distribution*) oder in *Plan 9 The Documents* nachlesen, oder man probiert einfach die nachfolgend beschriebene Installation aus. Außerdem benötigt man vier 3½-Zoll-Disketten, auf die man unter UNIX vier Image-Dateien *images/disk[1-4]* oder unter DOS vier Bäume *trees\disk[1-4]* von unserer CDROM kopiert. Es handelt sich dabei um die neueren Installationdisketten vom Netz; mit dem eingangs erwähnten Paket wurden ältere Disketten ausgeliefert, die durchaus einige Partitionen auf der PC-Festplatte zerstören.

Auf einem UNIX-System verwendet man zum Kopieren der Disketten einen Befehl wie

```
# dd if=images/disk1 of=/dev/rfd0a bs=36k conv=sync
```

wobei *images/disk1* auf das Image der Diskette auf der CDROM und */dev/rfd0a* auf die Gerätedatei für das Diskettenlaufwerk verweisen sollten.

Auf einem DOS-System kann man entweder *rawrite* benutzen, um ebenfalls die Images zu kopieren, oder man transferiert die Dateibäume mit Befehlen wie

```
c> xcopy trees\disk1 a:\ /e/v
```

wobei *trees\disk1* auf den Dateibaum der Diskette auf der CDROM und *a:* auf das Diskettenlaufwerk verweisen sollten.

rawrite.exe stammt aus der Slackware Linux-Ausgabe und befindet sich auf unserer CDROM. Es wird ohne Argumente aufgerufen und fragt dann nach der Quelldatei und dem Ziellaufwerk. Entscheidend ist, daß von *trees* in DOS-Dateisysteme oder von *images* ohne Dateisystem direkt auf die Diskette kopiert wird; im letzteren Fall ist die erste Diskette auch boot-fähig.

3.2 Die Festplatte

Plan 9 wird im folgenden auf eine Festplatte auf dem PC installiert, und zwar *hinter* alle dort eingerichteten Partitionen. Die Festplatte muß für Installation und Start von Plan 9 eine kleine DOS-Partition enthalten; dafür genügen weniger als 5 MB. Unter DOS muß man nur ein Kommando ausführen können; während der Einrichtung sind *edit* und *fdisk* hilfreich.

Plan 9 beschreibt seine eigenen Partitionen im physikalisch letzten (oder bei IDE-Platten vorletzten) Block der Festplatte und nicht in der Partitionstabelle, die Systeme wie DOS oder Linux im sogenannten *Master Boot Record* unterhalten. Das bedeutet, daß zur Installation von Plan 9 ein Bereich am Ende der Festplatte nicht durch ''normale'' Partitionen belegt sein darf, die mit Programmen wie *fdisk* unter DOS verwaltet werden. Auch die erzeugten Plan 9-Partitionen leben in diesem Bereich. Sollte die ganze Festplatte belegt sein, kann man unter Umständen mit dem Programm *fips* von unserer CDROM die letzte Partition verkürzen; auch dieses DOS-Werkzeug stammt aus der Slackware Linux-Ausgabe.

Plan 9 richtet während der Installation mehrere eigene Partitionen ein. Bei der ersten IDE-Platte beschreibt */dev/hd0disk* die ganze Platte, */dev/hd0partition* den letzten Block, in dem Plan 9 diese Partitionen beschreibt, */dev/hd0fs* das Wurzel-Dateisystem und */dev/hd0boot* einen Bereich mit dem Systemkern, der normalerweise gestartet wird. Man kann sich diese Tabelle in Plan 9 direkt ansehen:

```
term% cat /dev/hd0partition
plan9 partitions
boot 637056 639104
swap 639104 677804
nvram 677804 677805
fs 677805 832606
```

Zur Installation des Disketten-Systems genügen schon etwa 20 MB. Will man von der CDROM ein komplettes, heterogenes Entwicklungssystem aufbauen, sollte man mehr als 600 MB bereitstellen.

3.3 Phase 1

Von Diskette 1 wird entweder durch einen Boot-Vorgang oder von Hand im Wurzelkatalog mit *b.com* ein Plan 9-System geladen und gestartet, in dem *\rc\bin\cpurc* und daher *\386\bin\install* ausgeführt wird.* *install* ist ein Menüsystem, mit dem man hoffentlich ein sehr kleines Plan 9-System in einer existierenden DOS-Partition auf einer Festplatte einrichten kann.

---

* Pfade mit \ beziehen sich auf die unterliegenden DOS-Dateisysteme; unter Plan 9 befinden sich die Objekte an den typischen Stellen im Namensraum.

Zuerst bietet *install* die erkannten Festplatten an. An dieser Stelle kann man auch andere Treiber konfigurieren, um eventuell noch nicht entdeckte Festplatten verfügbar zu machen. Man muß sich schließlich für eine Platte entscheiden, deren Partitionen dann angezeigt werden. Von diesen wählt man eine DOS-Partition aus. Dorthin, in einen Baum unter *\plan9*, kopiert *install* dann den Inhalt der Diskette 1. Zwei Dateien werden überschrieben: *\plan9\lib\namespac* durch *\plan9\lib\namesdos* und *\plan9\rc\bin\cpurc* durch *\plan9\rc\bin\cpurcdos*.

Zum Schluß legt *install* noch eine Datei *\plan9\plan9.ini* an, die das neue System beschreibt. In einigen, kaskadierten Menüs kann man den VGA-Modus, den Anschluß der Maus, eine Netzkarte, weitere Platten- und CDROM-Treiber etc. konfigurieren, die dann in Phase 2 zur Verfügung stehen.

3.4 Phase 2

Phase 2 beginnt damit, daß man das DOS-System startet, in dem in Phase 1 der Baum unter *\plan9* angelegt wurde. Alternativ könnte man natürlich diesen Baum von Diskette 1 (oder unserer CDROM) selbst kopieren, die erwähnten Dateien ersetzen und insbesondere die Datei *\plan9\plan9.ini* unter Beachtung der Manualseite *plan9.ini*(8) mit DOS-Werkzeugen wie *edit* bearbeiten.

Wenn man mit dem Baum *\plan9* und mit *\plan9\plan9.ini* zufrieden ist, startet man Plan 9 vom Wurzelkatalog von DOS aus mit dem Befehl *plan9\b*. Wieder wird *\plan9\rc\bin\cpurc* ausgeführt; dort wird jetzt aber *\plan9\bin\build* aufgerufen.

build ist, wie *install*, ein Menüsystem, das diesmal aber die Einrichtung größerer Plan 9-Systeme von Disketten oder der ''offiziellen'' CDROM in eigenen Partitionen mit Plan 9-Layout zum Ziel hat. Arbeitet man mit den Installationsdisketten, wählt man den ersten Menüpunkt *Install 3 Diskette System to local drive* und läßt nacheinander die restlichen Disketten 2 bis 4 in die Plan 9-Partition *fs* auf die Festplatte übertragen.

Abschließend muß man den letzten Menüpunkt *Make the newly installed Plan 9 the default* wählen, der einen neuen *bootfile*-Wert in die Datei *\plan9\plan9.ini* einträgt, damit bei weiteren Systemstarts ein Systemkern *9pcdisk* aus der Plan 9-Partition *boot* verwendet wird, der nach Voreinstellung die Plan 9-Partition *fs* als primäres Dateisystem verwendet.

Der letzte Menüpunkt schlägt sofort vor, daß man das System neu startet.

3.5 Laden des Systems

Auf einem PC wird ein Plan 9-Systemkern normalerweise mit dem Kommando *b* geladen. Als Argument kann ein *bootfile* angegeben werden, das sich aus Plattenart, Laufwerknummer und einer Angabe für den Kern zusammensetzt:

`fd!0!9dos`	erste DOS-Diskette, Kern in Datei *\9dos*
`hd!0!/plan9/9dos`	erste IDE-Platte, DOS-Partition, Pfad zum Kern
`h!0`	erste IDE-Platte, Kern in Plan 9-Partition *boot*
`sd!0!/9dos`	erste SCSI-Platte, DOS-Partition, Pfad zum Kern
`sd!1`	zweite SCSI-Platte, Kern in Plan 9-Partition *boot*

b kann auch einen Kern vom Netz holen, siehe *b.com*(8). Außerdem sucht *b* nacheinander auf Disketten, IDE- und SCSI-Platten nach einer Datei *\plan9\plan9.ini* oder *\plan9.ini*, die weitere Parameter enthält, siehe *plan9.ini*(8).

Abgesehen von Konfigurationen für Netzkarten und andere Schnittstellen, nach denen beim Start von Plan 9 gesucht werden soll, enthält *plan9.ini* eine Voreinstellung für *bootfile* und als Wert von *rootdir* einen Pfad, bei dem sich die Wurzel eines Plan 9-Dateisystems im DOS-Dateisystem befindet, wenn der Kern *9dos* geladen wird.

Beispielsweise kann man auch nach Abschluß von Phase 2 auf *build* zurückgreifen. Dazu startet man das DOS-System, das *\plan9* aus Phase 1 enthält und das in Phase 2 verwendet wurde. Man erzeugt eine Sicherungskopie der Datei *\plan9\plan9.ini*, und im Original trägt man zusätzlich folgende Zeile ein:

```
rootdir=plan9
```

Anschließend lädt und startet man das gleiche System wie in Phase 2, indem man im Katalog *\plan9* folgendes Kommando eingibt:

```
c> b hd!0!/plan9/9dos
```

Ruft man *b* ohne Argumente auf, gelangt man wieder zum installierten Plan 9-Systemkern *9pcdisk*.

3.6 Systemstart

b lädt einen Systemkern, in dem als erster Prozeß ein Kommando */boot* ausgeführt wird, das im Kernel-Server *#/* selbst gebunden ist. *boot* muß für eine Benutzeridentifikation sorgen und Verbindung zu einem ersten File-Server aufnehmen. Wie im Abschnitt 4.6 näher erklärt wird, startet *boot* dann ein Programm *init*, das nach Anweisungen in der Datei */lib/namespace* den Namensraum ausgestaltet und schließlich den Kommandoprozessor *rc* ausführt, der zunächst ein Skript interpretiert und sich dann an der Konsole meldet.

9pcdisk ist ein Systemkern für ein *Terminal* und führt deshalb am Schluß des Systemstarts */rc/bin/termrc* aus. *9dos* ist ein Systemkern für einen CPU-Server und führt */rc/bin/cpurc* aus. In dieser Datei steht am Schluß *build* (oder auf Diskette 1 *install*) — so wird das Menüsystem zur Ausführung gebracht.

Startet man *9pcdisk*, verlangt *boot* zuerst nach einem File-Server:

```
root is from (local, 9600, 19200, il)[local!#H/hd0fs]:
```

Die Voreinstellung *local* bezieht sich hier auf einen File-Server */fs*, der im Kernel-Server *#/* selbst gebunden ist und von *boot* gestartet wird. Dieser Server ist auf der CDROM als Kommando *disk/kfs* verfügbar und benötigt eine Platten-Partition. Hier stammt sie vom Kernel-Server für IDE-Platten *#H*, und es ist die Partition *fs*, in der in Phase 2 installiert wurde.

Die anderen Angaben beziehen sich auf Netzverbindungen, für die ein ebenfalls im Kernel-Server gebundener Cache-File-Server eingesetzt werden kann, der auch als Kommando *cfs* verfügbar ist und nach Voreinstellung eine Partition *cache* dazu benutzt, Netzzugriffe zu minimieren.

Bleibt man bei der lokalen Platte, fragt *boot* als nächstes nach einem Benutzernamen und bietet *none* als Voreinstellung an. *disk/kfs* ist nicht besonders sicher. Man kann neue Benutzer erzeugen und ihnen Heimatkataloge geben:

```
term% disk/kfscmd 'newuser axel'
add user '2:axel:axel
```

Damit gibt es einen neuen Benutzer *axel* in der Datei */adm/users*, und ihm gehören sein neuer Heimatkatalog */usr/axel* und einige Kataloge darunter. Zwar fragt *boot* für diesen Benutzer nach einem Paßwort, aber man benötigt keins zum Zugriff auf den lokalen File-Server. Paßwörter kommen erst zum Tragen, wenn man ein separates File-Server-System verwendet und werden dann mit einem Authentifizierungs-Server verwaltet.

Mit *kfscmd*(8) und dem Argument *halt* sollte man auch den lokalen File-Server anhalten, bevor man das System durch Tippen von zweimal *ctrl-T* und dann *r* terminiert.

3.7 Handarbeit

Hat man den Systemstart erst einmal verstanden, kann man auch eingreifen. Dazu entfernt man aus der in Phase 1 erzeugten Datei *\plan9\rc\bin\cpurc* die letzte Zeile, in der *build* aufgerufen wird. Man muß dazu unter Unix editieren oder unter DOS einen Editor verwenden, der vor Zeilentrennern keine *return*-Zeichen einfügt, denn das würde unter Plan 9 Syntaxfehler verursachen. Die Datei *\plan9\rc\bin\cpurcdos* dient jedenfalls als Sicherungskopie.

Will man von Hand installieren, sollte man auf die Dateien *disk[234].vd* im Katalog *trees/disk[234]* von unserer CDROM entweder unterhalb von *\plan9* in der DOS-Partition oder auf ein geeignetes (SCSI) CD-Laufwerk zugreifen können. Anschließend startet man das Installationssystem:

```
c> b hd!0!/plan9/9dos
```

Diesmal meldet sich nicht das Menüsystem *build*, sondern *rc* an der Konsole und man ist auf sich selbst gestellt. Es hilft, wenn man jetzt *Plan 9 The Manuals* zur Verfügung hat oder im Internet unter *http://plan9.bell-labs.com/plan9/vol1.html* nachschlagen kann.

Zur manuellen Installation baut man zuerst die Plan 9-Partitionen mit *prep*(8) auf. Für eine IDE-Platte gibt man folgenden Befehl:

```
% disk/prep '#H/hd0'
```

Wenn man nur üben möchte, kann man mit der Option *–r* verhindern, daß eine Partitionstabelle wirklich geschrieben oder zerstört wird.

prep zeigt zunächst die Informationen im Master Boot Record, also die Partitionen, die Systeme wie DOS und Linux verwenden. Anschließend kann man Namen und Bereiche in Form von Blocknummern am Anfang und Ende für Plan 9-Partitionen eingeben, wobei man sich zunächst eine Voreinstellung als *autopartition* vorschlagen lassen kann. Eine Tabelle zeigt jeweils den aktuellen Stand und Überlappungen:

```
% disk/prep '#H/hd0'
Master Boot Record exists
   type        overlap  start   end     %   size
0  FATHUGE     0-           63  205631  24  205569
1 *EMPTY       -1       205632  832606  75  626975

Plan 9 partition table exists
Nr Name           Overlap   Start   End     %   Size
 0 /hd0disk       0123456   0       832607  100 832607
 1 /hd0partition  01-----   832606  832607    0      1
 2 /hd0boot       0-2----   205632  207680    0   2048
 3 /hd0swap       0--3---   207680  332665   15 124985
 4 /hd0nvram      0---4--   332665  332666    0      1
 5 /hd0fs         0----5-   332666  832606   60 499940
 6 /hd0dos        0-----6       63  205632   24 205569
```

Bestätigt man die Anfrage *Ok?*, wird die neue Tabelle geschrieben. Im Beispiel ist zusätzlich eine Partition definiert, über die die DOS-Partition am Anfang der Platte erreichbar ist.

Als nächstes startet man den lokalen File-Server *kfs*(4) und konfiguriert die Partition *fs* mit Blöcken, die jeweils 1024 Bytes enthalten:

```
% disk/kfs -r -b 1024 -f /dev/hd0fs
kfs: reaming the file system using 1024 byte blocks
initializing minimal user table
% disk/kfscmd check
checking file system: main
check free list
lo = 3; hi = 249969
       1 files
  249969 blocks in the file system
       2 used blocks
  249967 free blocks
       1 maximum qid path
```

Wie man mit *kfscmd*(8) sieht, wurde offenbar die richtige Partition verwendet.

```
% mount -c /srv/kfs /n/kfs
% disk/kfscmd allow
% vdexpand </disks/disk2.vd | disk/mkext -d /n/kfs
done
```

mount(1) stellt jetzt das von *kfs* verwaltete Dateisystem als */n/kfs* zur Verfügung. *allow* sorgt dafür, daß *kfs* nicht prüft, ob die gewünschten Zugriffe auch erlaubt sind. *vdexpand* dekomprimiert das Archiv *disk2.vd* von der zweiten Installationsdiskette, das vor dem Start nach *\plan9\disks\disk2.vd* kopiert wurde, und *mkext*(8) baut die Dateien aus dem Archiv wieder im File-System auf. So kann man die restlichen drei Installationsdisketten von Hand einspielen.

```
% disk/kfscmd user
% disk/kfscmd sync
```

user veranlaßt den File-Server *kfs* über *kfscmd*(8), die Datei */adm/users* zu lesen und damit seine Benutzertabelle neu zu initialisieren. Diese Datei wurde aus dem

Archiv auf die Platte geschrieben. Da *kfs* nicht unbedingt alle Informationen auf die Platte schreibt, sorgt *sync* dafür, daß fehlende Blöcke gesichert werden.

```
% cp /n/kfs/386/9pcdisk /dev/hd0boot
% disk/kfscmd halt
```

Damit befindet sich der Systemkern *9pcdisk* an der Stelle, an der ihn *b* und *\plan9\plan9.ini* erwarten, und *kfs* hat seine Tätigkeit eingestellt. Jetzt kann man das System anhalten und über DOS Plan 9 aus den neuinstallierten Partitionen starten. Mit *ls* kann man dort feststellen, daß man über */n/c:* Zugriff auf das DOS-Dateisystem besitzt — dafür wurde vorher die Partition *hd0dos* eingerichtet.

3.8 Grafik

Eigentlich müßte */rc/bin/termrc* die Konsole in den VGA-Modus bringen, der in Phase 2 in *\plan9\plan.ini* eingetragen wurde, aber ob das klappt, hängt von der verwendeten Grafikkarte ab. Ganz konventionell sind folgende Einträge:

```
mouseport=0
monitor=vga
vgasize=640x480x1
```

In diesem Fall sollte die Maus mit *com1* verbunden sein, und es wird der triviale VGA-Modus verwendet. Alternative Angaben zum *mouseport* sind *1* für *com2* und *ps2* für eine PS2-Maus. Als *monitor* kann man *multisync65*, *multisync75* oder gar *multisync135* für Monitoren verwenden, die bis zu 65, 75 oder 135 kHz als horizontale Frequenz akzeptieren. Höhere Auflösungen erhält man mit *800x600x1* oder *1024x768x1* für *vgasize*. Hat man erst einmal einen funktionierenden Wert, kann man */lib/vgadb* ansehen und möglicherweise mehr aus der Hardware herausholen.

Zeigt der Bildschirm nach dem Start allerdings nur grüne Buchstaben auf schwarzem Hintergrund, also keinen VGA-Modus, führt man folgende Kommandos aus:

```
term% aux/mouse 0
term% aux/vga -m vga -l 640x480x1
```

Damit existiert wenigstens der triviale VGA-Modus: schwarz/weiß mit 640 mal 480 Pixeln. Jetzt kann man *sam*(1) aufrufen und damit endlich verschiedene Dateien im System bequem ansehen.

Häufig wird die Grafikkarte nicht erkannt, obgleich sie zu einer Klasse gehört, die Plan 9 eigentlich unterstützt. Das Problem läßt sich manchmal dadurch beheben, daß man sich Texte und Positionen aus dem Hex-Dump notiert, den *aux/vga* beim ersten Mal ausgibt, wenn die Karte nicht erkannt wird. Die Karten werden in */lib/vgadb* durch Angaben wie

```
ctrl
    0xC0045="Stealth 64 DRAM Vers. 2.02"
```

beschrieben. Weicht die Version der eigenen Karte geringfügig ab, kann man eine passende Kombination aus Position und Text in */lib/vgadb* hinzufügen und einen neuen Versuch starten. Als Benutzer *none* darf man diese Datei editieren; ansonsten wäre wieder *disk/kfscmd allow* fällig.

3.9 Weitere Installationen

Besitzt man die CDROM mit dem kompletten System, kann man mit *build* komplette Plan 9-Netze einrichten, also insbesondere File-, Cpu- und speziell Authentifizierungs-Server konfigurieren. An Hardware benötigt man mindestens größere PCs oder geeignete Sparc- und Mips-Rechner mit passenden Netzverbindungen, wenigstens auf Ethernet-Basis. Mögliche Konfigurationen sind im Artikel *The Various Ports* in *Plan 9 The Documents* erläutert. Wie man vorgeht, steht dort in der Installationsbeschreibung *Installing the Plan 9 Distribution*. Beide Artikel sind auch im Netz einzusehen.

Hat man überhaupt keine passende Hardware zum direkten Umgang mit der CDROM, kann man sie als ISO9660-CD auf einem Unix-System ansehen und dort auf eine Festplatte kopieren. Anschließend sollte man mit einem *awk*-Skript Datei- und Katalognamen der Form *f0001234* und *d0005678* auf der Basis der Datei *_conform.map* umbenennen — das funktioniert nicht für ein paar obskure Unicode-Namen wie 8½, die man ganz zum Schluß unter Plan 9 korrigieren muß.

Im Katalog *sys/src/cmd/unix/u9fs* findet man auf der CDROM ein Programm *u9fs*, das unter Unix über TCP ein Dateisystem als File-Server für Plan 9 anbieten kann. Läuft dieser Server, kann das Dateisystem von Plan 9 aus mit *9fs* oder *srv* und *mount* erreicht werden, siehe *u9fs*(4). Dann kann man Kataloge mit *mkfs*(8) und *mkext*(8) auf ein lokales Plan 9-Dateisystem kopieren, wobei allerdings Zugriffsschutzinformationen verlorengehen.

Nach Einspielen der CDROM sollte man unbedingt die *Errata* nachtragen, die sich inzwischen bei *http://plan9.bell-labs.com/plan9/errata.html* angesammelt haben. Es gibt viele Korrekturen und einige neue Treiber sowie eine funktionsfähige Fassung des Web-Browsers *mothra*.

4
Namensraum

Plan 9 wurde für verschiedene Hardware-Architekturen konzipiert, zum Beispiel für einen Sparc-, 386- oder 68020-Prozessor. In einer einzelnen Sitzung ist es durchaus üblich, verschiedene Architekturen zu benutzen. So könnte beispielsweise die Oberfläche 8½ auf einem Intel-Rechner arbeiten, der mit einem MIPS-basierten CPU-Server verbunden ist, und mit Dateien, die sich auf einer Sparc als File-Server befinden.

Für den Anwender an einem Plan 9-Rechner soll aber die Sicht auf die Welt unabhängig von der aktuellen Architektur sein und auch unabhängig von ihr funktionieren. Erforderlich ist eine einheitliche Sicht der Welt für Shell-Skripte, Binär-Programme, Bibliotheks-Dateien, Include-Dateien usw. Wird zum Beispiel das Kommando

```
cat $home/tmp/x
```

aufgerufen, muß je nach der aktuellen Hardware-Architektur eine andere Binär-Datei ausgeführt werden. Das *cat*-Kommando muß für jede Architektur als Binär-Datei vorliegen. So befindet sich die Sparc-Version des *cat*-Kommandos im Verzeichnis */sparc/bin* und die anderen Versionen in den entsprechenden Verzeichnissen */386/bin*, */68020/bin* usw.

Nun soll aber das System selbst unabhängig von der aktuellen Architektur sein. Das bedeutet für dieses Beispiel, daß immer die richtige Version des *cat*-Kommandos im Verzeichnis */bin* zu finden ist. Daher bedarf es einer Technologie, um aus einem realen Dateisystem, in dem die Dateien für alle benötigten Architekturen vorhanden sind, ein virtuelles, immer gleichartiges Dateisystem zu schaffen.

Unter Plan 9 besitzt jeder Prozeß einen Namensraum, das heißt seine lokale virtuelle Sicht auf den Dateibaum. Dabei sind Namen in dem Dateibaum Pfade, die durch »/« getrennt sind und alle druckbaren Zeichen — jedes Zeichen außerhalb von hexadezimal 00-1F und 80-9F — außer Leerzeichen und Slash (»/«) enthalten dürfen. Absolute Pfade beginnen mit »/« oder »#«, relative nicht. Erzeugt ein Prozeß durch einen Aufruf von *rfork*(2) einen neuen Prozeß, erbt dieser den Namensraum seines Erzeugers, teilt sich den Namensraum mit dem Erzeuger oder startet mit einem leeren Namensraum. Näheres zum Namensraum-Management bei *rfork* ist in Kapitel 13.3 nachzulesen.

Ein Server unter Plan 9 realisiert ein Dateisystem, das heißt, er bietet Zugriff auf seine Ressourcen durch Operationen wie *open*-, *read*- oder *write*- bezügliche Dateien an, die durch Pfade identifiziert werden. Generell gibt es zwei verschiedene Arten von Servern: Kernel- und User-Server.

Bei Kernel-Servern beginnen Pfade mit *#xy*, wobei»x« ein Zeichen ist, welches den Kernel-Server identifiziert, und »y« optional ein Wort ohne Leerzeichen und »/« ist, welches innerhalb eines Kernel-Servers ein Dateisystem auswählt. Das zweite Zeichen ist nur für Kernel-Server wichtig, die mehrere Dateisysteme anbieten, und ist daher optional. Bei User-Servern sind die Pfade relativ zu einem Punkt im Namensraum, den *mount*(1,2) festgelegt hat. Kernel-Server sind fest in den Betriebssystemkern eingebaut, wohingegen User-Server eigenständige Prozesse sind.

Um einen einheitlichen Namensraum zu realisieren, gibt es *bind*(1,2); um User-Server anzusprechen, gibt es *mount*(1,2). Beide Operationen erweitern und verändern den Namensraum des aufrufenden Prozesses.

Der Kern kommuniziert mit beiden Server-Arten über das Plan 9-File-Protokoll 9P. Der Kern merkt sich, welcher Server für welchen Teil im Namensraum verantwortlich ist und wandelt Dateioperationen in 9P-Nachrichten an den zuständigen Server um. 9P ist ein zustandsbehaftetes Protokoll. Die einzelnen Nachrichten von 9P dienen zur Manipulation von Dateien und Dateibäumen. Man kann mit ihnen durch die Dateibäume der Server wandern, Dateien lesen und schreiben sowie Dateien kreieren und löschen. 9P ist als ein Satz von Transaktionen strukturiert. Eine Transaktion besteht aus einer Anfragenachricht des Betriebssytemkerns an den Server und der zugehörigen Antwortnachricht des Servers an den Kern. Das Protokoll 9P der Plan 9-Release von 1995 definiert 15 Nachrichten, die zu einem Server geschickt werden können, die sogenannten *T-Nachrichten*, und 16 Nachrichten, die vom Server an den Kern als Antwort auf die T-Nachricht geschickt werden können, die *R-Nachrichten*. Der Server antwortet auf eine T-Nachricht entweder mit einer R-Nachricht gleichen Typs wie die T-Nachricht oder mit einer Fehler-Nachricht. Diese Fehler-Nachricht ist der Grund, warum es eine R-Nachricht mehr als T-Nachrichten gibt. Typische 9P-Nachrichten sind zum Beispiel *walk*, *read*, *stat* oder *write*.

Kern-Server sind Bestandteile des Kerns. Es wäre daher ineffizient, wenn der Kern Kernel-Servern, das heißt praktisch sich selbst, Nachrichten schicken würde. Der Kern schickt einem Kernel-Server daher keine 9P-Nachrichten, sondern ruft im Server zu den 9P-Nachrichten analoge 9P-Funktionen auf.

Eine ausführlichere Beschreibung zu 9P ist in Kapitel 5.1 nachzulesen.

4.1 *bind*

Der anfängliche Namensraum, Kernel-Server

Im Plan 9-Betriebssystemkern existieren — unverdeckbar für jeden Prozeß — mehrere Server, die sogenannten *Kernel-Server*. Diese stellen jeweils eine gewisse Funktionalität als Dateisystem zur Verfügung. Die Namensräume der Kernel-Server können über #x angesprochen werden, wobei für das »x« jeweils ein Kernel-Server-typisches Zeichen einzusetzen ist. So verwaltet zum Beispiel der Server *root*, #/ das root-Dateisystem oder der Server *bit*, #b u.a. die Maus. Optional kann nach dem »x« ein Wort folgen, welches bei Kernel-Servern, die mehrere Dateisysteme anbieten, eines auswählt. Als Beispiel stellt der Kernel-Server *ip*, #I die Netzwerk-Protokolle *tcp, udp* und *il* als drei Dateisysteme zur Verfügung.

Eine 9P-Operation wie *read*, *write* oder *walk* zu #x ist im Kern ein Funktionsaufruf in den Server. *Walk* ist eine Suchoperation — sie bezieht sich auf einen Katalog (also zum Beispiel die Wurzel eines Kernel-Servers) und hat eine Komponente als Argument, die der Server dann finden muß. Systemaufrufe wie *create*(2) oder *chdir*(2) setzen sich zum Beispiel aus *read*, *write* und *walk* zusammen.

bind(2) manipuliert pro Prozeß seine Sicht der Ressourcen, das heißt seinen Namensraum, in dem ein existenter Katalog mit einem existenten Katalog verdeckt

oder verlängert wird. Die Funktionaliät von *bind*(2) steht auch als Kommando *bind*(1) zur Verfügung. (Die Verwendung von *bind*(1,2) mit Dateien wird im weiteren Verlauf als eigenständiger Teil erklärt.)

Damit eine anfängliche konventionelle Sicht entsteht, ist

```
bind #/ /
```

vordefiniert. Dieser Aufruf von *bind*(1) bedeutet, daß das Dateisystem des Kernel-Servers *root*, #/ im Wurzelverzeichnis »/« des aufrufenden Prozesses sichtbar sein soll. Der Server *root* stellt einige leere Kataloge und eine Datei *boot* zur Verfügung:

Katalog/Datei	Verwendung
/boot	
/dev	Geräte
/env	Environment-Variablen
/proc	Prozeßinformationen
/net	

Nach Konvention werden beim Systemstart, zum Beispiel durch

```
bind #w /dev
```

einige Kernel-Server mit bestimmten Pfaden verbunden:

Server	Katalog nach *Konvention*	Verwendung
`#w`	*/dev*	Platte
`#c`	*/dev*	Geräte
`#e`	*/env*	Environment-Variablen
`#p`	*/proc*	Prozeßinformationen
`#Itcp`	*/net/tcp*	tcp-Schnittstelle
`#Iil`	*/net/il*	il-Schnittstelle
usw.		

Eine komplette Beschreibung des Aufbaus des Namensraums während des Systemstarts folgt später in diesem Kapitel.

Jeder Prozeß hat seinen eigenen Namensraum. Er kann ihn mit seinen Abkömmlingen teilen oder auch einem neuen Prozeß einen leeren oder eine Kopie seines eigenen Namensraums zur Verfügung stellen. Nur im ersten Fall wirkt sich ein *bind*(1,2) noch auf einen anderen als auf den aufrufenden Prozeß aus.

Die Ausgabe von *ls -l* verrät, welcher Kernel-Server für eine Datei verantwortlich ist. In der Ausgabe steht das Zeichen nach den Zugriffsrechten für diesen Server. So symbolisiert im folgendem Beispiel das *e* den Kernel-Server *env*, #e:

```
% cd /env
% ls –l
    ...
––rw–rw–rw– e 0 dbkuehl dbkuehl     7 Apr  5  1995 cpu
––rw–rw–rw– e 0 dbkuehl dbkuehl     5 Apr  5  1995 cputype
    ...
––rw–rw–rw– e 0 dbkuehl dbkuehl     5 Apr  5  1995 objtype
    ...
```

```
--rw-rw-rw- e 0 dbkuehl dbkuehl     7 Apr  5  1995 sysname
--rw-rw-rw- e 0 dbkuehl dbkuehl     6 Apr  5  1995 terminal
--rw-rw-rw- e 0 dbkuehl dbkuehl  1162 Apr  5  1995 timezone
--rw-rw-rw- e 0 dbkuehl dbkuehl     7 Apr  5  1995 user
%
```

Bestehende Namensräume und *bind*

Ein bestehender Namensraum kann mit den Systemaufrufen *bind*(2) und *mount*(2) erweitert werden. Beide Operationen werden durch *unmount*(2) rückgängig gemacht. Alle drei Operationen stehen auch als Kommandos zur Verfügung.

bind verdeckt oder verlängert einen existenten Katalog mit einem existenten Katalog. Abgesehen von der bereits erwähnten Verknüpfung von Kernel-Servern (#w) mit konventionellen Namen (/dev/hd0) dient das zu Operationen, die an symbolische Links bei UNIX erinnern. Beispielsweise ist */tmp* a priori nicht schreibbar, aber durch

```
bind -c /usr/axel/tmp /tmp
```

überdeckt der Benutzer *axel* das Verzeichnis */tmp* mit seinem eigenen *tmp*-Katalog. Da Namensräume normalerweise kopiert und gemeinsam benutzt werden, erhalten neue Prozesse damit einen schreibbaren Bereich für temporäre Dateien.

Im nachfolgenden Text soll anhand eines Beispiels die Benutzung von *bind* und *unmount* näher erläutert werden. Zu Beginn werden die Verzeichnisse *test* und *test2* angelegt, die Datei *test/hello* erzeugt und mit dem Text *Hello World* gefüllt sowie die Datei *test2/x* angelegt. Erwartungsgemäß enthält das Verzeichnis *test* die Datei *hello* und das Verzeichnis *test2* die Datei *x*:

```
% mkdir test
% mkdir test2
% echo Hello World > test/hello
% touch test2/x
% ll test*
--rw-rw-r-- M 3 bernd bernd 12 Jun 10 21:22 test/hello
--rw-rw-r-- M 3 bernd bernd  0 Jun 10 21:22 test2/x
```

Durch

```
bind test test2
```

wird nun der Inhalt des Verzeichnisses *test2* durch den Inhalt von *test* verdeckt. Das zweite Argument von *bind* ist das Verzeichnis, dessen Inhalt durch das im ersten Argument angegebene Verzeichnis verdeckt wird. Wie zu sehen ist, enthält *test2* nun die Datei *hello*, und der alte Inhalt, also die Datei *x*, ist nicht mehr sichtbar:

```
% bind test test2
% ll test*
--rw-rw-r-- M 3 bernd bernd 12 Jun 10 21:22 test/hello
--rw-rw-r-- M 3 bernd bernd 12 Jun 10 21:22 test2/hello
% cat test2/hello
Hello World
```

Im Verzeichnis *test2* darf in diesem Fall keine neue Datei angelegt werden. Dieses kann durch die Flagge *-c* (*create*) beim Aufruf von *bind* erlaubt werden, das heißt, mit dem Aufruf

```
bind -c test test2
```

im obigen Beispiel wäre dann das Kommando

```
date > test2/date
```

erfolgreich ausgeführt worden. Änderungen im Quellverzeichnis sind auch im Zielverzeichnis (und umgekehrt bei Aufruf von *bind* mit der Option -c) sichtbar, wie an dem weiteren Beispiel von *date* klar wird:

```
% date > test2/date
rc: test2/date: mounted directory forbids creation
% date > test/date
% ll test*
——rw—rw—r—— M 3 bernd bernd 29 Jun 10 21:23 test/date
——rw—rw—r—— M 3 bernd bernd 12 Jun 10 21:22 test/hello
——rw—rw—r—— M 3 bernd bernd 29 Jun 10 21:23 test2/date
——rw—rw—r—— M 3 bernd bernd 12 Jun 10 21:22 test2/hello
% cat test2/date
Mon Jun 10 21:23:42 GMT 1996
```

Das Kommando *unmount test2* macht den Effekt von *bind* wieder rückgängig, das heißt, der alte Inhalt des Verzeichnisses *test2*, die Datei *x*, ist wieder sichtbar:

```
% unmount test2
% ll test*
——rw—rw—r—— M 3 bernd bernd 29 Jun 10 21:23 test/date
——rw—rw—r—— M 3 bernd bernd 12 Jun 10 21:22 test/hello
——rw—rw—r—— M 3 bernd bernd 0 Jun 10 21:22 test2/x
```

bind und *union-directories*

Mit *bind* können nicht nur existente Kataloge durch andere existente Kataloge überdeckt werden. Es besteht darüber hinaus die Möglichkeit, den aktuellen Inhalt eines Katalogs um den von anderen Katalogen zu erweitern. Solche »zusammengesetzten« Kataloge heißen *union-directories*. Der Umgang mit diesen wird nun im folgenden dargestellt.

Als Beispiel werden zuerst die drei Verzeichnisse *test*, *after* und *before* angelegt. Durch *bind* wird der (leere) Inhalt des Katalogs *test* mit den vom Kernel-Server *cons* zur Verfügung gestellten Dateien überdeckt. Dabei behandelt die Plan 9-Shell *rc* alles ab einem »#« in der Eingabezeile als Kommentar. Deshalb muß dieses Zeichen in der Shell für den *bind*(1)-Aufruf geschützt werden. Wie an der Ausgabe von *lc* zu sehen ist, enthält das Verzeichnis nun einige Dateien.

```
% mkdir test
% mkdir after
% mkdir before
% bind '#c' test
```

```
% lc test
authcheck      cputime        lights         ppid
authenticate   hostdomain     msec           swap
authenticator  hostowner      noise          sysname
clock          hz             null           sysstat
cons           key            pgrpid         time
consctl        klog           pid            user
% echo hello world! > test/cons
```

In den Katalogen *after* und *before* wird nun jeweils eine Datei angelegt. Dabei soll der Dateiname den Katalog eindeutig charakterisieren. Ruft man *bind* ohne Option oder nur mit der Option *-c* auf, so wird der Inhalt eines Katalogs überdeckt. Mit den Optionen *-a (after)* und *-b (before)* kann nun der aktuelle Inhalt erweitert werden. So fügt im unteren Beispiel das erste *bind* mit der Option *-b* den Inhalt des Katalogs *before* vor dem aktuellen Inhalt von *test* ein. Der zweite Aufruf von *bind* erweitert *test* am Ende um den Inhalt von *after*. Die Ausgabe von *lc* verdeutlicht, daß der Katalog *test* nun die Inhalte des Verzeichnisses *after*, die Dateien des Kernel-Servers *cons* und den Inhalt von *after* beinhaltet. Da das Kommando *lc* seine Ausgabe sortiert, wird mit dem Schalter *-n* die unsortierte Reihenfolge der Dateien in *test* ersichtlich.

```
% touch after/after_file
% touch before/before_file
% bind —b before test
% bind —ac after test
% lc test
after_file     consctl        lights         swap
authcheck      cputime        msec           sysname
authenticate   hostdomain     noise          sysstat
authenticator  hostowner      null           time
before_file    hz             pgrpid         user
clock          key            pid
cons           klog           ppid
% lc —n test
before_file    cputime        msec           sysname
authenticate   hostdomain     noise          sysstat
authcheck      hostowner      null           time
authenticator  hz             pgrpid         user
clock          key            pid            after_file
cons           klog           ppid
consctl        lights         swap
```

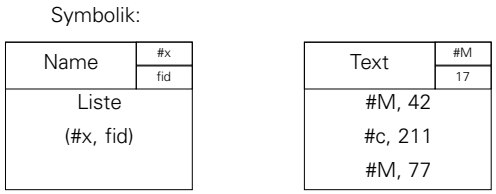
Was passiert nun, wenn im Katalog *test* eine neue Datei erzeugt werden soll? Was ist, wenn die Verzeichnisse *after* und *before* Dateien mit dem gleichen Namen beinhalten?

Ein Beispiel zum ersten Punkt enthält der folgende Ausschnitt einer Shell-Sitzung. Hier wird im Katalog *test* eine Datei namens *date* mit der Ausgabe desselben Kommandos gefüllt. Ein Listing des Verzeichnisses *test* zeigt, daß die Datei *date* angelegt worden ist. Die unsortierte Ausgabe und der *cat-Aufruf* verraten das Verzeichnis *after* als Ursprung der neuen Datei:

```
% date > test/date
% lc test
after_file     consctl       klog          ppid
authcheck      cputime       lights        swap
authenticate   date          msec          sysname
authenticator  hostdomain    noise         sysstat
before_file    hostowner     null          time
clock          hz            pgrpid        user
cons           key           pid
% lc –n test
before_file    cputime       msec          sysname
authenticate   hostdomain    noise         sysstat
authcheck      hostowner     null          time
authenticator  hz            pgrpid        user
clock          key           pid           after_file
cons           klog          ppid          date
consctl        lights        swap
% cat test/date after/date
Tue Mar 27 10:04:42 MET 2029
Tue Mar 27 10:04:42 MET 2029
```

Was genau geschah beim Kreieren der Datei *date*? Ein Verzeichnis unter Plan 9 setzt sich intern im wesentlichen aus einer Liste zusammen. Jeder Listeneintrag besteht aus einer *fid* und aus einem Verweis auf einen Kernel-Server. Eine *fid* ist innerhalb von 9P eine Integerzahl und identifiziert (analog zu File-Deskriptoren unter UNIX) während ihrer Lebensdauer eindeutig eine Datei oder ein Verzeichnis innerhalb des Servers. Eine genaue Beschreibung dieses Sachverhalts und des zugrundeliegenden File-Protokolls *9P* enthält Kapitel 5.1.

Der Katalog *test* sieht also wie folgt aus (die *fid*-Nummern wurden hier beispielhaft ausgewählt):



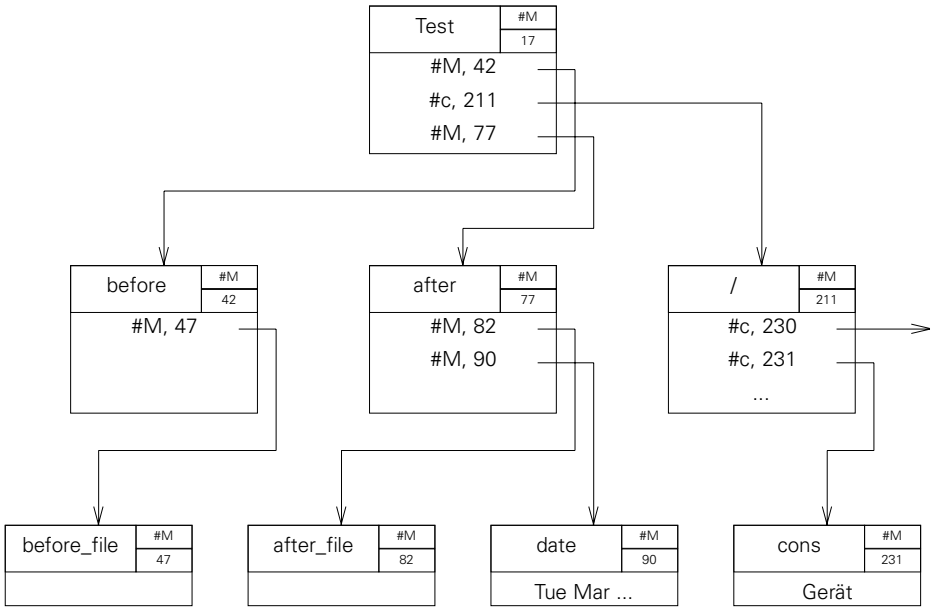
Die Liste für *test* besteht hier aus drei Einträgen. Der erste Eintrag gehört zum Katalog *before*, der zweite zum Kernel-Server *cons,* #c und der letzte zum Katalog *after*. Warum der Kernel-Server *#M (mnt)* die Kataloge *after* und *before* verwaltet, ist an dieser Stelle noch unklar. Wir werden aber später darauf zurückkommen.

Was passiert nun bei der Abarbeitung eines Kommandos wie

```
ls -l test
```

? Es wird pro Eintrag der zugehörige Server mit den 9P-Nachrichten *stat*, *open* und *read* nach den Informationen über das mit der *fid* verwiesene Objekt befragt.

Der gesamte Baum unterhalb von test hat dann folgendes Aussehen:



Die *fids* werden nicht (wie die Grafik vielleicht fälschlicherweise symbolisiert) einmal pro Lebensdauer der referierten Objekte (Datei, Katalog) kreiert, sondern dynamisch vom zuständigen Server erzeugt, siehe Kapitel 5.1 über *9P*.

Will man nun im Verzeichnis *test* eine neue Datei schaffen, werden von vorne nach hinten alle Einträge in der Liste durchlaufen. Der erste Eintrag ist der Katalog *before*. Dort darf keine neue Datei angelegt werden, da dieser Teil beim Aufruf von *bind* ohne die *-c (create)* Option dem Verzeichnis *test* hinzugefügt worden war. Gleiches gilt für den zweiten, den *cons*-Eintrag. Der letzte Eintrag repräsentiert den *after*-Katalog. Dieser ist mit der Option *-c* zu *test* hinzugefügt worden; das Erzeugen neuer Dateien ist also erlaubt, und auch die Zugriffsrechte innerhalb des Verzeichnisses *after* verbieten dieses nicht. Daher enthält nun der Katalog *after* die neue Datei.

Nun zum zweiten Problem. Enthalten in dem Beispiel die Kataloge *after* und *before* eine Datei oder ein Verzeichnis gleichen Namens und wird darauf zugegriffen, so werden wieder von vorne nach hinten alle Einträge abgearbeitet, und der erste passende wird benutzt:

```
% echo after double > after/double
% echo before double > before/double
% lc test
after_file     cputime       klog          swap
authcheck      date          lights        sysname
authenticate   double        msec          sysstat
authenticator  double        noise         time
before_file    hostdomain    null          user
     ...
```

```
% lc –n test
before_file   cputime       noise         time
double        hostdomain    null          user
authenticate  hostowner     pgrpid        after_file
authcheck     hz            pid           date
authenticator key           ppid          double
    ...
% cat test/double
before double
% lc after
after_file date       double
% lc before
before_file double
```

Aus der unsortierten Ausgabe von *ls -l* ist noch einmal pro Datei der zuständige Kernel-Server zu sehen:

```
% cd test
% ll –n
––rw–r––r–– M 4 dbkuehl dbkuehl  0 Mar 27 10:03 before_file
––rw–r––r–– M 4 dbkuehl dbkuehl 14 Mar 27 17:07 double
––rw–rw–rw– c 0 dbkuehl dbkuehl  0 Apr  5  1995 authenticate
    ...
––rw–rw––––– c 0 dbkuehl dbkuehl  0 Apr  5  1995 cons
    ...
––rw–rw–rw– c 0 dbkuehl dbkuehl 28 Apr  5  1995 user
––rw–r––r–– M 4 dbkuehl dbkuehl  0 Mar 27 10:03 after_file
––rw–r––r–– M 4 dbkuehl dbkuehl 29 Mar 27 10:04 date
––rw–r––r–– M 4 dbkuehl dbkuehl 13 Mar 27 17:07 double
```

bind und Dateien

Mit *bind* kann man nicht nur einen vorhandenen Katalog um einen anderen Katalog erweitern oder verdecken. Darüber hinaus besteht die Möglichkeit, einzelne Dateien mit einer anderen Datei zu überdecken:

```
% echo This file lies in the file tree above > file
% echo the directories after,test and before! >> file
% cat file
This file lies in the file tree above
the directories after,test and before!
% bind file test/after_file
% lc –n test
before_file   cputime       noise         time
double        hostdomain    null          user
authenticate  hostowner     pgrpid        after_file
authcheck     hz            pid           date
authenticator key           ppid          double
    ...
% cat test/after_file
This file lies in the file tree above
the directories after,test and before!
% unmount test/after_file
% cat test/after_file
%
```

Ein solcher Aufruf von *bind* gleicht sehr einem Link unter UNIX.

Binär-Dateien

Beim Systemstart bestimmt der Kern die aktuelle Rechnerarchitektur und setzt die Enviromentvariable *cputype* dementsprechend (386, sparc, mips, ...). Anschließend werden mit

```
bind /$cputype/bin /bin
bind -a /rc/bin /bin
```

die richtigen Binär-Dateien und die Shell-Skripten nach */bin* gebunden. Als Anwender kann man dann noch eigene Kommandos und Shell-Skripte zu */bin* hinzufügen:

```
bind -b $home/bin/rc /bin
bind -b $home/bin/$cputype /bin
```

Diese Befehle sorgen dafür, daß eigene Kommandos vor öffentlichen Kommandos und binäre Kommandos vor Skripten gefunden werden.

4.2 Typische Kernel-Server

Dieser Abschnitt stellt einige typische Kernel-Server und ihre Verwendung vor. Einen ersten kurzen Überblick über die vorhandenen Kernel-Server enthält die nachfolgende Tabelle:

Name	Name	zur Verfügung gestellter Dateibaum	Funktionalität
`#b`	bit	*/dev/bitblt, mouse, screen*	Bitmap-Schirm und Maus
`#c`	cons	*/dev/cons, ...*	Konsole und viele mehr
`#d`	dup	*/fd/0, ...*	Filedeskriptoren
`#e`	env	*/env/variable*	Environment
`#I`	ip	*/net/tcp/..., /net/il/..., /net/udp/...*	Internet-Protokolle
`#T`	kprof	*/dev/kpctl, kpdata*	Kernel-Profiling
`#l`	lance	*/net/ether/...*	Ethernet
`#p`	proc	*/proc/n/...*	Prozeß-Informationen
`#s`	srv	*/srv*	Server-Registratur
`#f`	floppy	*/dev/fd[0-3]disk, fd[0-3]ctl*	Floppy
`#H`	ata	*/dev/sd0disk, sd0partition, ...*	SCSI-Festplatte
`#w`	wren	*/dev/hd0disk, hd0partition, ...*	IDE-Festplatte
`#|`	pipe		Pipelines
`#/`	root	*/boot, /dev, /proc, /net, ...*	Wurzel des Namensraums

So wird zum Beispiel der Service des Kernel-Servers *env*, #e, durch das Kommando

```
bind -c #e /env
```

dem Namensraum im Standardverzeichnis */env* hinzugefügt, was aber lediglich eine Konvention ist. Der Anwender kann den Service an jeder von ihm gewünschten Stelle im Namensraum einbinden.

env

Der Kernel-Server *env* verwaltet ein flaches Dateisystem, das die Plan 9-Shell *rc* als Environment nützt und das mit den Funktionen *getenv(2)* und *setenv(2)* bearbeitet werden kann. Das Standardverzeichnis von *env* ist */env*. In dem von *env* verwalteten Dateisystem kann man Dateien erzeugen und mit Daten füllen. Im folgenden Beispiel wird die Variable *hello* erzeugt und mit dem Wert *Hello World* belegt:

```
% mkdir test
% bind '#e' test
% cd test
% lc                    # eine Auswahl
8½srv       cputype   home      pid       status    sysname   user
cpu         fn#ll     objtype   service   swap      timezone
% hello='Hello World'
% ll hello
--rw-rw-rw- e 0 dbkuehl dbkuehl 12 Apr  5  1995 hello
% cat hello
Hello World% echo $hello
Hello World
% echo Hello World 2 >hello2
% ls -l hello2
--rw-rw-rw- e 0 bischof bischof 14 Apr  5  1995 hello2
% cat hello2
Hello World 2
```

Im Beispiel ist eine Auswahl der existierenden Environment-Variablen dargestellt. Wird in der Shell *rc* eine Variable angelegt oder verändert, so wird der Wert der Variablen von *rc* in der Datei *#e/variablename* abgelegt und kann von dort über normale Dateioperation von *rc* oder jeden anderen Prozeß gelesen werden. Um alle Standard-Enviromentvariablen kennenzulernen, ist es sinnvoll, sich den Inhalt dieser Variablen anzuschauen.

Natürlich darf nicht nur *rc*, sondern jeder Prozeß im Dateisystem von *env* Dateien erzeugen, füllen und löschen.

dup

Der *dup*-Server, #d, unterhält pro Prozeß einen flachen Dateibaum, dessen Dateien als Namen Dezimalnummern haben. Nach Konvention wird das Dateisystem von *dup* nach */fd* montiert. Eine Datei mit dem Namen *n* korrespondiert mit einem offenen Filedeskriptor *n* im aktuellen Prozeß. Ein *open*(*2*)-Aufruf gegen die Datei *n* liefert einen Filedeskriptor, der die gleiche Datei wie *n* repräsentiert.

cons

Der Kernel-Server *cons*, #c, bietet eine Vielzahl von Gerätedateien an. Das Standardverzeichnis ist */dev*. Auch hierzu ein Beispiel:

```
% cd /dev
% cat swap
547/2867 memory 0/4096 swap
```

```
% cat time
 1844346936 % cat cons
Hey to all!
Hey to all!
^D% echo 440 2000 > noise
% echo Hello World > null
```

Die Datei *swap* enthält Informationen über den Zustand des Hauptspeichers und des Swap-Bereichs. *time* beinhaltet die Anzahl der vergangenen Sekunden seit dem 1. Januar 1970. Sie ist die Grundlage für Kommandos wie *date* oder C-Bibliotheksfunktionen wie *time*(*2*). Die Datei *cons* entspricht der Unix-Datei */dev/console. Noise* dient zur Ausgabe von Tönen. Das erste Argument gibt die Frequenz des Tons in Hertz an, das zweite die Zeitdauer in Millisekunden. Das obige Beispiel gibt den Kammerton A für zwei Sekunden aus. Die Datei *null* ist das *Null*-Gerät. Alle Daten, die in diese Datei gelenkt werden, werden ignoriert. Diese Beispiele zum Kernel-Server *cons* bilden lediglich eine kleine Auswahl der von *cons* zur Verfügung gestellten Dateien. Eine Beschreibung aller Dateien enthält die Manual-Seite *cons*(*3*). Wir empfehlen an dieser Stelle, diese zu lesen, um so einen Überblick über die Dateien und deren Funktionalität zu erhalten.

proc

Ein weiterer interessanter Kernel-Server ist *proc*, #p. Das Standardverzeichnis ist */proc*. *Proc* stellt in einem Dateibaum Informationen über die Prozesse des Systems zur Verfügung:

```
% pwd
/proc
% lc
1   112 114 116 12  179 189 191 3   35  44  51  66  82
11  113 115 117 13  18  190 2   33  40  5   59  8   9
% cd 82
% lc
ctl     note    notepg  segment text
mem     noteid  proc    status  wait
% cat status
clock  dbkuehl  Read  160  760  1936280  0  260  0  80  82
```

Eine Implementierung von *ps* wird damit zur *awk*-Fingerübung.

srv

Ein weiterer, im Zusammenhang mit User-Servern, wichtiger Kernel-Server ist *srv*, #s. Dieser bietet ein flaches Dateisystem an, welches eine Funktionalität analog zu *named pipes* unter UNIX bietet. Das Standardverzeichnis für *srv* ist /srv . Ein Prozeß kann in dem Dateisystem von *srv* eine Datei erzeugen und in dieser den Wert eines Filedeskriptors ablegen. Dies ist üblicherweise eine Seite einer Pipe, welche dann in dem Prozeß geschlossen wird. Öffnet nun ein anderer Prozeß mit *open*(2) die erzeugte Datei, so erhält er einen zum ursprünglichen Filedeskriptor koresspondierenden Filedeskriptor. War der Ursprung dessen eine Pipe, so besitzen nun zwei verschiedene Prozesse jeweils ein Ende der gleichen Pipe. Unter Plan 9 sind Pipes bidirektional.

Zur Vertiefung folgt an dieser Stelle ein kleines Beispiel. Im nachfolgenden Quelltext wird eine Pipe erzeugt und eine Seite publik gemacht:

```
int fd, p[2],n;
char buf[32];

pipe(p);

fd = create("/srv/namedpipe", 1, 0666);
sprint(buf, "%d", p[0]);
write(fd, buf, strlen(buf));
close(fd);
close(p[0]);

write(p[1], "hello\n", 6);

sleep(30);

n=read(p[1],buf,sizeof buf);
write(1,buf,n);
```

Nach dem *write*(2) kann nun ein anderer Prozeß die Datei */srv/namedpipe* mit *open*(2) öffnen und mit *read*(2) den Text *hello* lesen. Analog dazu empfängt der obere Prozeß mit *read*(2) Daten auf seiner Seite der Pipe. Im folgenden Auszug aus zwei parallelen Shell-Sitzungen wird in der linken Shell das Beispielprogramm unter den Namen *test_srv* gestartet. In der anderen, rechten Shell werden die Daten aus der Pipe gelesen und neue Daten in die Pipe geschrieben.

```
% test_srv
                    % pwd
                        /srv
                    % read < namedpipe
                        hello
                    % echo hi >> namedpipe
hi
```

Daß dieses auch in der Shell funktioniert, zeigt folgendes Beispiel:

```
% echo Hello World | { echo 0 > '#s/namedpipe'; }
% ls '#s'
#s/8½.dbkuehl.8
#s/boot
#s/cs
#s/namedpipe
% cat '#s/namedpipe'
Hello World
% cd /srv
% rm namedpipe
% echo Hello World | { echo 0 > namedpipe; }
% cat namedpipe
Hello World
```

Einen kompletten Überblick über die vorhandenen Kernel-Server bietet das Kapitel 3 der Manual-Seiten. Dort ist Näheres über die Funktionalität aller in der obigen Tabelle aufgeführten Kernel-Server nachzulesen. Lesen Sie diese Seiten und experimentieren Sie ein wenig mit den Kernel-Servern. Es lohnt sich!

Der Namensraum vieler Kernel-Server, deren Service zum Arbeiten mit Plan 9 notwendig ist, wird beim Systemstart an die zugehörigen Standard-Verzeichnisse montiert. Das genaue Vorgehen wird in Abschnitt 4.6 dieses Kapitels ausführlich beschrieben.

4.3 *mount*

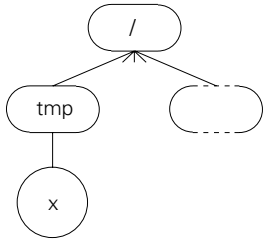
Um den Namensraum um Dateisysteme zu erweitern, die nicht in einem existenten Namensraum vorhanden sind, bedarf es einer Technik, die neue, zusätzliche Dateisysteme anbietet. Dies kann durch die sogenannten *User-Server* geschehen. User-Server sind eigenständige Prozesse, die intern ein oder mehrere Dateisysteme verwalten und diese dem Anwender zur Verfügung stellen. Ein User-Server hält eine Pipe-Verbindung, über die er Anfragen bekommt und beantwortet. Diese Verbindung wird mit *mount*(2) in den Namensraum eingebaut; alternativ kann sie als Ressource des speziellen Kernel-Servers *srv*, #s, abgelegt werden, damit sie für *mount*(1) zur Verfügung steht.

Nach dem Start eines neuen User-Servers erzeugt dieser eine Pipe als seine Kommunikationsleitung zum Rest des Systems. Viele User-Server hinterlegen die freie Seite der Pipe in eine Datei im Katalog */srv*, welches nach Konvention vom Kernel-Server *srv*, #s, verwaltet wird. Durch einen *mount*(1)-Aufruf auf die erzeugte Datei in */srv* wird nun die Wurzel eines Dateisystems des User-Servers in den aktuellen Namensraum montiert. Das Kommando *mount*(1) öffnet die Datei und erhält so einen mit der öffentlichen Seite der Kommunikationspipe verbunden Filedeskriptor. Mit dessen Wert als Argument und der gewünschten Position im Namensraum führt *mount*(1) den Systemaufruf *mount*(2) aus. Der Kern merkt sich im Zuge von *mount*(2), daß der User-Server für den ausgewählten Teilbaum des Namensraums verantwortlich ist. Zugriffe auf den zum Server gehörigen Baum werden in entsprechende 9P-Nachrichten über die Pipe an den Server übersetzt. Der User-Server liest die Nachrichten, verarbeitet sie und schickt wiederum über die Pipe die entsprechenden Antwort-Nachrichten an den Kern.

In einem Beispiel soll nun im folgenden der Service des User-Servers *ramfs* in Anspruch genommen werden. Der Server *ramfs* stellt die Funktionalität einer RAM-Disk zur Verfügung. Zu Beginn wird in das Wurzelverzeichnis gewechselt, dort wird die Datei *tmp/x* erzeugt. In diese wird der Text *hi* geschrieben:

```
% cd /
% echo hi > tmp/x
% ls –l tmp/x
––rw–rw–r–– M 3 bernd bernd 3 May 28 19:20 tmp/x
```

Der Aufbau des Namensraums ist in der nachfolgenden Abbildung dargestellt:



Es folgen die einzelnen Schritte beim Starten des User-Servers *ramfs* und beim Montieren des Dateisystems von *ramfs*:

```
% ramfs –s
% mount –c /srv/ramfs /tmp
% echo ramfs > tmp/y
% ls –l tmp
––rw–rw–r–– M 35 bernd bernd 6 May 28 19:20 tmp/y
% cat tmp/y
ramfs
```

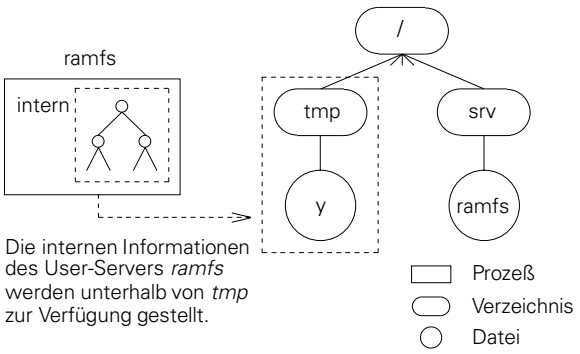
Durch das Kommando

```
ramfs -s
```

in der Shell wird der Server *ramfs* gestartet. Dieser erzeugt die Datei */srv/ramfs*. Der Anwender kann nun an einer von ihm gewünschten Stelle den Dateibaum von *ramfs* in seinen Namensraum montieren. Durch

```
mount -c /srv/ramfs /tmp
```

wird der alte Inhalt von */tmp* durch das Dateisystem von *ramfs* überdeckt. Die Option *-c* in dem *mount*-Aufruf erlaubt das Erzeugen von neuen Dateien in dem vom User-Server verwalteten Dateibaum. Zu Anfang ist das Dateisystem von *ramfs* leer. Durch *echo* mit zugehörender Dateiumlenkung wurde im obigen Beispiel eine Datei *y* im vom User-Server *ramfs* verwalteten Dateisystem erstellt. Wie zu erkennen ist, enthält */tmp* nun nicht mehr die Datei *x*, sondern das Dateisystem von *ramfs*, die Datei *y.* Der alte Inhalt von */tmp* ist nicht mehr sichtbar. Der Namensraum hat nun folgenden Aufbau:



Analog und mit der gleichen Funktionaliät wie bei *bind* gibt es auch für *mount* die Flaggen *-b* (*before*) und *-a* (*after*) zu *mount*.

Der Kern merkt sich im Laufe der Abarbeitung des *mount*-Befehls, daß *ramfs* für */tmp* verantwortlich ist. Alle Dateioperationen bezüglich Dateien unterhalb von */tmp* werden vom Kern in entsprechende 9P-Nachrichten an den User-Server *ramfs* umgewandelt. Der Server empfängt die 9P-Nachrichten, verarbeitet sie und antwortet dem Kern durch entsprechende 9P-Nachrichten.

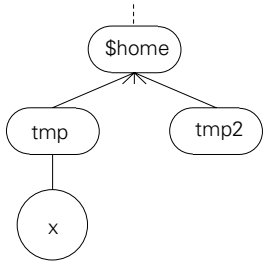
Durch *unmount /tmp* wird nun der Effekt von *mount* rückgängig gemacht, so daß der alte Inhalt von *tmp* wieder sichtbar ist. Abschließend muß noch der Prozeß *ramfs* beendet werden:

```
% unmount /tmp
% ls -l tmp/x
--rw-rw-r-- M 3 bernd bernd 3 May 28 19:20 tmp/x
% cat tmp/x
hi
% kill ramfs | rc
%
```

mount und *union-directories*

Plan 9 besitzt, wie bereits in einem vorherigen Abschnitt dieses Kapitels erklärt, sogenannte *union-directories*, das heißt, in einem Verzeichnis kann der Inhalt von mehreren Verzeichnissen sichtbar sein. Dies gilt auch für das Einbinden von User-Servern in den Namensraum, was im folgenden Beispiel verdeutlicht wird. Zu Beginn werden die Verzeichnisse *tmp* und *tmp2* sowie die Datei *tmp/x* erzeugt. Letztere wird mit dem Text *alt* gefüllt:

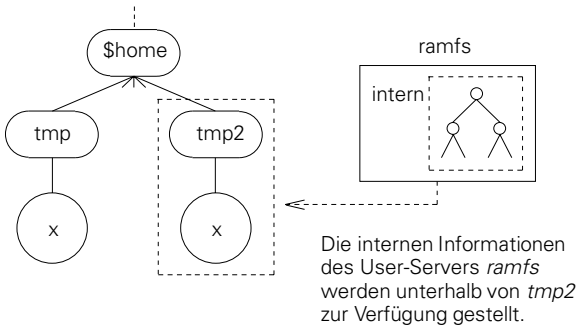
```
% cd $home
% mkdir tmp
% mkdir tmp2
% echo alt > tmp/x
% ls -l tmp
--rw-rw-r-- M 3 bernd bernd       4 May 28 19:52 tmp/x
```



Anschließend wird der User-Server *ramfs* gestartet und der Service von *ramfs* nach *tmp2* montiert. Dort wird eine Datei *x* mit dem Inhalt *ramfs* kreiert:

```
% ramfs -s
% mount -c /srv/ramfs tmp2
% echo ramfs > tmp2/x
% ls -l tmp2
--rw-rw-r-- M 51 bernd bernd       6 May 28 19:53 tmp2/x
```

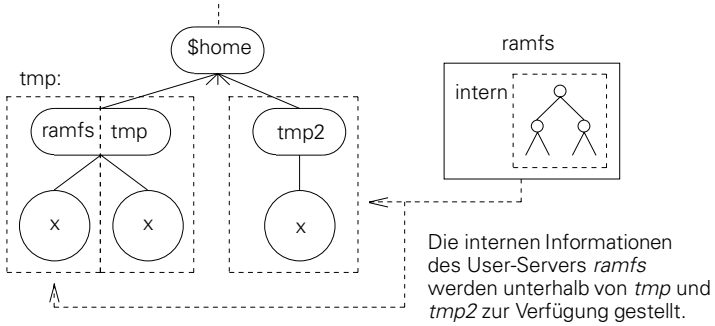
In der nachfolgenden Abbildung ist der Aufbau des Namensraums dargestellt:



Ein weiterer *mount*-Aufruf montiert nun den Service von *ramfs* zusätzlich in das Verzeichnis *tmp*. Wird dabei die Flagge *-b* genutzt, so ist in *tmp* die Datei *x* vom User-Server *ramfs* vor der Datei *x* aus *tmp* zu finden:

```
% mount –b /srv/ramfs tmp
% ls –l tmp
––rw–rw–r–– M 52 bernd bernd       6 May 28 19:53 tmp/x
––rw–rw–r–– M  3 bernd bernd       4 May 28 19:52 tmp/x
% cat tmp/x
ramfs
```

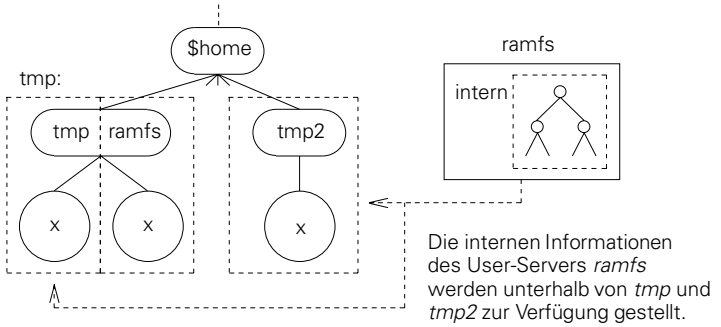
Der neue Aufbau des Namensraums ist nun wie folgt:



Durch die Option *-a* wird der Service von *ramfs* an das Ende von *tmp* montiert:

```
% unmount tmp
% mount –a /srv/ramfs tmp
% ls –l tmp
––rw–rw–r–– M  3 bernd bernd       4 May 28 19:52 tmp/x
––rw–rw–r–– M 53 bernd bernd       6 May 28 19:53 tmp/x
% cat tmp/x
alt
% unmount tmp
% kill ramfs | rc
```

Den so modifizierten Aufbau des Namensraums zeigt die folgende Abbildung:



Ein User-Server braucht die freie Seite der Kommunikations-Pipe nicht in */srv* publik zu machen, sondern kann durch einen internen *mount*(2)-Aufruf das von ihm verwaltete Dateisystem sofort dem aktuellen Namensraum zufügen. Der Server benutzt dabei normalerweise ein Standardverzeichnis als Montagepunkt. Der Anwender kann bei den meisten User-Servern das Montageverzeichnis aber auch durch ein Kommandozeilenargument beim Starten des Servers explizit angeben. Ruft man den User-Server *ramfs* ohne Argumente auf, montiert der Server sein Dateisystem standardmäßig nach */tmp*. Mit der Option *-m* kann aber auch jedes beliebige Verzeichnis im Namensraum als Montagepunkt angegeben werden.

Kommunikation User-Server — Kern

Doch wer verschickt nun aus dem Kern die 9P-Nachrichten an die zuständigen User-Server? Dies ist die Aufgabe des Kernel-Servers *mnt*, #M. Fügt man mit *mount* das Dateisystem eines User-Servers dem aktuellen Namensraum hinzu, so wird der Kernel-Server *#M* als zuständiger Server in die interne Liste eingetragen. Dieser merkt sich zum einen den eigentlich verantwortlichen User-Server und zum anderen einen Filedeskriptor als Kommunikationspfad zum User-Server. 9P-Nachrichten bezüglich des von einem User-Server verwalteten Dateisystems bestehen aus 9P-Funktionsaufrufen innerhalb von *#M*. Er bestimmt den zuständigen User-Server und wandelt den 9P-Funktionsaufruf in eine 9P-Nachricht über die Pipe an den User-Server um.

Nun ist auch das Beispiel zu *bind* und *union-directories* vollständig zu verstehen. Die Kataloge *after* und *before* werden nämlich vom User-Server *fs* verwaltet.

Typische User-Server

Unter Plan 9 gibt es kein *ftp*-Kommando. Ftp-Verbindungen zu anderen Rechnern werden mit Hilfe des User-Servers *ftpfs* durchgeführt. Dieser baut die Verbindung auf, fragt also gegebenenfalls nach Nutzernamen und Paßwort und stellt das Dateisystem des Ftp-Servers als Dateisystem in einem Montagepunkt des lokalen Namensraums zur Verfügung. *Ftpfs* benutzt normalerweise das Verzeichnis */n/ftp* als Montagepunkt. Mit der Flagge *-m* kann aber jedes beliebige Verzeichnis zum Montieren ausgewählt werden. Hierzu ein Beispiel:

```
% mkdir ftp
% ftpfs –m ftp ftp.informatik.uni–osnabrueck.de
    ...
220–    Angewandte Informatik –– Universitaet Osnabrueck
    ...
User[default = dbkuehl]: ftp
331 Guest login ok, send complete e–mail address as password.
Password:
    ...
% cd ftp
% lc
...  adm     pub ...
% cd pub
% lc
...  plan9 ...
% cd plan9
% lc
O9.tar  O9_BOOK_PS.tar intro.ooc  intro.spy  spy.tar
% cp spy.tar /tmp/spy.tar
% cd /tmp
% kill ftpfs | rc
% ll O9_BOOK_PS.tar
––rw–r––r–– M 4 dbkuehl dbkuehl 2304000 Jun 12 13:55 spy.tar
%
```

In dem Beispiel stellt *ftpfs* eine anonyme ftp-Verbindung zum Server *ftp.informatik.uni-osnabrueck.de* her. Durch die Flagge *-m ftp* wird *ftpfs* veranlaßt, seinen Service in das Verzeichnis *ftp* zu montieren, das heißt, *ftpfs* stellt dann das Ftp-Dateisystem an dieser Stelle zur Verfügung. Dateioperationen unterhalb von *ftp* werden vom Kern in 9P-Nachrichten an den User-Server *ftpfs* umgewandelt. Dieser empfängt die Nachrichten, verarbeitet sie und sendet entsprechende Ftp-Befehle an den Ftp-Server. Die Antworten des Ftp-Servers werden von *ftpfs* empfangen und die damit empfangenen Informationen verarbeitet, um schließlich die 9P-Nachricht zu beantworten. Im Beispiel wechselt der Anwender in das Verzeichnis *pub/plan9* und kopiert von dort, also vom *ftp*-Server, eine Datei in das lokale Filesystem.

Es gibt eine Vielzahl weiterer User-Server. Eine komplette Aufzählung enthält das Kapitel 4 der Manual-Seiten. Eine Auswahl interessanter User-Server befindet sich in der folgenden Tabelle:

Name	Datei in /srv	Standard-Montagepunkt	Funktion
8½	*8½.user.pid*	*/mnt/8½* bzw. */dev/windows*	Fenster-System
ftpfs		*/n/ftp*	*ftp*-Verbindung
9660srv	*9660*		ISO-9660-Dateisystem
u9fs			UNIX-Dateisystem
srv	*system*		9P vom fremden System
dossrv	*dos*	*/n/a:, /n/b:, /n/c:*	lokale DOS-Platte und Floppies
fs	*boot*	*/*	Verbindung zum File-Server

Untypische User-Server

Bislang kennen wir nur User-Server, die zur Kommunikation mit den Klienten eine Pipe erzeugen und dann eine Seite der Pipe der Welt bekannt machen oder direkt *mount(2)* aufrufen. Das muß nicht immer so sein. Der *mount*(2)-Aufruf bekommt als Argument einen Filedeskriptor als Verbindung zum User-Server. Hinter dem Filedeskriptor kann sich beliebiges verbergen, so z.B. neben einer Pipe auch eine offene Netzverbindung. In der Plan 9-Release ist ein UNIX-Programm *u9fs*(4) enthalten. Dieses wird auf einem UNIX-Rechner gestartet und horcht auf einen bestimmten Port. Die Aufgabe von *u9fs* ist, als File-Server für das Dateisystem des UNIX-Rechners zu agieren. Ist die Verbindung mit einem Klienten hergestellt, so erwartet *u9fs* 9P-Nachrichten über die Netzverbindung und verarbeitet diese bezüglich des lokalen Dateisystems. Ruft man unter Plan 9 das Kommando *mount*(2) mit einem Filedeskriptor auf, der eine Netzverbindung zu einem *u9fs*-Prozeß ist, so hat man den Namensraum um das UNIX-Dateisystem erweitert:

```
% cd /tmp
% mkdir thor
% srv tcp!thor!9fs
session...post...
% mount —c /srv/tcp!thor!9fs thor
% cd thor
% lc
?thor_root      NextLibrary     lost+found      tftpboot
.               bin             mbox            tmp
..              boot            mnt             tmp_mnt
.bash_history   cdrom           mnt1            tmp_rex
.cshrc          dev             mnt2            usr
.dvexpert       etc             mnt3            var
.login          export          mnt4            vmunix
.logout         home            mnt5            vmunix.old
.profile        home.was        net             vol
.rhosts.ifever  install.files   pcfs
News            kadb            sbin
NextDeveloper   lib             sys
% ps |grep srv
%
```

Das Kommando *srv* stellt eine *tcp*-Verbindung zum Rechner *thor* her und hinterlegt diese in der Datei */srv/tcp!thor!9fs*. Damit ist nach dem *mount*-Aufruf das Dateisystem von *thor* unterhalb des Katalogs *thor* sichtbar.

Im weiteren Verlauf werden wir diese speziellen Fälle nicht weiter erwähnen, sondern immer von einer Pipe als Kommunikationsleitung ausgehen.

4.4 Zusammenfassung

Abschließend zu *mount*, *bind*, Kernel- und User-Servern ist noch einmal zu bemerken, daß User-Server im Gegensatz zu Kernel-Servern eigenständige Prozesse sind. Mit beiden wird über 9P verhandelt. Bei Kernel-Servern wird pro 9P-Nachrichten-Typ eine zugehörige Funktion im Server aufgerufen. Für User-Server agiert der

Kernel-Server *mnt*, #M, als Platzhalter im Kern. Dieser bekommt die 9P-Nachrichten für User-Server und verteilt diese dann an die entsprechenden User-Server. Dabei werden die 9P-Nachrichten über einen beliebigen File-Deskriptor an die User-Server geschickt. Normalerweise ist dies eine Seite einer Pipe; dies kann aber auch z.B. eine Netzverbindung sein.

Möchte man als Entwickler eine neue Funktionalität als Dateisystem für das Einbinden in einen Namensraum anbieten, so steht man vor der Wahl, einen neuen Kernel- oder User-Server zu entwickeln. Nicht alle anfallenden Ideen für einen Server sind über die Idee der Kernel-Server zu programmieren. Diese sind nur für eine systemnahe Funktionalität sinnvoll, da das Entwickeln eines neuen Kernel-Servers einige Schwierigkeiten bereiten kann. So gestaltet sich das Testen des neuen Kernel-Servers schwierig, da dessen Code Teil des Betriebssytems ist. Ein Programmierfehler kann somit das ganze System zum Erliegen bringen. Auch wird der Kern mit jedem neuen Kernel-Server immer größer. Programmierfehler in User-Servern haben dagegen keinen Einfluß auf das gesamte System, da sie eigenständige Prozesse sind. Die Programmierfehler würden im schlimmsten Fall lediglich in einem Programmabsturz des User-Servers enden. Daher wird man in der Regel neue Server als User-Server implementieren. Eine kurze Einführung in die Entwicklung eigener Server enthalten die Kapitel 5.3-5.6.

4.5 *import*

Eine weitere Möglichkeit, den Namensraum zu erweitern, bietet *import*(1). Angenommen, man hat folgendes Problem: Auf dem aktuellen Rechner steht nicht das Protokoll *udp* zur Verfügung, obwohl es gerade gebraucht wird.

```
% cd /net
% lc
cs    ether il    tcp
```

Mit *import* kann man eine Datei oder einen Katalog von einem anderen Rechner auf seinem lokalen Rechner an einer gewünschten Stelle im Namensraum sichtbar machen. Damit kann man sich das erforderliche Protokoll einfach von einer anderen Maschine holen:

```
% import newage /net/udp
import: can't mount /net/udp: file does not exist:
% import –a newage /net
% lc
cs    cs    dns   ether ether il    il    tcp   tcp   udp
% lc –n
il    tcp   ether cs    il    tcp   udp   ether cs    dns
```

Die Syntax von *import* ist

```
import [-a] [-b] [-c] system file [mountpoint]
```

Die Flaggen *-a* (after), *-c* (create) und *-b* (before) dienen dem gewohnten Umgang mit *union-directories*, und *system* ist der Rechnername, von dem man den Katalog oder die Datei *file* importieren möchte. Optional kann man den lokalen Montagepunkt angeben. Wird er weggelassen, so wird *file* an der gleiche Stelle wie auf

dem Rechner *system* im Namensraum sichtbar. Daher funktioniert im obigen Beispiel der erste Versuch nicht, da es im lokalen Katalog */net* keinen Katalog *upd* zum Verdecken gab.

4.6 Aufbau des Namensraums während des boot-Vorgangs

Nachdem der Betriebssystemkern geladen wurde, setzt dieser die Umgebungsvariablen *cputype* und *terminal*. Deren Inhalt bestimmt die aktuelle Hardware-Architektur. Anschließend bindet der Kern das Dateisystem des Kernel-Servers *root*, #/, in das bislang leere Wurzelverzeichnis »/«. Der Namensraum besteht jetzt nur aus der von *root* zur Verfügung gestellten Datei */boot* und leeren Katalogen: */, /dev, /env, /proc* und */net*.

Die Kataloge *dev*, *env*, *proc* und *net* sind Platzhalter, um spätere Aufrufe von *bind* zu ermöglichen. Die Datei */boot* enthält einen für die aktuelle Architektur ausführbaren Code. Der Kern erzeugt nun einen Prozeß, dessen einzige Aufgabe es ist, */boot* auszuführen. Die Kommandozeilenargumente für */boot* sind unterschiedlich, je nach der aktuellen Rechner-Architektur. Näheres dazu wird auf der Manual-Seite *boot(8)* dargelegt. Im folgenden wird der weitere Fortgang an einem Terminal näher besprochen.

Das vom Kern gestartete Programm */boot* fragt den Anwender nach seinem User-Namen und Paßwort und stellt dann eine Verbindung zum File-Server her. Daraufhin montiert */boot* das Wurzelverzeichnis des File-Servers vor den aktuellen Inhalt von »/« und stellt die Verbindung zum File-Server in */srv/boot* für weitere *mount*-Aufrufe zur Verfügung. Durch das Ausführen von */$cputype/init -t* geht der Prozeß */boot* zu Ende.

Init setzt als erstes die Umgebungsvariable *objtype* auf den Inhalt von *cputype* (zum Beispiel *386* oder *sparc*). Dann wird der Inhalt von *service* (zum Beispiel *terminal* oder *cpu*) und von *timezone* gesetzt. Dabei enthält *timezone* eine Kopie der Datei */adm/timezone/local*. Daraufhin führt *init* die Funktion *newns*(2) mit dem Inhalt von *user* als erstes und *0* als zweites Argument aus. *Newns()* setzt den Inhalt der Umgebungsvariablen *user* und *home* und löscht dann durch einen Aufruf von *rfork*(2) mit dem Argument RFENVGIRFCNAMEG den aktuellen Namensraum, das heißt, danach ist nur noch das Dateisystem vom Kernel-Server *root* im Namensraum vorhanden. *Newns* baut nun durch Abarbeiten von */lib/namespace* einen neuen Namensraum auf. Die Datei */lib/namespace* hat zum Beispiel folgenden Inhalt:

```
# root
mount –a #s/boot /

# kernel devices
bind #c /dev
bind #d /fd
bind –c #e /env
bind #p /proc
bind –c #s /srv
```

```
# standard bin
bind /$cputype/bin /bin
bind -a /rc/bin /bin

# networks
mount -b /srv/cs /net
bind -b #l /net
bind -b #Iudp /net
bind -b #Itcp /net
bind -b #Iil /net
mount -a /srv/dns /net

# local fs
mount -c /srv/kfs /n/kfs

cd /usr/$user
```

Newns() liest zeilenweise die Datei und behandelt Zeilen, die mit »#« beginnen, als Kommentare. Zeilen, die aus einem *mount*- oder *bind*-Aufruf bestehen, werden von *newns()* in die einzelnen Argumente zerlegt, um mit diesen die zugehörige C-Funktion auszuführen.

Mit Hilfe des ersten Kommandos aus */lib/namespace* wird das Dateisystem des File-Servers wieder im Wurzelverzeichnis »/« sichtbar gemacht. Durch die nachfolgenden *bind*s und *mount*s wird der Namensraum nun wie gewünscht aufgebaut.

Nach dem Ausführen von *newns()* startet *init* eine Shell (*rc*). Diese führt zuerst die Kommandos in */rc/bin/termrc* und dann die in */usr/$user/lib/profile* aus. Für einen PC als Terminal enthält */rc/bin/termrc* beispielsweise folgende Befehle zur Manipulation des Namensraums:

```
/bin/bind -a '#H' /dev >/dev/null
/bin/bind -a '#w' /dev >/dev/null
/bin/bind -a '#f' /dev >/dev/null
/bin/bind -a '#t' /dev >/dev/null
    ...
/bin/dossrv
```

So wird hier zum Beispiel der Kernel-Server *floppy,* #f, der eine Kontrolle über die am PC angeschlossenen Floppy-Laufwerke bietet, dem Namensraum hinzugefügt. Die Funktionalität der anderen Kernel-Server ist Kapitel 3 der Manual-Seiten zu entnehmen. *Dossrv* dagegen ist ein für PCs typischer User-Server. Er hinterlegt die Datei */srv/dos* als Kommunikationspfad. Er bietet dem Anwender die Möglichkeit des Zugriffs auf ein DOS-Dateisystem, seien es sowohl Festplatten, als auch Floppy-Disks. Ein genaue Beschreibung liefert die Manualseite *dossrv*(4).

In der Datei */usr/$user/lib/profile* kann jeder Anwender die Kommandos eintragen, welche er beim Systemstart ausgeführt haben möchte, so etwa auch Befehle zur Manipulation des Namensraums. Eine typische *profile*-Datei sieht wie folgt aus:

```
fn ll { ls -l $* }
fn ld { ls -d $* }
fn lld { ls -dl $* }
```

```
# make my own things visible
bind –a $home/bin/rc /bin
bind –b $home/bin/$cputype /bin
bind –c $home/tmp /tmp
bind –b /bin/fb /bin

# printer entries
bind –bc $home/lpspool /sys/lib/lp/log
bind –bc $home/lpspool /sys/lib/lp/queue
bind –bc $home/lpspool /sys/lib/lp/tmp

font = /lib/font/bit/pelm/euro.8.font

switch($service){
case terminal
    prompt=('term% ' '  ')
    fn term%{ $* }
    swap=`{ swap $home/swap }       # allocate swap aerea
    exec 8½ –s –f $font –i $home/bin/init
case cpu
...
}
```

Auch hier wird der Namensraum durch verschiedene Aufrufe von *bind* umgebaut. So wird zum Beispiel das für den Anwender schreibbare Verzeichnis *$home/tmp* modifizierbar über das für den Anwender schreibgeschützte Verzeichnis */tmp* gebunden. Abschließend wird an einem Terminal die Fensteroberfläche 8½ aufgerufen, und das System ist gestartet.

4.7 Zusammenfassung

Eine Zusammenfassung des Aufbaus des Namensraums während des boot-Vorgangs liefert die folgende Tabelle:

Kern:
- setzt *cputype* und *terminal*
- bindet den Server *root*, #/, nach */*
- erzeugt ersten Prozeß:
  — führt */boot* aus

/boot:
- erfragt Nutzernamen und Paßwort
- stellt eine Verbindung zum File-Server her
- führt *init* aus

---

init:
- setzt verschiedene Umgebungsvariablen
- *newns()*:
  — Abarbeitung von */lib/namespace*
- startet Shell *rc*:
  — Abarbeitung von */rc/bin/termrc*
  — Abarbeitung von */usr/$user/lib/profile*

4.8 Ein typischer Namensraum

Nachdem die Dateien */lib/namespace*, */rc/bin/termrc* und */usr/$user/lib/profile* abgearbeitet worden sind, sollte der Namensraum einigen Konventionen entsprechend aufgebaut sein. Im folgenden werden diese Konventionen, welche Dateien wo im Dateibaum zu finden sind, näher beschrieben.

Als erstes wird das Aussehen des Dateisystems auf dem File-Server beschrieben, das heißt der Zustand des Namensraums nach dem Abarbeiten der ersten Zeilen von */lib/namespace*:

/	das Wurzelverzeichnis
/adm	In und unterhalb dieser Stelle sind alle Informationen zur Verwaltung des File-Servers gesammelt.
/adm/users	Dieses Verzeichnis enthält eine Liste der dem File-Server bekannten Anwender. Näheres siehe *users*(*6*).
/adm/keys	Authentifizierungs-Schlüssel der Anwender
/adm/timezone	Verzeichnis mit Zeitzonendateien
/adm/timezone/timezone	Zeitzonen-Beschreibung der lokalen Zeitzone; dies ist eine Kopie einer anderen Datei aus diesem Verzeichnis.
/bin	leeres Verzeichnis; Platzhalter für spätere Aufrufe von *bind*
/dev	analog zu */bin*
/env	analog zu */bin*
/fd	analog zu */bin*
/net	analog zu */bin*
/proc	analog zu */bin*
/srv	analog zu */bin*
/tmp	analog zu */bin*
/mnt	Dieses Verzeichnis enthält Montagepunkte für Applikationen.
/n	Dieses Verzeichnis enthält Montagepunkte für Dateisysteme, die von anderen Rechnern importiert werden.
/68020 */386* */sparc* */960* */mips*	Für jede von Plan 9 unterstützte Architektur (68020, 386, sparc, 960 und mips) existiert ein gleichnamiges Verzeichnis in »/«. Eine genaue Beschreibung des Dateibaums unterhalb dieser Verzeichnisse wird nur für */mips* vorgenommen. Der Aufbau ist für die anderen Architekturen analog.
/mips/init	Dieses Programm wird beim Booten aufgerufen.

/mips/bin	Dieses Verzeichnis enthält die Binär-Dateien für die MIPS-Architektur.
/mips/bin/aux */mips/bin/games* *etc.*	Diese Unterverzeichnisse von */mips/bin* enthalten Hilfsprogramme und sammeln zusammengehörige Programme an einer Stelle.
/mips/lib	Dieses Verzeichnis enthält die vom Lader verwendeten Bibliotheken.
/mips/include	Dieses Verzeichnis enthält die vom C-Compiler verwendeten Include-Dateien.
*/mips/9**	Die Dateien in */mips*, welche mit einer 9 beginnen, sind Binaries des Betriebssystems Plan 9.
/mips/mkfile	Diese Datei wird von *mk*(*1*) verwendet, wenn der Inhalt von *$objtype* mips ist.
/rc	Analog zu den von der Architektur abhängigen Verzeichnissen */mips*, */sparc* usw. sind im Verzeichnis */rc* Dateien bezüglich der Plan 9 Shell *rc* gesammelt.
/rc/bin	Shell-Skripten
/rc/lib	Shell-Bibliotheken
/lib	Das Verzeichnis */lib* enthält verschiedene Arten von Daten.
/lib/bible */lib/chess* */lib/sky* *etc.*	verschiedene Datenbanken
/lib/ndb	die Netzwerkdatenbank, welche von der Netzwerk-Software benutzt wird.
/lib/namespace	die Datei, welche von *newns*(*2*) zum Errichten des Namensraums benutzt wird.
/lib/font/bit	verschiedene Bitmap Font-Dateien
/sys	die System-Software
/sys/include	von der Architektur abhängige C-Include-Dateien
/sys/include/alef	*alef* Include-Dateien
/sys/lib	
/sys/lib/acid	verschiedene Module für den Debugger *acid*
/sys/lib/troff	Makros und Schriften für *troff*
/sys/man	Manual-Seiten
/sys/doc	Dokumentation
/sys/log	verschiedene *log*-Dateien
/sys/src	die Quellen von Plan 9
/mail	Dieses Verzeichnis enthält die Dateien für das elektronische Mailen.
/acme	verschiedene Hilfsdateien für *acme*
/cron	verschiedene Hilfsdateien für *cron*; Näheres siehe *cron*(*8*)

Nach dem kompletten Abarbeiten von */lib/namespace*, */rc/bin/termrc* und */usr/$user/lib/profile* haben die folgenden Verzeichnisse den angebenen Inhalt:

/bin Das Verzeichnis */bin* ist ein *union*-Verzeichnis. Es ist zusammengesetzt aus */$objtype/bin*, */rc/bin*, *$home/$objtype/bin* usw. In diesem Verzeichnis sind also immer die zur aktuellen Architektur gehörigen Binär-Dateien zu finden.

/dev Auch das Verzeichnis */dev* ist ein *union*-Verzeichnis. Es beinhaltet unter anderem die Dateien des Consolen-Kernel-Servers *cons*(3) und die des Kernel-Servers *bit*(3), welcher Zugang zur Maus und zum Graphik-Bildschirm bietet.

/env An dieser Stelle ist das Dateisystem des Kernel-Servers *env*(3) montiert. Dieser verwaltet die Umgebungsvariablen.

/net Nach */net* sind die Dateisysteme von all den Kernel-Servern gebunden, die sich mit dem Netzwerk beschäftigen.

/proc An dieser Stelle ist das Dateisystem des Kernel-Servers *proc*(3) montiert, der Informationen zu den aktuellen Prozessen zur Verfügung stellt.

/fd Standard-Montagepunkt für das Dateisystem des Kernel-Servers *dup*(3). Dieser stellt Informationen über die offenen Filedeskriptoren zur Verfügung.

/srv An dieser Stelle ist das Dateisystem des Kernel-Servers *srv*(3) montiert.

/srv/boot Die Kommunikationsleitung zum File-Server

/mnt/8½ Montagepunkt für das Fenstersystem 8½

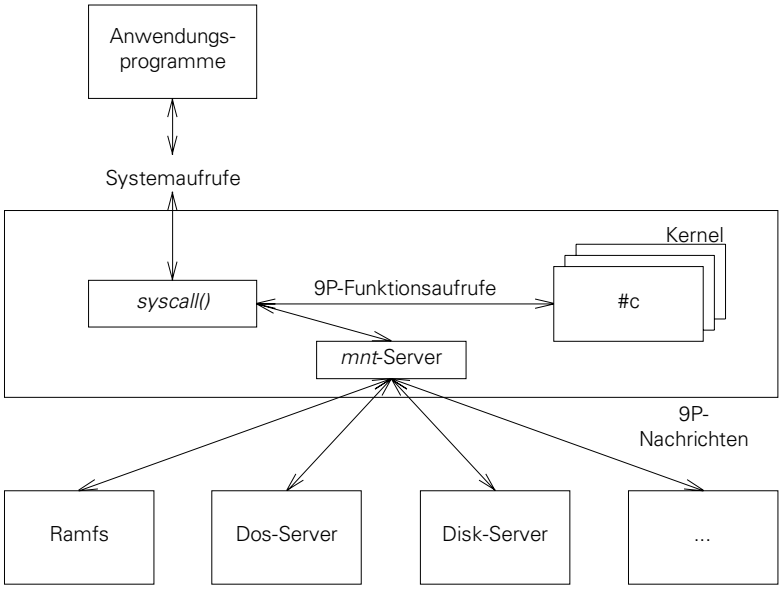
/mnt/term Montagepunkt für den Namensraum des Terminals nach einem *cpu*-Kommando

/tmp An diese Stelle sollte *$home/tmp* oder das interne Dateisystem von *ramfs*(4) gebunden sein.

5
Server und 9P

Unter Plan 9 besitzt jeder Prozeß einen Namensraum, d. h. seine lokale Sicht auf den Dateibaum. Dieser besteht aus Teilbäumen. Eine Vielzahl von Servern verwaltet dabei die Dateien in den einzelnen Teilbäumen des ganzen Dateibaums. Generell gibt es zwei Klassen von Servern:

Im Kern ist ein vordefinierter, fixierter Namensraum vorhanden, dessen Teilbäume von den sogenannten *Kernel*-Servern verwaltet werden. Die Kernel-Server sind daher fest im Kern eingebrannt. Die von ihnen verwalteten Namensräume können mit Hilfe von *bind*(1,2) an eine beliebige Stelle im Namensraum eingebunden werden. Eine Einführung zum Kernel-Server bietet das Kapitel 4. Dort werden auch einige typische Kernel-Server und deren Verwendung vorgeführt.



Die andere Serverklasse besteht aus den *User*-Servern. Sie sind im Gegensatz zu Kernel-Servern eigenständige Prozesse. Der von ihnen verwaltete Dateibaum wird einem Namensraum durch *mount* hinzugefügt. Auch hier bietet das Kapitel 4 eine Einführung. Im weiteren wird der Stoff aus Kapitel 4 vorausgesetzt.

Der Kern einer Plan 9-Maschine und die einzelnen Server verhandeln miteinander mit Hilfe des Protokolls 9P. Findet ein Systemaufruf statt, der eine Dateioperation beinhaltet, so wandelt der Kernel den Systemaufruf in 9P-Transaktionen um. Ist der zuständige Server ein Kernel-Server, so werden in dem Server 9P-Funktionen aufgerufen. User-Server dagegen werden im Kern durch den Kernel-Server *mnt*, #M, vertreten. Dieser empfängt die 9P-Funktionsaufrufe bezüglich einer von einem User-Server verwalteten Datei und schickt dem User-Server die entsprechenden 9P-Nachrichten.

In diesem Kapitel geht es um die Technik, die hinter der Plan 9-Server-Technologie steckt, d.h. um einen genauen Blick in das Protokoll 9P, und um eine Anleitung zur Entwicklung neuer Server. Eine Einführung in die Benutzung und eine Übersicht über die vorhandenen Server bietet das Kapitel 4.

5.1 Das Protokoll 9P

Ein Plan 9-User-Server bietet seinen Klienten, normalerweise dem Kern (»*#M*«) einer Plan 9-Maschine, ein oder mehrere Dateisysteme über eine bidirektionale Kommunikationsleitung an. Auf dieser Verbindung wird mit 9P-Nachrichten verhandelt. Mehrere Klienten können sich dabei dieselbe Verbindung teilen. Somit können beliebig viele Klienten nebeneinander auf den Dateisystemen eines Servers arbeiten. Der User-Server beantwortet 9P-Anfragenachrichten von Klienten bezüglich Operationen auf Objekte, Dateien und Verzeichnisse in seinen Dateisystemen, so z.B. Dateien zu kreieren, zu entfernen, zu lesen, zu schreiben oder durch das Dateisystem zu wandern.

Bei einem Kernel-Server werden vom Kern zu den einzelnen 9P-Nachrichten analoge Funktionen im Kernel-Server aufgerufen.

Das Protokoll 9P im Plan 9-Release von 1995 definiert 15 Nachrichten, die vom Klienten zu einem Server geschickt werden können, die sogenannten *T-Nachrichten*, und 16 Nachrichten, die vom Server an den Klienten geschickt werden können, die R-Nachrichten. Der Klient sendet dem Server 9P T-Nachrichten und dieser antwortet mit R-Nachrichten des gleichen Typs oder mit der R-Nachricht *Rerror* bei einem Fehler. Dies darf auch asynchron geschehen, d.h., der Klient schickt dem Server mehrere T-Nachrichten, bevor dieser eine davon beantwortet. Eine T-Nachricht und die zugehörende R-Nachricht bilden zusammen eine Transaktion. 9P ist, wie im folgenden gezeigt wird, ein zustandsbehaftetes Protokoll.

Jede 9P-Nachricht besteht aus einer Sequenz von Bytes, wobei das erste Byte den Typ der Nachricht definiert. Im folgenden sind alle 9P-Nachrichten aufgeführt, wobei hinter dem Nachrichtentyp (z.B. *Tnop*) die Parameter der Nachricht und in eckigen Klammern die Anzahl ihrer Bytes folgen:

```
Tnop     tag[2]
Rnop     tag[2]
Tsession tag[2] chal[8]
Rsession tag[2] chal[8] authid[28] authdom[48]
Rerror   tag[2] ename[64]
Tflush   tag[2] oldtag[2]
Rflush   tag[2]
Tattach  tag[2] fid[2] uid[28] aname[28] ticket[72] auth[13]
Rattach  tag[2] fid[2] qid[8] rauth[13]
Tclone   tag[2] fid[2] newfid[2]
Rclone   tag[2] fid[2]
Tclwalk  tag[2] fid[2] newfid[2] name[28]
Rclwalk  tag[2] fid[2] qid[8]
Twalk    tag[2] fid[2] name[28]
Rwalk    tag[2] fid[2] qid[8]
```

```
Topen    tag[2] fid[2] mode[1]
Ropen    tag[2] fid[2] qid[8]
Tcreate  tag[2] fid[2] name[28] perm[4] mode[1]
Rcreate  tag[2] fid[2] qid[8]
Tread    tag[2] fid[2] offset[8] count[2]
Rread    tag[2] fid[2] count[2] pad[1] data[count]
Twrite   tag[2] fid[2] offset[8] count[2] pad[1] data[count]
Rwrite   tag[2] fid[2] count[2]
Tclunk   tag[2] fid[2]
Rclunk   tag[2] fid[2]
Tremove  tag[2] fid[2]
Rremove  tag[2] fid[2]
Tstat    tag[2] fid[2]
Rstat    tag[2] fid[2] stat[116]
Twstat   tag[2] fid[2] stat[116]
Rwstat   tag[2] fid[2]
```

Die zwei-, vier- und acht-Byte Felder enthalten vorzeichenlose Integer in *little-endian* Reihenfolge, d.h., die weniger signifikanten Bytes kommen zuerst. Datenfelder für Namen sind 28 Byte lang und enthalten ein den String abschließendes Nullbyte. Bis auf das Nullbyte sind innerhalb des Namens alle Zeichen erlaubt, die auch in Dateinamen vorkommen dürfen. Das sind unter Plan 9 alle Zeichen außerhalb von hexadezimal 00-1F und 80-9F außer Leerzeichen und Slash (»/«).

Ein Kernel-Server realisiert pro 9P-Nachrichten-Typ eine Funktion, z.B. *rattach()* für die *attach*-Nachricht, welche dann vom Kern aufgerufen wird. Der Server #M kann im Namensraum einen Prozeß als Server vertreten und schickt dann 9P-Nachrichten mit der in obiger Tabelle definierten Bytefolge an den Prozeß.

Ein User-Server besteht immer aus 15 Prozeduren, die das verwaltete Dateisystem manipulieren und pro Klient zustandsbehaftet sind. 9P dient dazu, diese Prozeduren als RPC zu verschicken. Trifft eine 9P-Nachricht beim User-Server ein, so wird der Typ der Nachricht bestimmt. Pro Typ gibt es nun eine zugehörige Prozedur, die dann mit den empfangenen Daten aufgerufen wird. In der Funktion werden die Daten verarbeitet und die Antwort-Nachricht bestimmt. Dies kann eine R-Nachricht gleichen Typs oder eine Rerror-Nachricht sein. Eine Rerror-Nachricht enthält in *ename* eine Fehlermeldung im Klartext. Der Kernel-Server kann am Typ der Antwort erkennen, ob eine Fehler-Nachricht vom User-Server geliefert worden ist, und beantwortet seinen 9P-Funktionsaufruf mit dem gleichen Fehler.

Tag, qid und fid

Jede 9P-Nachricht enthält ein Etikett (*tag*). Für eine T-Nachricht, die der Klient verschicken möchte, erfindet er zur späteren Identifikation der Nachricht eine positive ganze Zahl als *tag*. Eine Antwort auf diese T-Nachricht muß den gleichen *tag* beinhalten. Da nicht synchron geantwortet werden muß, können mehrere Antworten des Servers auf T-Nachrichten ausstehen. Anhand der *tag* wird die Antwort auf eine T-Nachricht identifiziert. Deshalb hat der Klient dafür zu sorgen, daß bei zwei ausstehenden Antworten auf derselben Verbindung die *tags* unterschiedlich sind.

Der Server hat für jedes Objekt in seinem Dateisystem intern eine eindeutige Bezeichnung namens *qid*. Eine *qid* besteht aus einem internen Namen und einer internen Versionsnummer für das Objekt. Wird in einem Server ein Objekt erzeugt, so belegt der Server dieses mit einer eindeutigen Erkennungszahl, der *qid*. Die Versionsnummer erhöht sich bei jeder Veränderung der Ressource. Die *qid* einer Datei ist in etwa analog zu den *i-nodes* in einem UNIX-Filesystem.

Der Klient identifiert eine Ressource des Servers mit einer eindeutigen *fid*, d. h. einer kleinen Integerzahl. Der Server erfährt durch *Tattach* oder *Tclone* die gewünschte *fid* und koppelt sie mit der Ressource bis der Klient diese Bindung wieder löst. Danach kann der *fid*-Wert wiederverwendet werden. Anhand der aktiven *fid* erfolgen dann die einzelnen Dateioperationen.

Die 9P-Nachrichten im einzelnen

9P ist ein zustandsbehaftetes Protokoll. Wird z.B. eine *Tread*-Nachricht des Klienten an den Server nicht durch eine vorhergehende *open*-Transaktion vorbereitet, so antwortet dieser mit einer Fehlerantwort. Wird die *Tread*-Nachricht dagegen vorbereitet, so liefert der Server die gewünschten Daten.

Im folgenden werden die 9P-Nachrichten in logischer Reihenfolge aufgeführt und erläutert. Sie sind außerdem im Kapitel 5 der Manual-Seiten ausführlich beschrieben.

Nop

```
Tnop    tag[2]
Rnop    tag[2]
```

Die *nop*-Anfrage ist die ''leere'' Anweisung. Sie dient z.B. zur Synchronisation der Kommunikation. Die *tag* bei der *nop*-Anfrage sollte die Konstante NOTAG (0xFFFF) sein.

Session

```
Tsession    tag[2] chal[8]
Rsession    tag[2] chal[8] authid[28] authdom[48]
```

Attach

```
Tattach tag[2] fid[2] uid[28] aname[28] ticket[72] auth[13]
Rattach tag[2] fid[2] qid[2] rauth[13]
```

Tattach koppelt *fid* mit dem in *aname* angegebenen Dateisystem des Servers. Bei Erfolg repräsentiert *fid* die Wurzel des im Server ausgewählten Dateisystems. Der Server bekommt über *uid* den Benutzer übergeben. Damit der Server dem Klienten die Identifikation glaubt, wird zuvor mit einer *session*-Transaktion eine Umgebung aufgebaut, in der sich der Klient ausweisen muß. Der Ausweis besteht aus *ticket* und *auth* in *Tattach*. Die Felder *chal*, *authid* und *authdom* werden zur Erzeugung von *ticket* und *auth* benötigt. Die genaue Vorgehensweise ist recht komplex und wird in *auth* aus Kapitel 6 der Manual-Seiten erklärt. Die Legalität eines Zugriffs innerhalb des Serves kann dann immer an der Klienten-ID gemessen werden, die aus *Tattach* stammt.

Sobald erneut *Tsession* geschickt wird, bricht die Umgebung zusammen, d. h., etwa ausstehende 9P-Nachrichten und I/O-Verbindungen werden abgebrochen. Alle 9P-Nachrichten zwischen zwei *session*-Anfragen bilden eine sogenannte *session*. Die *tags* und *fids* innerhalb einer *session* müssen eindeutig sein. Solange nicht erneut *Tsession* geschickt wird, sind mehrere *Tattach* und damit mehrere Verbindungen zum Server möglich.

Clone

```
Tclone  tag[2] fid[2] newfid[2]
Rclone  tag[2] fid[2]
```

Die *clone*-Anfrage enthält im Feld *fid* eine existente *fid* und im Feld *newfid* eine unbenutzte *fid*. Der Klient möchte, daß die *fid newfid* die gleiche Datei oder das gleiche Verzeichnis wie *fid* repräsentiert. *Rclone* signalisiert Erfolg dadurch, daß die ursprüngliche *fid* enthalten ist.

Clunk

```
Tclunk  tag[2] fid[2]
Rclunk  tag[2] fid[2]
```

Die *clunk*-Anfrage informiert den Server darüber, daß der Klient nicht mehr an *fid* interessiert ist, d.h., daß *fid* nicht länger ein Objekt im Dateisystem des Servers repräsentieren soll.

Walk

```
Twalk   tag[2] fid[2] name[28]
Rwalk   tag[2] fid[2] qid[8]
```

Der Klient sucht in dem durch *fid* identifizierten Katalog *name*. Wenn dieser Name existiert, bezieht sich *fid* anschließend darauf, und *qid* ist die Server-interne *qid* der Datei.

Clwalk

```
Tclwalk tag[2] fid[2] newfid[2] name[28]
Rclwalk tag[2] fid[2] qid[8]
```

Die *clwalk*-Anfrage ist eine Kombination einer *clone*-Anfrage, gefolgt von einer *walk*-Anfrage bezüglich *newfid* und *name*.

Error

```
Rerror  tag[2] ename[28]
```

Es gibt keine *error*-Anfrage-Nachricht. Die *error*-Antwort wird benutzt, um einen Fehlerstring *ename* zurückzuliefern, welcher die Art des aufgetretenen Fehlers beschreibt. Die *error*-Antwort erfolgt an Stelle einer korrekten R-Nachricht. Die *tag* der Antwort ist die der zugehörigen T-Nachricht.

Flush

```
Tflush  tag[2] oldtag[2]
Rflush  tag[2]
```

Wenn eine ausstehende Antwort auf eine T-Nachricht nicht länger gebraucht wird, so kann der Klient eine *Tflush*-Nachricht an den Server schicken. *Oldtag* identifiziert die entsprechende T-Nachricht. Der Server muß eine *flush*-Anfrage unverzüglich verarbeiten und darf die identifizierte T-Nachricht nicht mehr beantworten.

Open

```
Topen   tag[2] fid[2] mode[1]
Ropen   tag[2] fid[2] qid[8]
```

Die *open*-Anfrage öffnet die Datei oder das Verzeichnis, das durch *fid* repräsentiert wird, für I/O Zugriffe durch nachfolgende *read*- und *write*-Anfragen. Das *mode*-Feld beschreibt die Art des I/O Zugriffs. Dabei bedeuten 0, 1, 2 und 3 Lese-, Schreib-, Lese- plus Schreib- sowie Ausführungszugriff. Der Zugriffswunsch wird gegen die Rechte bezüglich der Datei abgeglichen.

Create

```
Tcreate tag[2] fid[2] name[28] perm[4] mode[1]
Rcreate tag[2] fid[2] qid[8]
```

Die *create*-Anfrage dient zum Erzeugen neuer Dateien oder Verzeichnisse. Der Klient möchte im Katalog *fid* unter dem Namen *name* und mit dem Zugriffsschutz *perm* ein neues Objekt erzeugen und anschließend wie bei *open* mit *mode* darauf zugreifen können.

Read

```
Tread   tag[2] fid[2] offset[8] count[2]
Rread   tag[2] fid[2] count[2] pad[1] data[count]
```

Die *Tread*-Nachricht ist eine Anfrage an den Server, von dem Objekt, das durch *fid* repräsentiert wird, maximal *count* Bytes beginnend ab *offset* Byte hinter dem Dateianfang zu lesen. In der Antwort sind im Feld *data* die angeforderten Daten. *Count* beschreibt in der Antwort, wie viele Bytes sich im Datenfeld befinden.

Write

```
Twrite  tag[2] fid[2] offset[8] count[2] pad[1] data[count]
Rwrite  tag[2] fid[2] count[2]
```

Die *write*-Anfrage veranlaßt den Server, in der Datei, die durch *fid* repräsentiert wird, *count* Bytes aus dem Datenfeld *data* in der Datei ab der Position *offset* zu speichern. In der Antwort beschreibt *count* die Anzahl gespeicherter Bytes. Es liegt ein Fehler vor, wenn nicht alle Daten geschrieben wurden.

Remove

```
Tremove tag[2] fid[2]
Rremove tag[2] fid[2]
```

Die *remove*-Anfrage entfernt im Server die Datei, die durch *fid* repräsentiert wird. Darüber hinaus wird die *fid* wie bei einer *clunk*-Anfrage freigegeben.

Stat

```
Tstat   tag[2] fid[2]
Rstat   tag[2] fid[2] stat[116]
```

Die *stat*-Anfrage veranlaßt den Server, in der Antwort im *stat*-Feld nähere Informationen über die durch *fid* repräsentierte Datei zu liefern. Das *stat*-Feld ist wie folgt eingeteilt:

Feld		Inhalt
name	[28]	Dateiname
uid	[28]	Besitzer der Datei
qid	[28]	Gruppe des Besitzers
qid.path	[4]	Pfadname der qid der Datei im Server
qid.vers	[4]	Versionsnummer der qid der Datei im Server
mode	[4]	Zugriffsrechte und Flaggen der Datei
atime	[4]	Datum des letzten Zugriffs
mtime	[4]	Datum der letzten Änderung
length	[8]	Länge der Datei
type	[2]	nur für den Kern
dev	[2]	nur für den Kern

Wstat

```
Twstat  tag[2] fid[2] stat[116]
Rwstat  tag[2] fid[2]
```

Die *wstat*-Anfrage veranlaßt den Server, die Informationen über die Datei oder das Verzeichnis, welches durch *fid* repräsentiert wird, zu ändern. Das *stat*-Feld ist analog zur *Rstat*-Antwort. Es dürfen allerdings nicht alle Einträge geändert werden.

5.2 9P in Aktion

Eine erste Hilfe zum Verständnis von 9P bietet der bereits erwähnte User-Server *ramfs*. Beim Start von *ramfs* wird durch die Option *-d* (*debug*) die Ausgabe der eintreffenden und ausgehenden 9P-Nachrichten verlangt:

```
ramfs [-i] [-s] [-d] [-m mountpoint]
```

Hierbei gilt:

- -i Deskriptoren 0 und 1 bilden den Kommunikationskanal
- -d Ausgabe der 9P-Nachrichten
- -s nicht implizit montieren, sondern Dateiverbindung unter */srv/ramfs* speichern
- -m implizit auf *mountpoint* montieren (statt auf */tmp*)

Durch die *debug*-Option eignet sich *ramfs* daher für Experimente mit 9P.

Start von *ramfs*

Starten von *ramfs* mit der *debug*-Flagge:

```
% ramfs –d
ramfs:<–Tattach tag 5 fid 176 uname dbkuehl aname  auth
ramfs:–>Rattach tag 5 fid 176 qid 0x80000000|0x0
```

Ramfs montiert hier seinen Service nach */tmp*. Dabei verzichtet es auf eine Authentifizierung, d.h., daß intern die C-Funktion *mount* anstatt *amount* verwendet wird. Daher findet lediglich eine *attach*- und keine vorhergehende *session*-Transaktion statt. Auf die Wurzel des Dateisystems in *ramfs* wird mit der *fid* 176 verwiesen.

Verzeichnis ausgeben

Mit *ls* wird nun das Verzeichnis */tmp* ausgegeben:

```
% ls /tmp
ramfs:<–Tclone tag 5 fid 176 newfid 173
ramfs:–>Rclone tag 5 fid 176
ramfs:<–Tstat tag 5 fid 173
ramfs:–>Rstat tag 5 fid 173 stat '.' 'dbkuehl' 'dbkuehl'
    q 0x80000000|0x0
    m 020000000775 at 1846833249 mt 1846833249 l 0 t 1 d 19269
ramfs:<–Tclunk tag 5 fid 173
ramfs:–>Rclunk tag 5 fid 173
ramfs:<–Tclone tag 5 fid 176 newfid 169
ramfs:–>Rclone tag 5 fid 176
ramfs:<–Topen tag 5 fid 169 mode 0
ramfs:–>Ropen tag 5 fid 169 qid 0x80000000|0x0
ramfs:<–Tclone tag 5 fid 176 newfid 188
ramfs:–>Rclone tag 5 fid 176
ramfs:<–Topen tag 5 fid 188 mode 0
ramfs:–>Ropen tag 5 fid 188 qid 0x80000000|0x0
ramfs:<–Tread tag 5 fid 188 offset 0 count 5800
ramfs:–>Rread tag 5 fid 188 count 0 ''
ramfs:<–Tclunk tag 5 fid 188
ramfs:–>Rclunk tag 5 fid 188
ramfs:<–Tclunk tag 5 fid 169
ramfs:–>Rclunk tag 5 fid 169
```

Um den Verweis auf das Wurzelverzeichnis (*fid* 176) nicht zu verlieren, werden mit Hilfe neuer *fids* die anstehenden Operationen durchgeführt. *ls* untersucht, ob */tmp* ein Katalog ist, öffnet (zweimal!) und liest ihn. Allerdings ist das Verzeichnis noch leer.

Datei erzeugen

Erzeugen und Füllen einer Datei:

```
% echo –n Hello World > /tmp/hello
ramfs:<–Tclone tag 5 fid 176 newfid 173
ramfs:–>Rclone tag 5 fid 176
ramfs:<–Tclone tag 5 fid 173 newfid 188
ramfs:–>Rclone tag 5 fid 173
```

```
ramfs:<—Twalk tag 5 fid 188 name hello
ramfs:—>Rerror tag 5 error file does not exist
ramfs:<—Tclone tag 5 fid 176 newfid 169
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 169 name hello
ramfs:—>Rerror tag 5 error file does not exist
ramfs:<—Tclunk tag 5 fid 169
ramfs:—>Rclunk tag 5 fid 169
ramfs:<—Tclunk tag 5 fid 188
ramfs:—>Rclunk tag 5 fid 188
ramfs:<—Tcreate tag 5 fid 173 name hello perm 0x1b6 mode 1
ramfs:—>Rcreate tag 5 fid 173 qid 0x1|0x0
ramfs:<—Twrite tag 5 fid 173 offset 0 count 11 'Hello World'
ramfs:—>Rwrite tag 5 fid 173 count 11
ramfs:<—Tclunk tag 5 fid 173
ramfs:—>Rclunk tag 5 fid 173
```

Hier sieht man, wie zunächst eine Datei *hello* (zweimal!) gesucht wird und wie dann die Datei angelegt und gefüllt wird.

Datei ausgeben

Ausgabe der Datei:

```
% cat /tmp/hello
ramfs:<—Tclone tag 5 fid 176 newfid 169
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 169 name hello
ramfs:—>Rwalk tag 5 fid 169 qid 0x1|0x1
ramfs:<—Topen tag 5 fid 169 mode 0
ramfs:—>Ropen tag 5 fid 169 qid 0x1|0x1
ramfs:<—Tread tag 5 fid 169 offset 0 count 8192
ramfs:—>Rread tag 5 fid 169 count 11 'Hello World'
Hello Worldramfs:<—Tread tag 5 fid 169 offset 11 count 8192
ramfs:—>Rread tag 5 fid 169 count 0 ''
ramfs:<—Tclunk tag 5 fid 169
ramfs:—>Rclunk tag 5 fid 169
```

Hier ist zu sehen, wie die existente Datei *hello* gesucht, gefunden und gelesen wird.

Zugriffsschutz ändern

Folgende 9P-Nachrichten erzeugen ein *chmod*-Aufruf:

```
% chmod —w /tmp/hello
ramfs:<—Tclone tag 5 fid 176 newfid 163
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 163 name hello
ramfs:—>Rwalk tag 5 fid 163 qid 0x1|0x1
ramfs:<—Tstat tag 5 fid 163
ramfs:—>Rstat tag 5 fid 163 stat 'hello' 'dbkuehl' 'dbkuehl'
    q 0x1|0x1
    m 0664 at 1846833390 mt 1846833365 l 11 t 0 d 1
```

```
ramfs:<—Tclunk tag 5 fid 163
ramfs:—>Rclunk tag 5 fid 163
ramfs:<—Tclone tag 5 fid 176 newfid 169
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 169 name hello
ramfs:—>Rwalk tag 5 fid 169 qid 0x1|0x1
ramfs:<—Twstat tag 5 fid 169 stat 'hello' 'dbkuehl' 'dbkuehl'
    q 0x1|0x1
    m 0444 at 1846833390 mt 1846833365 l 11 t 77 d 20
ramfs:—>Rwstat tag 5 fid 169
ramfs:<—Tclunk tag 5 fid 169
ramfs:—>Rclunk tag 5 fid 169
```

Zunächst wird der aktuelle Status von *hello* abgeholt und anschließend neu geschrieben.

Datei entfernen

Ein *rm* auf eine nichtexistente Datei liefert:

```
% rm /tmp/file
ramfs:<—Tclone tag 5 fid 176 newfid 163
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 163 name file
ramfs:—>Rerror tag 5 error file does not exist
ramfs:<—Tclone tag 5 fid 176 newfid 173
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 173 name file
ramfs:—>Rerror tag 5 error file does not exist
ramfs:<—Tclunk tag 5 fid 173
ramfs:—>Rclunk tag 5 fid 173
ramfs:<—Tclunk tag 5 fid 163
ramfs:—>Rclunk tag 5 fid 163
rm: /tmp/file: file does not exist
```

Hier liefert *ramfs* eine *Rerror*-Nachricht anstelle von *Rwalk* als Antwort auf die *Twalk*-Anfrage. Der in *Rerror* enthaltene Fehlertext wird als Fehlermeldung ausgegeben.

Ein *rm* auf die existente Datei *hello* liefert:

```
% rm /tmp/hello
ramfs:<—Tclone tag 5 fid 176 newfid 169
ramfs:—>Rclone tag 5 fid 176
ramfs:<—Twalk tag 5 fid 169 name hello
ramfs:—>Rwalk tag 5 fid 169 qid 0x1|0x1
ramfs:<—Tremove tag 5 fid 169
ramfs:—>Rremove tag 5 fid 169
```

Das von *ramfs* realisierte Dateisystem bietet gegenüber einem »normalen« Dateisystem eine reduzierte Sicherheit. So findet zu Beginn keine Authentifizierung statt, und Dateioperationen werden teilweise nicht gegen die Zugriffsrechte bezüglich dieser Dateien überprüft. Daher führt im Beispiel *ramfs* das *rm*-Kommando durch, obwohl weiter oben mit *chmod* verboten wurde, die Datei zu schreiben. Jetzt existiert *hello* nicht mehr.

```
% unmount /tmp
ramfs:<—Tclunk tag 5 fid 176
ramfs:—>Rclunk tag 5 fid 176
ramfs: mount read: write to hungup stream
```

Und damit existiert dann auch der Server-Prozeß nicht mehr.

5.3 Kernel-Server

Ein großer Vorteil von Plan 9 ist, daß die kompletten Quellen vorhanden sind. Dieser Abschnitt erklärt zunächst den strukturellen Aufbau des Quellbaums. Anschließend werden die Kern-Quellen untersucht, um schließlich zu zeigen, wie ein eigener Kernel-Server programmiert und in einen Kern integriert wird.

Plan 9-Quellen

Da Kernel-Server fest im Betriebssystemkern eingebrannt sind, ist es bei der Entwicklung von Kernel-Servern unabdingbar zu wissen, wie der Kern für eine bestimmte Architektur gebaut wird.

Die kompletten Quellen zu Plan 9 sind unterhalb von */sys/src* zu finden. Im Unterkatalog *cmd* befinden sich die Quellen zu den Kommandos:

```
% lc /sys/src/cmd
2a          cp.c        ip          plot        sum.c
2c          cpp         join.c      pm          swap.c
2l          cpu.c       ka          postscript  syscall
6a          date.c      kc          ppp         tail.c
6c          db          ki          pr.c        tapefs
6l          dc.c        kl          primes.c    tar.c
8a          dd.c        kprof.c     prof.c      tbl
8c          deroff.c    ktrans      proof       tcs
8l          dict        lex         ps.c        tee.c
8½          diff        look.c      pwd.c       telco
...
%
```

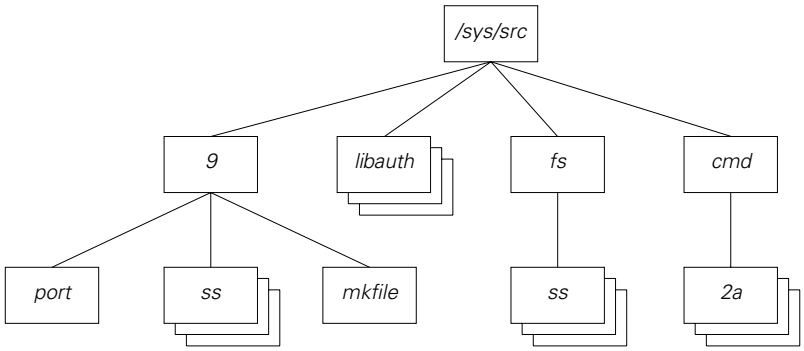
Pro Bibliothek gibt es ein Unterverzeichnis. Der Name des Verzeichnisses setzt sich aus dem Präfix *lib* und dem Namen der Bibliothek als Suffix zusammen:

```
% lc —d /sys/src/lib*
libauth     libg        liblex      libregexp
libbio      libgeometry liblock     libstdio
libc        libgnot     libmach     libtiff
libfb       libip       libndb
libframe    liblayer    libpanel
%
```

Die Quellen des File-Servers sind in *fs* und die zu einem Terminal oder CPU-Server sind unterhalb von *9* zu finden. Dort befinden sich im Katalog *port* alle Architekturunabhängigen Quellen des Kerns. Darüber hinaus gibt es pro unterstützte Architektur ein Verzeichnis, so z.B. *pc* für Intel-Prozessoren oder *ss* für Sparc Stations:

```
% lc /sys/src/fs
6280   cyc    magnum mkfile pc     port   power  ss
% lc /sys/src/9
boot     indigo3k mkfile   port     ss10
chm      indigo4k next     power
gnot     magnum   pc       ss
%
```

Es ergibt sich insgesamt folgender Aufbau:



Kern-Quellen

Zum lokalen Experimentieren kann mit

```
{ cd /sys/src; tar c 9 } | { cd /tmp; tar x }
```

eine Kopie der Quellen angelegt werden.

Im Verzeichnis *9* und in den verschiedenen Unterverzeichnissen (*ss*, *pc*, ...) existiert jeweils ein *mkfile*, mit dem ein Kern gebaut werden kann. Im Verzeichnis *ss* hat das *mkfile* folgenden Inhalt:

```
CONF=ss
CONFLIST=ss sscpu sscd

objtype=sparc
</$objtype/mkfile
CFLAGS=-w

DEV=`{rc ../port/mkdevlist $CONF}
STREAM=`{rc ../port/mkstreamlist $CONF}
MISC=`{rc ../port/mkmisclist $CONF}

OBJ=\
    l.$O\
    alarm.$O\
    chan.$O\
    ...
    $DEV\
    boot$CONF.root.$O\
    stream.$O\
    $MISC $STREAM\
```

```
    bbmalloc.$O\
    ...
    auth.$O

default:V: 9$CONF

9$CONF: $OBJ $CONF.c /sparc/lib/libgnot.a /sparc/lib/libg.a
    $CC $CFLAGS '-DKERNDATE='`{date -n} $CONF.c
    $LD -o 9$CONF -H0 -l -T0xE0004000 -R0x4 $OBJ \
                    $CONF.$O -lgnot -lg -lauth -lc
    size 9$CONF
    ...

<../port/portmkfile

    ...
```

Die Variable *CONFLIST* enthält eine Liste aller für eine Sparc möglichen Kerne. So ist der Kern *sscpu* der Kern eines CPU-Servers, *sscd* ein Terminal-Kern, der die CD als Wurzel verwenden kann und *ss* ein »normaler« Sparc-Kern. Durch *mk all* im Verzeichnis *9* werden für alle Architekturen alle Kerne (*CONFLIST*) gebaut. Ein *mk* in einem der Unterverzeichnisse (*ss*, *pc*, ...) übersetzt den in *CONF* spezifizierten Kern.

Pro zu übersetzenden Kern gibt es eine Konfigurationsdatei gleichen Namens, die die Konfiguration des Kerns bestimmt. Die Datei *ss* kann wie folgt aussehen:

```
dev
    root    /
    cons    c
    env     e
    pipe    |
    proc    p
    srv     s
    mnt     M
    dup     d
    lance   l
    scc     t
    rtc     r
    ip      I   tcpinput tcpoutput tcptimer tcpif stil
    arp     a
    bit     b
    iproute P
    scsi    S   scsi scsibuf
    wren    w
    kprof   T
    cdrom   R

...

port
    int cpuserver = 0;
    long cfslen = 0; ulong cfscode[1];
    long fslen = 0; ulong fscode[1];
    void consdebug(void) {}

boot terminal
    il
    tcp
```

Im Abschnitt *dev* werden die in dem Kern zu integrierenden Kernel-Server ausgewählt. So werden hier z.B. die Server *root* (»*#/*«), *cons* (»*#c*«), *env* (»*#e*«), ... spezifiziert. In einem anderen Abschnitt (*port*) wird die Variable *cpuserver* auf Null gesetzt, d.h., dieser Kern ist kein CPU-Server-Kern.

Analog dazu sieht die Konfigurationsdatei für einen CPU-Server (*sscpu*) fast gleich aus:

ss	sscpu
`dev`	
`cdrom R`	—
`port`	
`int cpuserver =0 ;`	`int cpuserver = 1;`
`boot terminal`	`boot cpu`

Wie zu sehen ist, werden bei dem CPU-Server-Kern im Standard keine CD-ROM-Geräte unterstützt. Außerdem wird *cpuserver* auf 1 gesetzt, und es wird ein anderer Parameter zum Aufruf von */boot* mit angegeben.

Pro in der Konfigurationsdatei angegebenen Kernel-Server gibt es nun eine Datei, die den Quell-Code des Servers beinhaltet. Für Architektur-unabhängige Server ist die Datei in */sys/src/9/port* und für Architektur-abhängige im aktuellen Verzeichnis zu finden. Der Name der Datei setzt sich aus dem Präfix *dev*, dem Namen des Servers und dem Suffix *.c* zusammen:

```
% pwd
/sys/src/9/ss
% lc dev*
devrtc.c
% lc ../port/dev*.c
dev.c         devcdrom.c    devfloppy.c   devlance.c    devroot.c
devXXX.c      devcons.c     devicmp.c     devmnt.c      devscc.c
devarp.c      devdk.c       devip.c       devpipe.c     devscsi.c
devbit.c      devdup.c      deviproute.c  devproc.c     devsrv.c
devboot.c     devenv.c      devkprof.c    devreboot.c   devwren.c
%
```

So sind für eine Sparc Station alle Kernel-Server bis auf den Server *rtc* im *port*-Verzeichnis zu finden. Der Server *rtc* ist als Schnittstelle zum NVRAM sicherlich Architektur-abhängig.

Entwicklung eines Kernel-Servers

Die Entwicklung eines Kernel-Servers ist ein nicht ganz einfaches und in aller Regel aufwendiges Unterfangen. In diesem Abschnitt wird gezeigt, wie man einen Kernel-Server entwickelt.

Die Entwicklung und das Testen eines Kerns unterscheidet sich grundsätzlich von der Entwicklung einer Applikation. Tritt im realisierten Kern ein Fehler auf, erhält man als Reaktion keine Reste, die man mit einem geeigneten *debugger* analysieren könnte, sondern der Rechner bleibt meistens ohne weitere Reaktion stehen. Ein Plan 9 Kern ist interrupt-getrieben. Daraus folgt, daß dieses Verhalten bei der Designphase berücksichtigt werden muß.

Am einfachsten geht die Entwicklung voran, wenn ein Rechner zur Entwicklung und einer zum Testen des neuen Kerns verwendet werden kann. Wir booten unsere Plan 9-Terminals über das Netz. Soll ein neuer Kern installiert werden, wird dieser unter Plan 9 übersetzt und mittels *ftpfs* und *cp* in den boot-Bereich kopiert. Der daraufhin folgende boot-Vorgang wird dann mit dem neuen Kern unternommen.

5.4 Ein einfacher eigener Kernel-Server

Als Beispiel entwickeln wir einen Kernel-Server *example,* #E, der lediglich zwei Dateien *in* und *out* verwaltet. Daten, die in die Datei *in* geschrieben werden, können aus *out* wieder herausgelesen werden. Initial enthält *out* einen fixen Text.

```
% ls –l '#E'
––w––w––w– E 0 dbkuehl dbkuehl  0 Jun 30 17:00 #E/in
––r––r––r–– E 0 dbkuehl dbkuehl 38 Jun 30 17:00 #E/out
% mkdir test
% bind '#E' test
% cd test
% cat out
Hello!
I'm the Kernel–Server example!
% echo Hello World ! > in
% ll
––w––w––w– E 0 dbkuehl dbkuehl  0 Jun 30 17:00 in
––r––r––r–– E 0 dbkuehl dbkuehl 14 Jun 30 17:00 out
% cat out
Hello World !
%
```

Da die Funktionalität von dem Server *example* von der momentanen Architektur unabhängig ist, ist der richtige Platz für die Quelldatei *devexample.c* der *port*-Katalog. Die Datei *devXXX.c* dient als Blaupause bei der Entwicklung von neuen Kernel-Servern. Darüber hinaus lohnt sich ein Blick in die Implementation der anderen Kernel-Server. Wir kopieren als erstes die Datei *devXXX.c* nach *devexample.c*.

Die erste zu lösende Aufgabe besteht darin, den neuen Kernel-Server so in den Kern einzubauen, daß dieser einen Server unter dem Bezeichner »#E« anbieten kann. Die Kern-Konfiguration hängt ausschließlich von einer Konfigurationsdatei, z.B. *ss*, ab. Beim Generieren eines Kerns wird ein zweidimensionaler Vektor gefüllt, in dem alle Funktionen aller Kernel-Server aufgelistet sind, die aufgrund der verwendeten Konfiguration benutzt werden.

```
% mk
...
rc ../port/mkdevc ss > ss.c
...
```

Die entstehenden Tabellen sehen wie folgt aus:

```
% cat ss.c
...
Dev                         devtab[]={
    { rootreset, rootinit, rootattach, rootclone,
      rootwalk, rootstat, rootopen, rootcreate,
      rootclose, rootread, rootwrite, rootremove,
      rootwstat, },
      ...
    { examplereset, exampleinit, exampleattach, exampleclone,
      examplewalk, examplestat, exampleopen, examplecreate,
      exampleclose, exampleread, examplewrite, exampleremove,
      examplewstat, },
      ...
};
Rune *devchar=L"/ce|psMdltrIabPSwTER";

...
```

Mnt ruft abhängig von der 9P-Nachricht die Funktion des verantwortlichen Servers auf. Das Auswählen einer Funktion durch *mnt* erfolgt immer ausschließlich über Indizes und den Komponentennamen. Der Index ist die Position des entsprechenden Zeichens des Kernel-Servers in *devchar[]. Dev* ist in *port/portdat.h* definiert.

Damit die Tabellen automatisch initialisiert werden können, müssen bestimmte Konventionen in der Namensgebung der Funktionen und Quellen eingehalten werden. Jeder Funktionsname eines Kernel-Servers muß dem *dev* nachfolgenden Text vorangestellt sein. In unserem Falle kann dies durch

```
% cat devXXX.c | sed 's/XXX/example/g' > devexample.c
```

erreicht werden. Falls in der Konfigurationsdatei der Server wie folgt eingetragen ist

```
example      E
```

müssen die Quellen in der Datei *devexample.c* zu finden sein, damit die Übersetzung erfolgreich verlaufen kann.

exampleattach() muß wie folgt modifiziert werden:

```
Chan *
/* XXXattach(char *spec)            was     */
   devexampleattach(char *spec)
{
/*  return devattach('X', spec);    was     */
    return devattach('E', spec);
}
```

Dann kann *example* nach Übersetzung und erfolgreichem boot-Vorgang mit

```
%term bind '#E' /tmp
```

montiert werden. Damit die Dateien *in* und *out* angelegt werden, muß *exampletab* wie folgt modifiziert werden:

```
enum{
    examplein,
    exampleout,
};

enum{
    exampleinqid=1,
    exampleoutqid
};

Dirtab exampletab[]={
    "in",       {exampleinqid}, 0,  0222,
    "out",      {exampleoutqid},    0,  0444,
};
#define Nexampletab (sizeof(exampletab)/sizeof(Dirtab))
```

Das Array *exampletab* enthält eine Beschreibung aller vom Kernel-Server angebotenen Dateien. Die Struktur *Dirtab* stammt aus *port/portdat.h* und ist dort folgendermaßen definiert:

```
struct Dirtab
{
    char    name[NAMELEN];
    Qid qid;
    long    length;
    long    perm;
};
```

Die Struktur *Qid* ist in *port/lib.h* definiert:

```
struct Qid
{
    ulong   path;
    ulong   vers;
};
```

Damit ist der neue Kernel-Server insoweit komplett, als er nach dem Montieren die Dateien *in* und *out* anbietet. Es stellt sich die Frage nach der Programmierung der einzelnen 9P-Funktionen, sprich, wie hat z.B. die Twrite auszusehen, damit ein Programm via

```
cat program > in
```

Daten in die Datei *in* schreiben kann? Der Server *example* speichert Daten, die über *in* geschrieben und über *out* gelesen werden, in einer globalen Variablen *data*. Diese wird initial mit einem festen Text in der Funktion *exampleinit* gefüllt. Jeder Kernel-Server enthält diese Initialisierungs-Funktion, die einmal zu Beginn aufgerufen wird.

```
#define BSIZE   1024
static char data[BSIZE];

void
exampleinit(void)
{
    strcpy(data,"Hello!\nI'm the Kernel—Server example!\n");
    exampletab[exampleout].length=38;
}
```

In der Funktion *examplewrite* werden Daten, die in *in* geschrieben werden, in die globale Variable *data* kopiert.

```
long
examplewrite(Chan *c, char *a, long n, ulong offset)
{
    switch(c->qid.path & ~CHDIR){
    case exampleoutqid:
        error(Eperm);
        break;
    case exampleinqid:
        if(offset+n>BSIZE)
            error(Etoobig);
        memcpy(data+offset,a,n);
        exampletab[exampleout].length=offset+n;
        return n;
        break;
    default:
        error(Enonexist);
    }
    return 0;
}
```

In der Funktion *exampleread* werden Daten, die aus *out* gelesen werden, der globalen Variablen *data* entnommen.

```
long
exampleread(Chan *c, void *a, long n, ulong offset)
{
    int dataLength=exampletab[exampleout].length;

    if(c->qid.path&CHDIR)
        return devdirread(c, a, n, exampletab, Nexampletab,
                          devgen);

    switch(c->qid.path & ~CHDIR){
    case exampleoutqid:
        if(offset>=dataLength)
            return 0;
        if(dataLength<=n+offset)
            n=dataLength-offset;
        memcpy(a,data+offset,n);
        return n;
    case exampleinqid:
        error(Eperm);
        break;
    default:
        error(Enonexist);
    }
    return 0;
}
```

Damit ist der einfache Kernel-Server *example* auch schon fertig. Aller anderen 9P-Funktionen brauchen nicht abgeändert werden.

Ein Debuggen innerhalb des Kerns erfolgt i.a.R. ausschließlich über *print*-Anweisungen. Bei uns hat sich folgende Technik bewährt:

```
#ifdef DEBUG
void P(char * x) {  if ( doDebug ) putstrn(x, strlen(x) );  }
#else
#   define P(x)      /*  */
#endif

...
    P("hello");      /* Nutzung  */

...
```

Je nach Übersetzung ist es möglich, debug-Information ausgeben zu lassen, falls der Wert von *doDebug* auf »!= 0« gesetzt werden kann. Die Idee dabei ist, daß man über den Kernel-Server *example* Texte wie

```
debug on
```

oder

```
debug off
```

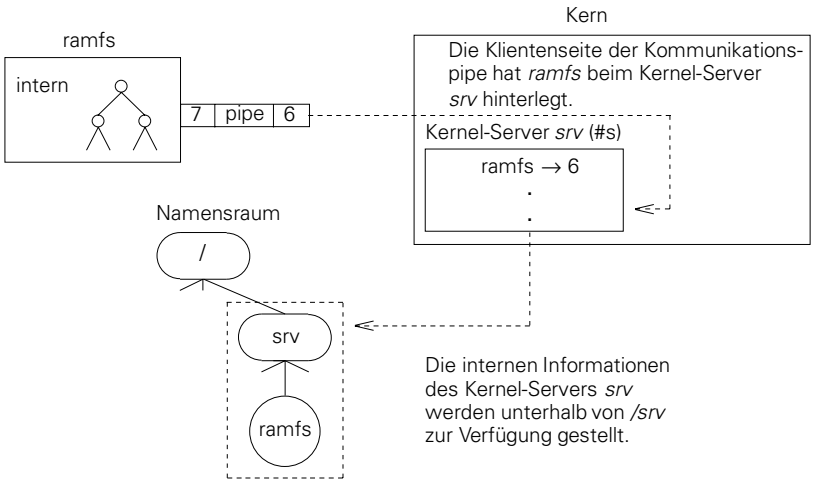
in die Datei *in* schreibt und in der Funktion *examplewrite* je nach Inhalt des Textes die Variable *doDebug* setzt oder nicht. Über die Datei *out* kann der Server außerdem beliebige Informationen über den Kern nach außen hin zur Verfügung stellen.

5.5 User-Server

Analog zu den fest im Kern eingebauten Servern erlaubt Plan 9 das Hinzufügen von eigenen Servern, den sogenannten *User-Servern*. Sie sind im Gegensatz zu Kernel-Servern eigenständige Prozesse. In Kapitel 4 wird erklärt, daß die Kommunikation der User-Server mit der Außenwelt über eine bidirektionale Kommunikationsleitung (z.B. eine Pipe oder eine Netzverbindung) geschieht. Mittels *mount*(2) erweitert ein Prozeß den Namensraum um ein Dateisystem des User-Servers, der über die Kommunikationsleitung durch 9P-Nachrichten angesprochen wird. Die Umformulierung von Server-Aufrufen in Nachrichten, also die Abwicklung der RPC, übernimmt der Kernel-Server *mnt*, #M. Das Kommando *mount*(1) dient zur Benutzung des Systemaufrufs *mount*(2). Das Kommando erwartet als Argument eine offene, bidirektionale Dateiverbindung, die jedoch als Datei angegeben werden muß. Damit User-Server sich für das *mount*-Kommando anbieten können, gibt es den Kernel-Server *srv*, #s, bei dem eine offene Verbindung abgelagert und als Datei angeboten werden kann.

ramfs

Als Beispiel folgen die einzelnen Schritte beim Starten des User-Servers *ramfs* und beim Montieren des Dateisystems von *ramfs*:

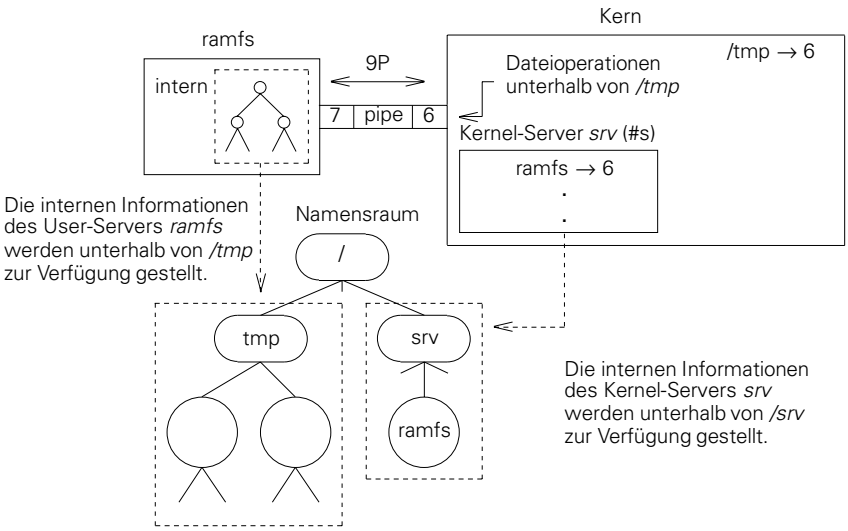


Durch das Kommando *ramfs -s* in der Shell wird der Server *ramfs* gestartet. Dieser erzeugt mit *pipe*(2) die Kommunikationspipe (z.B. p[0]=6 und p[1]=7) und hinterlegt in der Datei */srv/ramfs* die Klientenseite der Pipe (6). Da das Verzeichnis */srv* nach Konvention vom Kernel-Server *srv* verwaltet wird, ist dieser für die Datei */srv/ramfs* zuständig. Der Kernel-Server *srv* merkt sich die Beziehung zwischen der Datei *ramfs* und dem Filedeskriptor (6) und stellt seine Informationen im Verzeichnis */srv* zur Verfügung.

Das Kommando

```
mount /srv/ramfs /tmp
```

öffnet die Datei */srv/ramfs*, erhält deshalb vom Kernel-Server *srv*, #s, Zugriff auf die zugrundeliegende Dateiverbindung, also die Pipe zum Prozeß *ramfs*, und fügt diese Dateiverbindung mit dem Systemaufruf *mount*(2) an der Stelle */tmp* im Namensraum ein. Danach ist der alte Inhalt von */tmp* verdeckt und nicht mehr sichtbar. Der Kern merkt sich im Laufe von *mount*(2), daß *ramfs* für */tmp* verantwortlich ist, und speichert den zugehörenden Filedeskriptor. Alle Dateioperationen bezüglich Dateien unterhalb von */tmp* werden vom Kernel-Server *mnt*, #M, als Platzhalter in entsprechende 9P-Nachrichten über die Pipe an den User-Server *ramfs* umgewandelt. Es ergibt sich folgende Abbildung:



Der User-Server braucht die Klienten-Seite der Pipe nicht in */srv* publik zu machen, sondern kann durch einen Aufruf *mount*(2) selbst sein Dateisystem dem aktuellen Namensraum zufügen. Der Server benutzt dabei normalerweise ein Standardverzeichnis als Montagepunkt. Der Anwender kann ein Verzeichnis aber auch durch ein Kommandozeilenargument beim Starten des User-Servers explizit angeben.

Programmierung von User-Servern

In diesem Kapitel sollen in kurzen Schritten die Programmierung eines existenten User-Servers (*ramfs*) und darauf aufbauend die Entwicklung eines eigenen, neuen User-Servers besprochen werden.

Die Programmierung von *ramfs*

Eine Einführung in die Programmierung von User-Servern findet hier exemplarisch anhand der Quelle von *ramfs* (*/sys/src/cmd/ramfs.c*) statt. Diese unterliegt dem Copyright von AT&T. An einigen Stellen ist der Quelltext modifiziert worden, um ihn verständlicher und übersichtlicher gestalten zu können.

Ramfs verwendet zwei Strukturen: *Ram* repräsentiert eine Datei und *Fid* einen *fid*-Wert, den ein Klient eingeführt hat:

```
typedef struct Fid Fid;
typedef struct Ram Ram;

struct Fid
{
    short   busy;           /* aktiv ?      */
    short   open;           /* geoeffnet ?  */
    short   rclose;     /* remove after close   */
    int fid;        /* die fid  */
    Fid *next;      /* naechste fid in Fid-Liste    */
    Ram *ram;       /* zugehoeriges Objekt  */
};
```

```
struct Ram          /* pro internes Objekt eine Ram–Struktur */
{
    short   busy;       /* aktiv ?  */
    short   open;       /* geoeffnet ?  */
    long    parent;     /* index in ram–Vektor  */
    Qid qid;        /* die qid des Objektes */
    char    name[NAMELEN];  /* Name des Objektes    */
    char    *data;      /* der Inhalt   */
    long    ndata;      /* Anzahl Bytes in data */
};
```

Im Hauptprogramm erzeugt *ramfs* eine bidirektionale Pipe als Kommunikationsmittel mit den Klienten. Die Wurzel des internen Dateisystems wird erzeugt. Dann legt man den eigentlichen Server (Sohnprozeß) als verwaisten Prozeß an, und über *mount*(2) richtet man die Pipe-Verbindung zwischen dem Kern (*#M*) und dem Server ein:

```
enum { Nram = 512 };

Ram ram[Nram];      /* Objekt–Vektor    */
int nram;       /* Anzahl Objekte in ram    */
int mfd[2];     /* Pipe–Filedeskriptoren    */

void main(int argc, char *argv[])
{   Ram *r;
    char *defmnt;
    int p[2];

    defmnt = "/tmp";    /* Standard–Montageverzeichnis  */
    ARGBEGIN{ . . . }ARGEND /* Kommandozeilenabarbeitung */

    if(pipe(p) < 0)     /* Kommunikationspipe erzeugen  */
        error("pipe failed");

    mfd[0] = p[0];
    mfd[1] = p[0];

    nram = 1;       /* Wurzelverzeichnis erzeugen   */
    r = &ram[0];
    r–>busy = 1;
    r–>data = 0;
    r–>ndata = 0;
    r–>qid.path = CHDIR;
    r–>qid.vers = 0;
    r–>parent = 0;
    strcpy(r–>name, ".");

    switch(rfork(RFFDG|RFPROC|RFNAMEG|RFNOTEG)){
    case –1:
        error("fork");
    case 0:         /* Der eigentliche Server   */
        close(p[1]);
        io();
        break;
```

```
        default:
            close(p[0]);    /* Service montieren    */
            if( defmnt && mount(p[1], defmnt, MREPL|MCREATE, "")
                < 0)
                error("mount failed");
        }
        exits(0);
    }
```

9P-T-Nachrichten, die vom Kern in die Pipe geschrieben werden, empfängt der Server-Prozeß innerhalb von *io()* und schreibt die zugehörigen R-Nachrichten zurück in die Pipe. Die Funktion *io()* ist von der Funktionalität her ein RPC-Verteiler. Sie empfängt die 9P-Nachrichten und ruft je nach Typ der Nachricht eine Funktion in dem User-Server auf.

```
#include <fcall.h>

char    mdata[MAXMSG+MAXFDATA];
Fcall   rhdr;   /* eintreffende 9P—Nachricht    */
Fcall   thdr;   /* zu verschickene 9P—Antwort   */

void io(void)
{   char *err;  /* Fehlerstring */
    int n;

    for(;;){    /* auf eintreffende 9P—Nachricht warten */
        n = read(mfd[0], mdata, sizeof mdata);
        if(n == 0)
            continue;
        if(n < 0)
            error("mount read");
        /* Fcall—Struktur rhdr mit der 9P—Nachricht fuellen */
        if(convM2S(mdata, &rhdr, n) == 0)
            continue;

        thdr.data = mdata + MAXMSG;
        if(!fcalls[rhdr.type])
            err = "bad fcall type";
        else    /* Die zum Typ gehoerige Funktion aufrufen  */
            err = (*fcalls[rhdr.type])(newfid(rhdr.fid));
        if(err){    /* Fehler ? */
            thdr.type = Rerror;
            strncpy(thdr.ename, err, ERRLEN);
        }else{
            thdr.type = rhdr.type + 1;
            thdr.fid = rhdr.fid;
        }
        thdr.tag = rhdr.tag;

        n = convS2M(&thdr, mdata);  /* Antwort verschicken  */
        if(write(mfd[1], mdata, n) != n)
            error("mount write");
    }
}
```

Der Datentyp *Fcall* ist eine Struktur aus *fcall.h*, die 9P-Nachrichten repräsentiert:

```
#define MAXFDATA8192
#define MAXMSG160     /* max header sans data */

typedef struct Fcall
{  char type;
   short    fid;
    ushort  tag;
    union { struct {ushort  oldtag;     /* Tflush          */
        Qid qid;                        /* Rattach, Rwalk  */
                                        /*  Ropen, Rcreate */
        char    rauth[AUTHENTLEN];      /* Rattach         */
    };
    struct {    char    uname[NAMELEN]; /* Tattach         */
        char    aname[NAMELEN];         /* Tattach         */
        char    ticket[TICKETLEN];      /* Tattach         */
        char    auth[AUTHENTLEN];       /* Tattach         */
    };
    struct {    char    ename[ERRLEN];  /* Rerror          */
        char    authid[NAMELEN];        /* Rsession        */
        char    authdom[DOMLEN];        /* Rsession        */
        char    chal[CHALLEN];          /* Tsession/Rsessi.*/
    };
    struct {    long    perm;           /* Tcreate         */
        short   newfid;                 /* Tclone, Tclwalk */
        char    name[NAMELEN];          /* Twalk, Tclwalk, */
                                        /* Tcreate         */
        char    mode;                   /* Tcreate, Topen  */
    };
    struct {    long    offset;         /* Tread, Twrite   */
        long    count;                  /* Tread, Twrite,  */
                                        /* Rread           */
        char    *data;                  /* Twrite, Rread   */
    };
    struct {    char    stat[DIRLEN];   /* Twstat, Rstat   */
    };
};
} Fcall;
```

Die Funktion *io* verwendet folgenden Vektor, um die empfangenen Nachrichten auf einzelne Funktionen zu verteilen. Dabei gibt es pro 9P-T-Nachrichtentyp eine Funktion:

```
char    *(*fcalls[])(Fid*) = {
    [Tflush]    rflush,
    [Tsession]  rsession,
    [Tnop]      rnop,
    [Tattach]   rattach,
    [Tclone]    rclone,
    [Twalk]     rwalk,
    [Tclwalk]   rclwalk,
    [Topen]     ropen,
```

```
    [Tcreate]   rcreate,
    [Tread]     rread,
    [Twrite]    rwrite,
    [Tclunk]    rclunk,
    [Tremove]   rremove,
    [Tstat]     rstat,
    [Twstat]    rwstat,
};
```

Tflush, *Tsession*, ... sind dabei *enum*-Konstanten aus *fcall.h*. Diese Syntax zur Initialiserung eines Vektors ist eine Besonderheit von Plan 9-C.

Die Funktion *newfid* muß entweder eine neue *fid* anlegen oder eine existente finden:

```
Fid *fids;

Fid * newfid(int fid)
{   Fid *f, *ff;

    ff = 0;
    for(f = fids; f; f = f->next)
        if(f->fid == fid)
            return f;
        else if(!ff && !f->busy)
            ff = f;
    if(ff){
        ff->fid = fid;
        return ff;
    }
    f = emalloc(sizeof *f);
    f->ram = 0;
    f->fid = fid;
    f->next = fids;
    fids = f;
    return f;
}
```

Pro 9P-T-Nachrichtentyp gibt es nun eine Funktion, die die empfangene Nachricht verarbeitet. Alle diese Funktionen aufzuführen und zu erklären, würde den Rahmen dieses Kapitels sprengen. Daher werden im weiteren die Funktionen zu *attach* und *clone* explarisch erläutert.

Für *attach* wird ein Zugriff auf die Wurzel eingerichtet. Dies wurde in *main()* angelegt:

```
char* rattach(Fid *f)
{
    f->busy = 1;
    f->rclose = 0;
    f->ram = &ram[0];
    thdr.qid = f->ram->qid;
    return 0;
}
```

Für *clone* muß eine zusätzliche *fid* angelegt werden. Das ist nicht legal, wenn die ursprüngliche *fid* nicht existiert oder wenn auf sie ein *Topen*-Zugriff besteht:

```
char    Enotexist[] =   "file does not exist";
char    Eisopen[] =     "file already open for I/O";

char* rclone(Fid *f)
{   Fid *nf;

    if(f->open)
        return Eisopen;
    if(f->ram->busy == 0)
        return Enotexist;
    nf = newfid(rhdr.newfid);
    nf->busy = 1;
    nf->open = 0;
    nf->rclose = 0;
    nf->ram = f->ram;
    return 0;
}
```

Der vollständige Quelltext der Funktionen *rattach* und *rclone* und die weiteren 9P-Verarbeitungsfunktionen können im Quelltext (*/sys/src/cmd/ramfs.c*) studiert werden.

Alle anderen User-Server sind nach dem gleichen Prinzip aufgebaut. Über eine bidirektionale Pipe empfängt ein Server 9P-T-Nachrichten, analysiert den Typ und ruft pro Typ eine Funktion auf. In der Funktion werden die empfangenen Daten verarbeitet. Als Antwort wird bei einem Fehler eine *Rerror*-Nachricht oder eine R-Nachricht des gleichen Typs geschickt. Dieser ganze Ablauf hat starke Ähnlichkeit mit RPCs.

5.6 Ein eigener User-Server: *mailsrv*

An dieser Stelle soll die Entwicklung eines eigenen User-Servers beschrieben werden. Der neue Server namens *mailsrv* bietet dem Anwender den Inhalt seiner Mailbox als Dateisystem an.

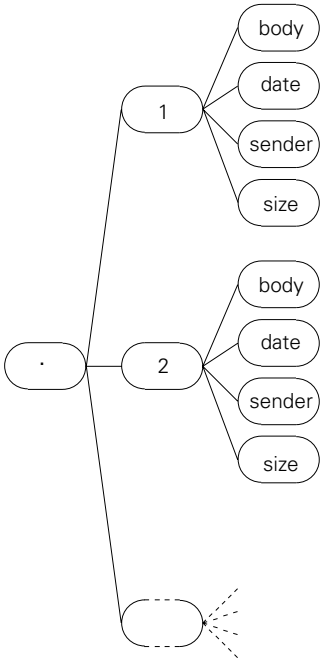
mail unter Plan 9

Unter Plan 9 kann sich jeder Benutzer durch Aufruf von *mail -c* eine Mailbox einrichten und mit *mail Empfänger* Mails an den spezifizierten Anwender verschicken. Dieser kann sich dann seine eingetroffenen Mails ansehen:

```
% mail
2 messages
?b
  1     58 mak Mon Jun 30 14:26
  2     44 bischof Mon Jun 30 14:16
?2
From bischof Mon Jun 30 14:16:01 MET 1997
Hallo Bernd!

Denkst Du an heute Abend?

hp
?x
%
```

Ein Mail-Server

Die Idee eines User-Servers *mailsrv* ist, dem Anwender den Inhalt seiner Mailbox als Dateisystem anzubieten. Dabei hat das von *mailsrv* angebotene *mail*-Dateisystem folgenden Aufbau:



Pro Mail gibt es ein Verzeichnis, welches wiederum 4 Dateien beinhaltet. Die Datei *sender* enthält den Absender der Mail, *date* das Empfangsdatum, *body* den Text und *size* die Größe der Mail.

Die Nutzung könnte wie folgt aussehen:

```
% mkdir test
% mailsrv –m test
% cd test
% ll
d–r–xr–xr–x M 42 dbkuehl dbkuehl 0 Jun 30 14:29 1
d–r–xr–xr–x M 42 dbkuehl dbkuehl 0 Jun 30 14:29 2
% ll 1
––r––r––r–– M 42 dbkuehl dbkuehl 58 Jun 30 14:29 1/body
––r––r––r–– M 42 dbkuehl dbkuehl 28 Jun 30 14:29 1/date
––r––r––r–– M 42 dbkuehl dbkuehl  3 Jun 30 14:29 1/sender
––r––r––r–– M 42 dbkuehl dbkuehl  2 Jun 30 14:29 1/size
% cat */sender
makbischof% echo `{cat 1/sender} `{cat 2/sender}
mak bischof
```

```
% cat 2/body
Hallo Bernd!

Denkst Du an heute abend?

hp
% cat 2/date
Mon Jun 30 14:16:01 MET 1997%
```

Der Aufruf von *mailsrv* ist analog zu *ramfs*:

```
mailsrv [-s] [-d] [-m mountpoint]
```

Dabei haben auch die Optionen die gleiche Bedeutung:

- `-d` Ausgabe der 9P-Nachrichten und der eingelesenen Mailbox
- `-s` nicht implizit montieren, sondern */srv/mailsrv* anlegen
- `-m` implizit auf *mountpoint* montieren (statt auf */tmp*)

In dem obigen Beispiel wurde durch die Flagge *-m* der Service von *mailsrv* unterhalb des Verzeichnisses *test* montiert. Wandert man nun durch das Dateisystem, so sieht man das bereits beschriebene Mail-Dateisystem.

Aufbauend auf *mailsrv* sind verschiedene Mailtools sehr leicht als Shell-Skripte realisierbar.

Implementierung von *mailsrv*

Der Server *mailsrv* arbeitet intern mit zwei Strukturen. Die Struktur *mbox* dient zur Verwaltung einer Mailbox, d.h. zur Verwaltung der verschiedenen Mails eines Benutzers. Die einzelnen Mails werden durch die Struktur *message* repräsentiert:

```
typedef struct message message;
struct message {        /* eine mail:           */
    char    *sender;    /* Absender             */
    char    *date;      /* Empfangsdatum        */
    char    *body;      /* Mail—Text            */
    int size;       /* Laenge des Textes        */
    int max;        /* aktuelle Laenge von body */
};

typedef struct mbox mbox;
struct mbox {           /* eine mailbox:        */
    message * mess;     /* die verschiedenen mails  */
    int max;        /* Anzahl moeglicher mails  */
    int in;         /* Anzahl mails in mailbox  */
    long time;      /* zum Test, ob neue mails  */
    char mbox_file[256];    /* mailbox—Datei        */
};
```

Die eingegangenen Briefe eines Anwenders werden vom System in der Datei */mail/box/$user/mbox* gespeichert. Ein Brief beginnt mit der Zeile, die am Anfang den Text »From « enthält. Die Datei hat für das obige Beispiel folgenden Aufbau:

```
From dbkuehl Tue Jul 18 15:04:07 MET 1996

Marions Geburtstag nicht vergessen!


From bischof Tue Jul 18 15:07:50 MET 1996
Hallo Bernd!

Denkst Du an heute abend?

hp

%
```

Diese Datei wird von *mailsrv* beim Start gelesen. Dabei wird eine Variable vom Typ *mbox* mit den Informationen aus der Datei gefüllt:

```
mbox m;

void main(int argc, char *argv[])
{
    ...
    switch(rfork(RFFDG|RFPROC|RFNAMEG|RFNOTEG)){
    case -1:
        error("fork");
    case 0:
        if(init_mbox(&m,Nmess)) /* Mailbox initialisieren */
            exits("no mailbox");
        if(read_mbox(&m))       /* Mailbox einlesen */
            exits("error mailbox read");
        makeFileSystem(&m);     /* Dateisystem erzeugen */

        close(p[1]);
        io();
        break;
    default:
        ...
    }
    exits(0);
}
```

Den Code der Funktionen *init_mbox* und *read_mbox* an dieser Stelle darzustellen, würde zu weit führen. Der komplette Quelltext von *mailsrv* ist an ??? (noch zu klären: ftp, beiliegende Diskette) zu finden.

Die Programmierung von *mailsrv* setzt auf den bereits bestehenden User-Server *ramfs* auf. Nachdem die verschiedenen Mails des Anwenders eingelesen worden sind, wird nun das Mail-Dateisystem durch das Eintragen von 5 Elementen pro Mail in den *ram*-Vektor von *ramfs* kreiert, einer für das Mail-Verzeichnis (1, 2, ...) und vier für die Dateien *body*, *date*, *sender* und *size.* Der folgende Auszug aus dem Quelltext zeigt exemplarisch das Erzeugen der Einträge für das Verzeichnis und für die Datei *body*:

```
void makeFileSystem(mbox * m)
{   Ram *r;
    int i;

    . . .
                              /* pro mail ein Verzeichnis:   */
    for(i=0;i<=(m->in);i++) {
        r=&ram[i*5+1];
        r->busy = 1;
        r->data = 0;
        r->ndata = 0;
        r->perm = CHDIR | 0555;
        r->qid.path = CHDIR | (++path);
        r->qid.vers = 0;
        r->parent = 0;      /* the root */
        r->user = user;
        r->group = user;
        r->atime = m->time;
        r->mtime = r->atime;
        sprint(r->name,"%d",(m->in-i)+1);   /* neue Mail   */
                                    /*zuerst, mit 1 anfangen*/
        makeMail(m,i*5+1);
        nram+=5;
    }
}
void makeMail(mbox *m,int parent)
{   Ram *r;
    int i=parent+1;
    message * mess;

    mess=&(m->mess[(parent-1)/5]);      /* aktuelle mail   */
                                        /* bestimmen       */
    ...
    /* make body */
    r=&ram[i++];
    r->busy = 1;
    r->ndata = mess->size-1;            /* letzte Zeile    */
                                        /* (^$) loeschen   */
    r->data=mess->body;
    r->perm = 0444;
    r->qid.path =++path;
    r->qid.vers = 0;
    r->parent = parent;
    . . .
    sprint(r->name,"body");
    ...
}
```

Jetzt ist der Server bereits fertig, da die eigentliche Programmierarbeit, das Schreiben der 9P-Bearbeitungsfunktionen, bereits in *ramfs* erledigt ist. Es ist jedoch sinnvoll, das Schreiben von Dateien und das Erzeugen von neuen Dateien oder Verzeich-

nissen innerhalb des Dateisystems von *mailsrv* zu verbieten. Gleiches gilt für das Löschen von Dateien und Verzeichnissen:

```
char    Eperm[] =   "permission denied";

char * rcreate(Fid *f)
{
    return Eperm;
}

char* rwrite(Fid *f)
{
    return Eperm;
}

char * rremove(Fid *f)
{
    return Eperm;
}
```

5.7 Zusammenfassung

Ein User-Server unter Plan 9 muß seinem Klienten, normalerweise dem Kernel-Server *mnt*, eine Seite einer bidirektionalen Pipe zur Kommunikation zur Verfügung stellen. Er liest auf seiner Seite der Pipe eintreffende 9P-Nachrichten, verarbeitet sie und beantwortet sie über die Pipe. Eine erste und einfache Anleitung zur Programmierung von User-Servern bietet der existente Server *ramfs*. Auf ihm aufbauend, ist die Entwicklung von neuen eigenen User-Servern, wie anhand von *mailsrv* gezeigt worden ist, relativ einfach.

Einen eigenen Kernel-Server zu implementieren, ist dagegen sicherlich schwieriger. Eine gute Anleitung sind die Quellen der bestehenden Kernel-Server, und *devXXX.c* kann als Vorlage benutzt werden.

6
rc
Ein Kommandoprozessor für Plan 9

rc (*run command*) ist der Kommando-Interpreter für Plan 9. *rc* ist für den Benutzer auf den ersten Blick eine ähnliche Schnittstelle zu Plan 9 wie Bournes */bin/sh* zu UNIX, ist aber vor allem im Makroprozessorbereich wesentlich einfacher konzipiert. Frei nach Tom Duff bietet sie in vielen Bereichen eine weniger verklausulierte Syntax. Es gibt *rc* auch für UNIX, sowohl von Tom Duff als auch in einer Public Domain Version von Byron Rakitzis (byron@archone.tamu.edu).

Tom Duff's »Rc — A Shell for Plan 9 and UNIX Systems« (in Plan 9 — The Early Papers) ist eine sehr gute Einführung in *rc* und motiviert seine Entwurfsentscheidungen.

Im Gegensatz zu */bin/sh* liest *rc* die Eingabe genau einmal. Daraus resultieren merkliche syntaktische und semantische Unterschiede zu */bin/sh*. Leider arbeitet der Scanner nicht unbedingt systematisch im Hinblick auf Zwischenraum.

6.1 Einfache Kommandos

Beim einfachsten Gebrauch, sprich Absetzen eines Kommandos auf einer Zeile, erwartet den Nutzer nichts Neues. Ein Wort ist ein Kommando:

```
% date
Sat Jul 18 09:46:39 GPT 1994
```

Weitere Wörter sind Argumente für das Kommando:

```
% echo hello world
hello world
```

Die Standardausgabe eines Kommandos kann mit '>' in eine Datei gelenkt und mit '>>' an eine Datei angehängt werden. Die Standardeingabe eines Kommandos kann mit '<' aus einer Datei geholt werden:

```
% who > who.out
% cat < who.out
bischof
bootes
none
```

Dateinamen — ein Dateiname darf in diesem Fall nicht mit einem Großbuchstaben bzw. einem Kleinbuchstaben aus der Menge { a - v } beginnen — können unter Verwendung von Metazeichen ausgewählt werden.

```
% ls [~A-Za-v]?*
who.out
```

Die Standardeingabe und Standardausgabe zweier Kommandos können mittels einer Pipe verbunden werden:

```
% who | wc
      3       3      20
```

Kommandos kann man im Hintergrund ablaufen lassen:

```
% date &
% Sat Jul 18 09:53:34 GPT 1994
```

Kommandos, die auf einer Zeile stehen, werden durch Semikolon getrennt:

```
% who > who.out; echo "who is done"
"who is done"
```

Die Ausführung eines Kommandos kann vom erfolgreichen Ablauf des vorangegangenen Kommandos abhängig gemacht werden:

```
% wc who.out && echo 'wc did ok'
      3       3      20 who.out
wc did ok
% rm not_exist || echo rm failed
rm: not_exist: file does not exist
rm failed
```

6.2 Zitieren

Enthält ein Argument Leerzeichen oder ein syntaktisch signifikantes Zeichen, so muß mit einfachen Anführungszeichen zitiert werden. Soll ein Apostroph in einem Argument verwendet werden, so gilt die übliche Spielregel: zwei stehen für eins.

```
% echo 'It''s nice to meet you.'
It's nice to meet you.
% echo 'wh*' wh*
wh* who.out
```

Schlüsselworte müssen zitiert werden, wenn sie nicht als solche erkannt werden sollen:

```
fn for if in not switch while.
```

Folgende Sonderzeichen sind für *rc* signifikant:

```
# ; & | ^ $ = ` ' { } ( ) < > ~ ! @
```

6.3 Variablen

rc stellt Variablen zur Verfügung. Variablen entstehen durch Zuweisung:

```
% home=this_is_my_home
```

Den Inhalt einer Variablen erhält man durch die Anwendung des $-Operators:

```
% echo $home
this_is_my_home
```

Anders als in der Bourne-Shell betrachtet *rc* den Wert einer Variablen als Liste, die Zeichenketten und weitere Listen enthält. Die Listen können Zeichenketten oder Variablen, sprich Listen, enthalten. Variablen entstehen durch Zuweisung und werden über die Operatoren *$var* (Werteliste), *$#var* (Anzahl Listenelemente), *$var(n)* (*n*tes Listenelement) und *$"var* (Wert als ein String) angesprochen. Geht die Zuweisung einem Kommando unmittelbar voraus, bleibt der Wert der Variablen lokal.

Beispiele

Zuweisung und anschließender Zugriff auf eine Variable:

```
% yacc=(yet another compiler compiler)
% echo $yacc
yet another compiler compiler
```

Zugriff auf die Elemente einer Liste in umgekehrter Reihenfolge. Zwischen dem Namen der Variablen und der linken Klammer darf sich kein Leerzeichen befinden:

```
% echo $yacc(4 3 2 1)
compiler compiler another yet
```

Anzahl der Listenelemente:

```
% echo $#yacc
4
```

Zuweisung einer Liste als Zeichenkette:

```
% new_yacc=( `{  echo $"yacc } )
% echo $#new_yacc
1
```

»$"<<« sorgt dafür, daß der Wert der Variablen als ein String betrachtet wird. »`{ ... }« wird vor der Zuweisung durch die Ausgabe des Kommandos innerhalb der geschweiften Klammern ersetzt.

Listen in Listen:

```
% a=( ( A AA ) ( a aa ) )
% b=( ( B BB ) ( b bb ) )
% c=( $a $b )
% echo $c
A AA a aa B BB b bb
```

So kann dies auch unter Verwendung von einfachen Anführungszeichen erfolgen:

```
% btw='by the way'
% echo $btw $#btw
by the way 1
```

Allerdings enthält die auf diesem Wege erzeugte Variable genau einen String. Ein weiterer subtiler Unterschied zwischen den beiden Methoden zeigt das nachfolgende Beispiel:

```
% nil=''   echo $#nil
0
% empty=() echo $#empty
1
```

Im ersten Fall wird die Variable angelegt, enthält allerdings keinen Inhalt; im zweiten Fall enthält die Variable eine leere Zeichenkette. Variablen, denen noch kein Wert zugewiesen worden ist, haben den Wert ().

Wird eine Variable definiert, wird unter Plan 9 eine Datei im Environment-Server »#e« mit dem Namen der Variablen angelegt. Der Inhalt der Datei entspricht dem Wert der Variablen. Die Antwort auf die Frage, ob eine bestimmte Umgebungsvariable im Namensraum vorhanden ist, reduziert sich auf ein schlichtes *ls /env/name*.

```
% cd /env
% lc
cputype  font   objtype  sec    terminal ipaddr   sysname
...
% echo cputype
sparc
% echo sysname
garm
% echo ipaddr
131.173.161.111
```

Da Dateinamen keine Leerzeichen enthalten können, sollten auch Variablennamen keine Leerzeichen enthalten. Die unausgesprochene Pointe ist aber, daß fast alles andere geht!

Um den Zugriff auf eine Variable zu beschleunigen, puffert *rc* den Wert der Variablen im Hauptspeicher. Erst bei Aufruf einer neuen Shell, die denselben Namensraum haben soll wie die ursprüngliche Shell, wird der Environment-Server bemüht. Somit sind die nachfolgenden Effekte leicht erklärbar.

```
% ok='value von ok'
% ls -l /env/ok; echo $ok
--rw-rw-rw- e 0 bischof bischof 13 Jan  3  1993 /env/ok
value von ok
% cat /env/ok%
```

Man kann jedoch auch problematische Namen erzeugen:

```
% 'not ok'='value von not ok'
% ls -l /env/'not ok'
rc: can't open #e/not ok
ls: /env/not ok: bad character in file name
% echo $'not ok'
value von not ok
```

Die erste Fehlermeldung stammt vom Export-Versuch in *rc*; die zweite zeigt, daß auch *ls* seine Probleme mit der Variablen hat. In *rc* ist die Variable vorhanden. Normalerweise werden Variablen exportiert, da sie in »#e« leben, aber bei problematischen Namen geht dies eben nicht.

```
% rc
% echo $ok
value von ok
% echo $'not ok'

%
```

»$« ist ein Operator, der für einen Variablennamen den zugehörigen Wert liefert. Somit sind Verweisketten möglich, wie das nachfolgende Beispiel zeigt:

```
% a=b                # a erhaelt den Wert b
% b='that is b'      # b erhaelt den Wert 'that is b'
% echo $$a           # ausgewertet wird,
that is b            #  solange dies moeglich ist
```

rc verwaltet verschiedene Variablen selbst:

***	Argumentliste der Kommandozeile
apid	Die Prozeß-Id des zuletzt in den Hintergrund geschickten Prozesses
home	voreingestellter Katalog für *cd*
ifs	*input field separator*
path	Suchpfad
pid	Prozeßid der aktuellen Shell
prompt	Die erste Komponente der Variablen wird ausgegeben, bevor ein Kommando eingelesen wird, die zweite Komponente wird am Anfang jeder weiteren Zeile eines Kommandos ausgegeben.
status	Wird mit der *wait message* des zuletzt terminierten Prozesses initialisiert. Ein Kommando war erfolgreich, falls *$status* leer ist. Verläuft ein Kommando erfolgreich, enthält *status* eine leere Zeichenkette.

```
% echo 'Hi'
Hi
% echo '---'$status'+++'
---+++
% cat no_exist
cat: can't open no_exist: file does not exist
% echo $status
cat 2538:error
```

Werden Kommandos mittels Pipes verknüpft, liefert *rc* den Status-Wert der einzelnen Kommandos auf einer Zeile. Die Status-Werte der Kommandos sind durch ein »|«-Symbol voneinander abgetrennt. Verläuft jedes Kommando korrekt, wird nur das »|«-Symbol als Status-Wert geliefert.

```
% who | wc > /dev/null
%  echo $status
|
% ls x | cmp xx xxx | wc
xx: file does not exist
      0       0       0
% echo $status
rc 89:ls 91:sys: write on closed pipe pc=0x3188|cmp 92:open|
```

Im weiteren wird auf die Verwendung der Variablen näher eingegangen.

6.4 Kommandoersatz

Es ist oftmals sehr nützlich, eine Argumentliste aus der Ausgabe von Kommandoabläufen erstellen zu können. *Rc* erlaubt an jeder Stelle, an der ein Argument stehen kann, ein Kommando, das von

```
`{ ... }
```

umgeben sein muß.

```
% echo heute ist `{date}
heute ist Thu Jul 18 10:41:34 MET DST 1996
```

Die Ausgabe des Kommandos wird an allen Zeichen, die in der Variablen *ifs* abgelegt sind, in einzelne Wörter zerlegt. Möchte man aus aus der Ausgabe von *date* die Sekunden extrahieren, so kann man dies u.a. durch setzen von *ifs* erreichen.

```
%  echo `{ ifs=' :'
    d=`{ date } ; echo $d(6) Sekunden }
51 Sekunden
```

date liefert eine Ausgabe der Form »Tue Oct 6 18:43:15 GPT 1994«. Wird diese Zeichenkette am Leerzeichen und Doppelpunkt in einzelne Wörter zerlegt, enthält das sechste Wort die Anzahl Sekunden.

6.5 Argumente

Ein *rc*-Skript kann mit Argumenten aufgerufen werden. Die Argumente werden in der Variablen *** abgelegt. Der Zugriff auf die Argumente erfolgt über *$*(i)* bzw. *$i* mit $1 \le i \le \$\#*$. *$0* enthält den Namen des Skripts. Das »alte« Shell-Problem, »wie oft müssen Sonderzeichen in Argumenten geschützt werden, wenn diese wieder als Argumente für ein anderes Skript dienen sollen« wurde, wie das nachfolgende Beispiel zeigt, in *rc* eliminiert.

```
% cat narg                ##########
#!/bin/rc

echo $0   ': # arguments: ' $#*
echo '$*(1): '             $*(1)
echo 'next: arg $*'
arg $*

% cat arg                 ##########
#!/bin/rc

echo $0    ': # arguments : ' $#*
echo '$*(1): '             $*(1)
echo '$1:    '             $1

% narg 'it''s nice'       ##########
./narg : # arguments:  1
$*(1):  it's nice
next: arg $*
./arg : # arguments :  1
$*(1):  it's nice
$1:     it's nice
```

Möchte man eine Operation auf alle Argumente der Kommandozeile anwenden, ist keine Schleife nötig, denn alle Argumente sind via *echo $** erreichbar. Beispielsweise wandelt das Skript *cla* in allen Argumenten alle Groß- in Kleinbuchstaben und löscht alle Nichtbuchstaben.

```
#!/bin/rc
#
# fold case, delete funny chars

echo $0    ': arguments  before: ' $*       # vorher
*=`{echo $*|tr A-Z a-z|tr -dc 'a-z \012'}   # Kommandoersatz
echo $0    ': arguments   after: ' $*       # nacher
```

Ein Ablauf:

```
% cla AbCdEf GhIj+-!
./cla : arguments  before:  AbCdEf GhIj+-!
./cla : arguments   after:  abcdef ghij
```

6.6 Verkettung von Listen

In *rc* enthalten Variablen Listen aus Zeichenketten. Variablen können als Ganzes oder komponentenweise, das heißt distributiv, verkettet werden. Bei einer komponentenweisen Verkettung müssen beide Listen die gleiche Anzahl Elemente enthalten, bzw. eine Liste darf nur aus einem Element bestehen. Das Resultat der Verknüpfung ist eine Liste.

```
% echo ( a aa ) ^ ( b bb )
ab aabb
% echo ( a aa ) ^ ( b bb ) ( c cc )
ab aabb c cc
% echo ( a aa ) ^ ( b bb ) ^ ( one_element )
abone_element aabbone_element
```

Ein Variable, die eine Liste enthält, kann unter Verwendung des »"«-Operators als ein Wort verwendet werden. Somit können Listen aus anderen Listen, die in Variablen abgelegt sind, zusammengefügt werden.

```
% a=(a bb)
% b=(C DD)
% echo $a ^ $b
aC bbDD
% echo $"a ^ $"b
a bbC DD
```

Wird der Verknüpfungsoperator weggelassen, werden Listen bzw. die Werte von Variablen aneinandergefügt.

```
% a=(A AA)
% b=(B BB)
% echo $"a $"b
A AA AAA B BB BBB
% echo ( aaa bbb ) ( ccc ddd )
aaa bbb ccc ddd
% echo ( aaa bbb ) $"b
aaa bbb B BB
```

6.7 Dateiverbindungen

Wie unter Unix sind die Filedeskriptoren 0, 1 und 2 mit *stdout, stdin* und *stderr* verbunden.

Standardeingabe und Standardausgabe werden mit '<' bzw. '>' umgelenkt. Hier-Dokumente erstrecken sich von Marke zu Marke, wobei kein Textersatz im Inhalt des Hier-Dokuments stattfindet, falls die erste Marke von einfachen Anführungszeichen umgeben ist. Kommandoersatz findet in keinem Fall statt. Kommandos, die sich über mehrere Zeilen erstrecken, werden von *rc* eingerückt.

```
% hi=(Moin Moin)
% cat << end
$hi nice world. Date: `{ date }
end
Moin Moin nice world. Date: `{ date }
% cat << 'end'
$hi nice world.
end
$hi nice world.
```

Der Zugriff auf die Filedeskriptoren erfolgt über eine ungewohnte Syntax. In diesem Beispiel wird die Diagnose-Ausgabe (*fd* 2) in eine Datei namens *error* gelenkt.

```
% rm x >[2]  error
% cat error
rm: x: file does not exist
```

Das Schließen eines Filedeskriptors erfolgt durch »>[fd-Nummer=]«.

```
% rm ex  >[2=]
```

Das Kommando

```
% wc ex ex_not_exist >[2=1] > wc_out
```

lenkt zuerst den Filedeskriptor 2 zu einer Kopie von 1, also zum Terminal, dann erst den Filedeskriptor 1 in *wc_out*.

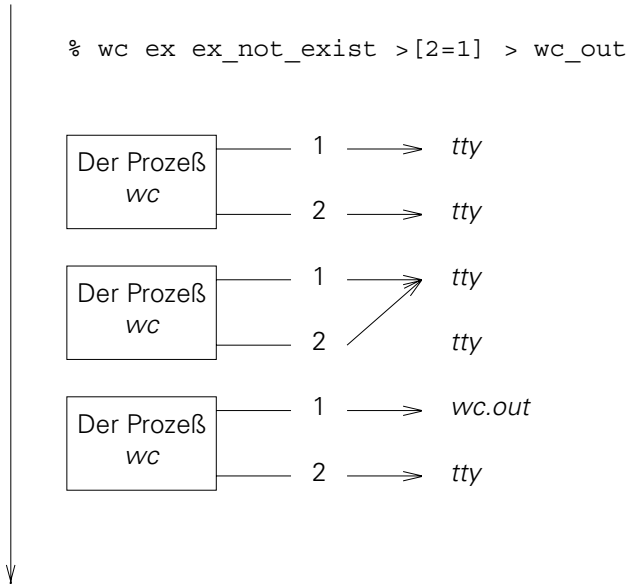
```
% wc ex ex_not_exist > wc_out >[2=1]
```

lenkt zuerst den Filedeskriptor 1 in *wc_out* und dann den Filedeskriptor 2 auch dorthin.

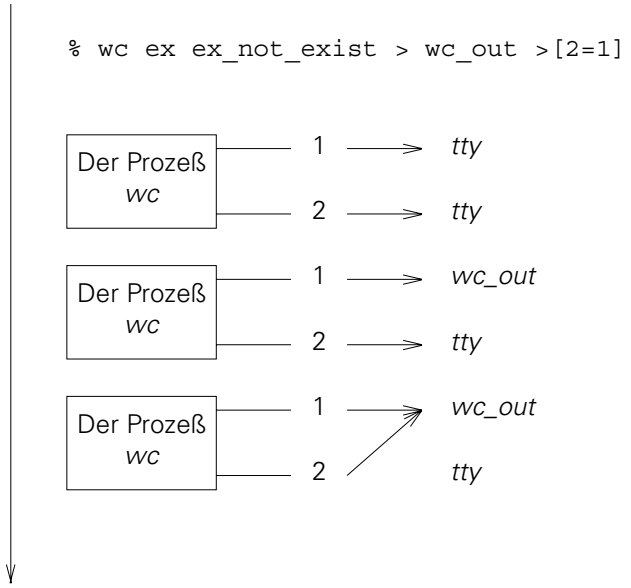
Die Syntax *>[2=1]* geht auf den Systemaufruf *dup()* zurück, d.h., ein Filedeskriptors wird dupliziert. Zur Umsetzung der Syntax *>[1]* ist hingegen eine Kombination von *close()* und *create()* erforderlich, .d.h., so daß eine, eine eventuell zuvor konstruierte Kopie davon unbeeinflußt bleibt.

Die Syntax *'>>'* führt zu *close()/open()*, also zum Anfügen an eine existente Datei, falls vorhanden.

Die beiden Umlenkungen sind in den nachfolgenden Abbildungen graphisch dargestellt.



Abarbeitung der Zeile von links nach rechts



Abarbeitung der Zeile von links nach rechts

Die von UNIX bekannte Pipeline-Verknüpfung zwischen zwei Kommandos ist auch mit *rc* nutzbar. Mit der Sequenz

```
% ls -l | grep '^-' | awk ' { print $6, $NF } ' |
            sort -nr | awk ' { print $2 } '       | sed 3q
xxx
xx
x
% echo $status
|
```

werden die drei größten Dateien im aktuellen Arbeitskatalog ermittelt. Der Wert von *status* sagt aus, daß die Pipe-Kette korrekt abgearbeitet worden ist.

Rc bietet allerdings einige Erweiterungen. Die Syntax

$$cmd_1 \;\; |[n=m] \;\; cmd_2$$

besagt, daß $Filedeskriptor_m$ von cmd_1 wird $Filedeskriptor_n$ von cmd_2.

$$cmd_1 \;\; |[0=m] \;\; cmd_2$$

ist äquivalent zu

$$cmd_1 \;\; |[m] \;\; cmd_2 \; .$$

Das Kommando

```
%  ls -l exist not_exist |[0=2] wc -l
--rw-rw-rw- M 13 none sys 0 Oct  7 19:17 exist
       1
```

zeigt existente Dateien mit *ls* an, nichtexistente Dateien werden gezählt. Der Filedeskriptor 0 (*stdin*) von *wc* wird mit dem Filedeskriptor 2 (*stderr*) von *ls* verbunden.

Filedeskriptoren können durch »[n=]« geschlossen werden, dies verhindert eine Ausgabe auf den Filedeskriptor *n*. Die Fehlerausgabe von *troff*

```
% troff tr > tr.out
troff: can't open file ooh; line 4, file tr
% troff tr >[2=] > tr.out
```

kann so z.B. »ignoriert« werden. Dies funktioniert aber nur, weil *troff* nicht kontrolliert, ob Fehlermeldungen erfolgreich ausgegeben werden konnten.

Falls die Kommandos ihre Eingabe aus Dateien lesen, kann anstelle einer Datei ein Kommando der Form *<{ command }* stehen. Ein Beispiel:

```
% cmp <{date} <{date}
% cmp <{date} <{ sleep 1 && date }
/fd/7 /fd/6 differ: char 19
```

Man sieht, daß bei den letzten Kommandos die Dateiverbindung über zwei Filedeskriptoren aufgebaut wird, das heißt, '<' und '>' haben nichts mit Dateiumlenkung zu tun, sondern es werden unter Verwendung des Servers »#d« Namen erzeugt und benutzt. Auf die Vergabe der beiden Filedeskriptoren hat man leider keinen Einfluß.

```
% who
bischof
bootes
none
```

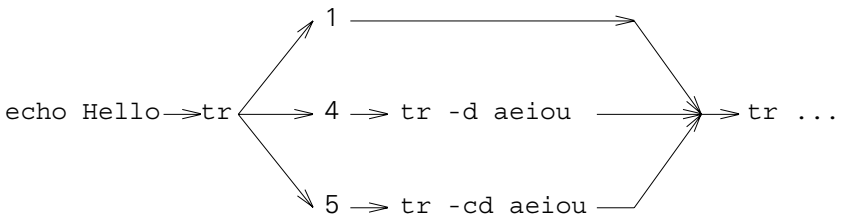
```
% sort -o >{tr  a-zA-Z A-Za-z}          \
        <{who | grep -v bischof } <{date | tr 0-9 A-Z }
% wED jUL cc bf:dc:cb gpt cacg
BOOTES
NONE
```

In dem vorigen Beispiel besteht die Eingabe für *sort* aus zwei '<{ ... }', die Ausgabe von *sort* wird von *tr* bearbeitet.

Verknüpft man die Techniken, können Kommandosequenzen wie die folgende geschrieben werden.

```
% echo Hello |
            tee /fd/4 /fd/5 |[4] tr -d aeiou |[5]
            tr -cd aeiou | tr a-z A-Z
HELLO
HLL
EO%
```

Das letzte Beispiel ist in der nachfolgenden Graphik visualisiert.



6.8 Kommandos zusammenfassen

Eine Folge von Wörtern ist ein Kommando. Zuweisungen können vor der Folge und E/A-Umlenkungen können vor und in der Folge angegeben werden. Ist das Kommando leer, sind die Zuweisungen global, und Ausgabedateien werden erzeugt: Mit «{« »}« werden Kommandos zusammengefaßt. Durch Angabe von »!« vor einem Kommando wird der Status des Kommandos invertiert. Beginnt ein Kommando mit einem »@«, wird das Kommando in einer Subshell abgearbeitet.

```
% { echo hi; date  } | wc
      2       7      32
%  date | wc; echo $status
      1       6      29
|
%  date | ! wc; echo $status
      1       6      29
|false
% a=42; echo $a
42
% @a='hal'; echo $a
42
```

6.9 Kontrollstrukturen

Es gibt eine Entscheidung mit *if* sowie *for*- und *while*-Schleifen und eine Auswahl mit *switch*. Als Bedingung gelten ein $status aus 0 und »|« sowie leere Zeichenketten als wahr, jeder andere als falsch.

```
for ( name in list ) command
for ( name ) command
```

Im ersten Fall wird *command* für jedes Element der Liste ausgeführt. Mit *$name* kann im Kommando auf den Wert des Listenelements zugegriffen werden. Im zweiten Fall wird *command* für jedes Element der Liste ausgeführt, welche die Variable »*« enthält. Wird ein *rc*-Skript mit Argumenten aufgerufen, sind diese im Skript über die Variable »*« zugänglich.

```
% *=( es 'ist an' 'der Zeit' )
% for ( i ) echo $i
es
ist an
der Zeit
```

Die nachfolgende *for*-Schleife kopiert alle *rc*-Skripte aus */rc/bin*, welche mit einem Buchstaben aus der Menge { *a, b, e, r* } beginnen, in den aktuellen Katalog unter modifiziertem Namen:

```
% for ( i in `{ls /rc/bin/[aber]*} )
        cp $i `{basename $i} ^ .orig
% lc *orig
CC.orig       broke.orig   cc.orig       false.orig   psh.orig
ar89.orig     bundle.orig  chown.orig    grep.orig    rmdir.orig
art2pic.orig  c89.orig     eject.orig    install.orig true.orig
```

if

rc bietet keine direkte *if — else* Kontrollstruktur. Liefert *commandlist* falsch und folgt dem *if* ein *if not*, wird das dem *if not* zugeordnete Kommando ausgeführt.

```
if ( commandList ) command_if     # command)if falls wahr
if not command_if_not             # command_if_not falls nicht
```

```
% for ( i )      {
> if ( test -f /tmp/$i ) echo $i exists
> if not cp $i /tmp/
> }
```

kopiert jede Datei aus »$*« nach */tmp*, falls sie dort nicht schon vorhanden ist.

```
if ( test 2 -lt `{ who | wc -l } ) tetris
if not echo 'Sorry, try again'
```

aktiviert *tetris*, falls die Anzahl angemeldeter Nutzer kleiner als zwei ist.

Das Kommando »~« vergleicht ein Wort mit einem Muster und setzt einen entsprechenden Status-Wert.

```
%  ~ date d*e; echo '--'$status'++'
--++
%  ~ date d*w ; echo $status
no match
% if ( ~ $file *.c ) $CC $file.c
```

aktiviert den C-Compiler für *$file* nur, falls *$file* mit ».c« endet. Der »~«-Operator erlaubt auch eine oder-Verknüpfung von Mustern.

```
% if ( ~ $file *.c *.m ) $CC $file
```

aktiviert den C-Compiler für *$file* nur, falls *$file* mit ».c« oder ».m« endet.

while

Die *while*-Schleife hat folgende Struktur:

```
while ( commandlist ) command
while ( ) command
% while ( ! test -f waiting ) sleep 5
```

Die Schleife terminiert erst, wenn die Datei *waiting* angelegt worden ist.

switch

switch wählt unter mehreren Mustern:

```
switch ( word ) {
       case pattern...
              commandlist
       ...
}
% while ( ~ $1 -* ) {
>          switch ($1)      {
>                   case -x
>                            xFlag=1
>                   case -*
>                            echo 'unexpected flag' $1
>          }
>
>          shift
> }
```

Solange die Elemente der Variablen »*« mit einem Minuszeichen beginnen, wird die Liste abgearbeitet. Unerwartete Flaggen werden zurückgewiesen.

6.10 Funktionen

Funktionen werden als */env/fn#name* im Environment gespeichert. Sie erhalten ihre Argumente als *$**, wobei aber die ursprüngliche Liste nach dem Aufruf wiederhergestellt wird. Notes rufen Funktionen mit besonderen Namen auf (siehe *rc(1)*). *sigexit* wird am Schluß von *rc* aufgerufen. Eine Funktion definiert für jeden angegebenen Namen eine neue Kopie des Funktionskörpers. Fehlt der Körper, werden die Funktionen gelöscht.

```
fn name [name_2 ...] {     # anlegen
    commandlist }
fn name [name_2 ...]       # loeschen
```

Eine mehrspaltige Auflistung aller Dateien und Kataloge mit ihrer Größe liefert

```
% fn sm { ls -l | awk ' { print $6, $NF }' | mc }
% sm
7 access.c  54 frodo.h   150 mk_D_1   95 sig_exit
```

An eine Funktion können Argumente übergeben werden.

```
% fn arg { echo $#*; if ( ~  $#* 1 )
> echo Ein Argument ($1) wurde uebergeben
> }
% arg
0
% arg one
1
Ein Argument one wurde uebergeben
% arg one two
2
```

Die Anzahl der übergebenen Argumente wird mit Hilfe des »#«-Operators bestimmt. Falls genau ein Argument übergeben worden ist, wird dieses ausgegeben.

6.11 Nachrichten

Ein *rc*-Skript terminiert normalerweise durch das Eintreffen einer Nachricht (*note*). Ist eine Funktion mit dem Namen der entsprechenden Nachricht in Kleinbuchstaben definiert, wird diese aufgerufen, wenn die Nachricht eintrifft. Von Interesse sind:

`sighup`	*hangup*. Das kontrollierende Terminal ist verlorengegangen.
`sigint`	Das *interrupt*-Zeichen (*NumLock*) wurde am kontrollierenden Terminal getippt.
`sigquit`	Das *quit*-Zeichen (^-) wurde am kontrollierenden Terminal getippt.
`sigterm`	Diese Nachricht wird nomalerweise von *kill(1)* geschickt.
`sigexit`	Diese künstliche Nachricht wird geschickt, wenn *rc* terminiert.

Ist nur »{ }« als Funktionskörper gesetzt, wird die entsprechende Nachricht ignoriert.

```
% cat sig_exit
#!/bin/rc

fn sigexit { echo $0  ": sigexit" }

echo  $0 ^ ': chrr chrr ... '
sleep 10
exit ''
% sig_exit
./sig_exit: chrr chrr ...
/rc/lib/rcmain ": sigexit"
```

Das Skript vereinbart, daß bei Eintreffen der *sigexit*-Nachricht ein Text ausgegeben wird. Nach 10 Sekunden terminiert das Programm, und die Nachricht *sig_exit* wird an das Skript geschickt.

Da man gleichen Funktionen mehrere Namen geben kann, kann man den gleichen Handler für verschiedene Nachrichten verwenden. Es bleibt offen, ob man verschachteln kann und wie weit Variablen bekannt sind etc.

6.12 Eingebaute Kommandos

Ein Kommandoname wird nacheinander unter den Funktionen, dann unter den eingebauten Kommandos und schließlich absolut, wenn er mit »/« beginnt, oder relativ zu den Elementen von *path* gesucht. Dadurch kann sehr leicht eine Paketierung von Kommandos erreicht werden. Die Kommandos zur Verwaltung der Netzwerkdatenbank befinden sich alle unter */bin/ndb*.

```
% echo $path
. /bin
% ndb/query sys garm
sys=garm dom=garm.informatik.uni-osnabrueck.de
bootf=/sparc/9ss
ip=131.173.161.111 ether=080020038625 proto=il
fs=bliss
sys=garm ip=131.173.160.24
```

Eine Datei wird im aktuellen Kommandoprozessor mittels des ».«-Kommandos ausgeführt. Variablen, Funktionen etc., die in der Datei initialisiert werden, sind dann im aktuellen Kommandoprozessor zugänglich.

```
% . $home/lib/profile
```

Werden eingebaute Kommandos, wie z.B. *cd*, durch Funktionen überdeckt, können diese immer noch mit Hilfe des *builtin*-Kommandos angesprochen werden.

```
% cd .
% fn cd { builtin cd $* && pwd }
% cd .
/usr/bischof/tmp
```

eval verkettet wie *$"* und bewertet das Ergebnis nochmals.

```
% w=world x='$'w echo hello $x
hello $w
% w=world x='$'w eval echo hello $x
hello world
```

shift löscht einzelne Elemente aus der Argumentliste »*«.

```
% *=(1 2 3)
% shift; echo $*
2 3
% shift 2; echo $#*
0
```

Mit *wait* kann man auf das Ende eines bestimmten Prozesses oder auf die Terminierung aller Prozesse warten.

```
% sleep 1 & wait $apid
% sleep 1 & sleep 2 & sleep 3 & wait
```

Die Frage »wo findet *rc* ein Kommando«, klärt *whatis*.

```
% whatis  path cd shift who
path=(. /bin)
fn cd {builtin cd $* && pwd}
builtin shift
/bin/who
```

Mit *exit* wird der aktuelle Kommandoprozessor zur Terminierung gezwungen. *exec* führt ein (nicht eingebautes) Kommando anstelle von *rc* aus. *rfork* startet eine neue Prozeßgruppe, deren *rfork*-Parameter einstellbar sind. Dieses Kommando ermöglicht es, *fork(2)* in Teilen interaktiv zu testen. Die E-Flagge (RFCENVG) startet die neue Prozeßgruppe mit einem leeren Environment.

```
% old_env='hi'
% ls /env/o*
/env/old_env
% rfork E
% ls /env
/env/status
```

Beispiel: Endlosschleife

```
% while ( !  ~ * ) date
Tue Oct  6 15:28:21 GPT 1994
...
% while ( ) date
Tue Oct  6 15:28:42 GPT 1994
...
```

Beispiel: chop

chop liefert eine um ein Element verkürzte Liste.

```
fn chop {
    ~ $#* 0 1 ||
    ans = () {
        while(! ~ $#* 1) {
            ans = ($ans $1)
            shift
        }
        echo $ans
    }
}
% x = (a b c d)
% while (x = `{chop ($x)}; ! ~ $#x 0) echo $x
a b c
a b
a
```

Beispiel: show_args

Wie Argumente abgearbeitet werden können, zeigt das nachfolgende Skript.

```
#!/bin/rc

fn usage {
    echo 'Sorry, I dont understand '$1'.'
    exit 'something failed'
}

while ( ~ $1 -* )   {
    switch( $1 )    {
        case -d
                echo 'dFlag=0'
            case -?
                usage $1
    }
    shift
}

while (  ~ $1 *.* ) {
    switch ( $1 )   {
            case *.c
                echo 'Found C source ('$1').'
            case *.h  *.m
                echo 'Found *.[mh]'$1'.'
            case *
                echo 'Found unspecified names.'
        }
    shift
}

exit
```

Beispiel: setCd

Das folgende Beispiel setzt den aktuellen Katalog im Prompt. Um CPU-Ressourcen zu schonen, sollte *pwd* nicht immer bemüht werden. Pfade, die mit ».« bzw. »..« beginnen, werden nicht optimal behandelt, was das Sparen von CPU-Ressourcen betrifft. Die Funktion sollte wohl Teil von *$home/lib/profile* bzw. */rc/lib/rcmain* sein, damit sie permanent verfügbar ist.

```
#!/bin/rc

ps1 = ' '       # prompt if at home
ps2 = '> '      # secondary prompt

fn pbd {        # print basename of $1
            # or / if $1  == /
    switch ( $1 )   {
    case '/'
        echo '/'
```

```
    case *
        basename $*(1)
    }
}
fn do_cd {      #  cd $1 && set prompt
    builtin cd $1
    if ( ~ $status '' ) {
            ps1=`{ pbd $dir } ^ ' '
            prompt=( $ps1 $ps2)
    }
    if (!  ~ $status '' ) {
    dir=$dir_was
    }
}
fn cd {         # cd with pwd in prompt
    dir_was=$dir
    switch ($#*) {
    case 0      # cd alone goes home
        dir=$home
        do_cd $dir
    case 1
        switch ($1) {
        case /*
            dir=$1
        case .
            dir=$dir
            case ..*
                    dir=`{ builtin cd $1 && pwd }
        case *
            dir=$dir ^ '/' ^  $1
        }
        do_cd $dir
    case 2
        echo Grrrrr
    }
}
```

Beispiel: rcmain

Tom Duff's *rc* führt beim Start die Datei */rc/lib/rcmain* aus. Man entdeckt interessante Interna:

```
# rcmain: Plan 9 version
if(~ $#home 0) home=/
if(~ $#ifs 0) ifs='
'
switch($#prompt){
case 0
    prompt=('% ' '  ')
```

```
case 1
    prompt=($prompt '    ')
}
if(flag p) path=/bin        # –p ignoriert Environment
if not{
    finit                   # finit laedt Funktionen
    if(~ $#path 0) path=(.  # aus Environment
/bin)
}
fn sigexit                  # sigexit unbedingt geloescht
if(! ~ $#cflag 0){          # –c String ist cflag
    if(flag l && /bin/test –r $home/lib/profile) .
$home/lib/profile
    status=''
    eval $cflag
}
if not if(flag i){          # –l liest profile
    if(flag l && /bin/test –r $home/lib/profile) .
$home/lib/profile
    status=''
    .
–i '#d/0' $*                # –i: Standard–Eingabe interaktiv
}
if not if(~ $#* 0) .
'#d/0'                      # ohne Argumente: Standard–Eing.
if not{
    status=''
    .
$*                          # mit Argumenten: Skript
}
exit $status
```

6.13 Zusammenfassung

Es gibt Shells, die man als Entwickler nicht besonders mag, und andere, die man nach einer gewissen Gewöhnungsphase schätzen und lieben gelernt hat. *rc* fällt ohne jede Frage in die zweite Kategorie. Die klar strukturierte Syntax erleichtert das Lernen der Sprache außerordentlich. Die Gewißheit, daß jede Zeile genau einmal gelesen wird, ermöglicht es, Skripte zu schreiben, deren Komplexität nicht durch die Anzahl »#« auf einer Zeile festgelegt ist. Variablen als Listen verpackt mit den zugehörigen Operationen eröffnen interessante Lösungsmöglichkeiten.

rc erfüllt mit einer Ausnahme problemlos alle Anforderungen, die an einen Kommandoprozessor gestellt werden, mit Auszeichnung. Die *history*-Möglichkeit vermißt man auch nach längerem Arbeiten doch ab und an, wenn man sich an *set -i vi* gewöhnt hat. Mein Rat: Programmieren von Shell-Skripten in *rc* und Absetzen von Kommandos in der *bash*, sofern man sie hat.

7
make
unter Plan 9

Soll aufgrund der Altersverhältnisse zwischen Dateien eine Aktion angestoßen werden, ist die Zeit gekommen, Plan 9s *mk* einzusetzen. *mk* basiert auf derselben Idee wie Feldmans *make* [Fel83]: Ein Ziel entsteht aus Quellen durch Anwendung eines Rezepts. Ist eine Quelle neuer als das Ziel, wird das Ziel unter Anwendung einer Regel neu erstellt. Die vernünftige Verwendung von *make* bzw. *mk* garantiert, daß das Ziel in bezug auf die Quellen aktuell ist, wobei die minimale Anzahl Regeln abgearbeitet worden ist.

Unter Plan 9 gibt es kein Äquivalent zu *cc*, da alle Übersetzungen von *mk* gesteuert werden sollen. Die Rolle von *cc* wird von vorgegebenen Standard-*mkfiles* übernommen. Unter einer Plan 9-Umgebung ist es ein Leichtes, für unterschiedliche Architekturen zu übersetzen, weil jede generierte Datei eine architekturspezifische Endung bzw. Anfang bekommt. Generierte Dateien stören sich in einem Katalog also nicht.

mk verwendet ähnlich wie Todd Brunhoffs *imake* spezifische Regel- und Makrodateien und ist somit in seiner Funktionalität beliebig erweiterbar. Die Verwendung von *imake* erzwingt de facto, daß für einzelne Projekte spezielle Regel- und/oder Makrodateien implementiert werden, welche dann allerdings die Verwaltung eines Projekts sehr stark erleichtern. In großen Projekten, wie z.B. X11 Release 6, hat sich gezeigt, daß sich diese Technologie sehr stark arbeitserleichternd auswirkt [Du93]. Für *imake* werden die Regeln mit Hilfe des C-Präprozessors *cpp* definiert, das heißt, die Entwicklung einer Regeldatei erfordert exzellente *cpp*-Kenntnisse /Bi93/. Im Gegensatz dazu werden die Regeln für *mk* in der Sprache definiert, die verwendet wird, um ein *mkfile* zu definieren. Dies erleichtert den Lernaufwand erheblich.

7.1 Einführung

In diesem Kapitel wollen wir uns *mk* anhand von Beispielen nähern. Das Abhängigkeitsverhältnis von Zielen und Quellen wird durch Regeln definiert. Eine normale Regel hat die Form:

```
target:     sources
            recipe
```

Beispielsweise müßte man für eine Sparc-Architektur das Kommando *echo.c* mit den folgenden Befehlen übersetzen und binden:

```
% kc echo.c          # liefert echo.k
% kl echo.k          # liefert k.out
% k.out              # k.out ist ausfuehrbar.
echo ...
```

Für *mk* schreibt man folgendes:

```
# mk_0: das erste mkfile

echo.k: echo.c echo.h
    kc echo.c

echo:   echo.k
    kl –o echo echo.k
```

Das Objekt, *echo.k*, hängt von *echo.[ch]* ab und wird mit *kc*, dem Compiler für die Sparc-Architektur, übersetzt. Das Programm, *echo*, hängt vom Objekt, *echo.k*, ab und wird mit dem Lader für die Sparc-Architektur, *kl*, gebunden. Der Suffix des Objekts sowie der erste Buchstabe des Compilers bzw. Laders sind gleich und spezifisch für die Architektur, für die übersetzt bzw. gebunden wird. Falls ein Kommando eines Rezepts mit einem Fehlerstatus endet, terminiert *mk* als Konsequenz daraus. Ein Ablauf:

```
%  mk -f mk_0 echo
kc echo.c

kl -o echo echo.k
```

Normalerweise liest *mk* die Regeln aus der Datei *mkfile*; die Angabe *-f* entfällt dann. *mk_0* hat aber noch einen anderen entscheidenden Nachteil: So lassen sich nur Übersetzungen für die Sparc-Architektur vornehmen. Dies ist insbesondere deswegen sehr unglücklich, weil unter Plan 9 auf jeder Architektur für jede Architektur übersetzt werden kann. Man behebt dieses Manko, indem man dafür sorgt, daß Architektur-abhängige Kommandos durch Verwendung von Variablen wie *CC, LD* usw. verborgen werden. Die Variablen müssen für jede Architektur entsprechend initialisiert werden. Ein *mkfile* wird aufgeteilt in

- einen Projekt-unabhängigen, aber Architektur-abhängigen Teil, in dem i.a.R. nur Variablen initialisiert werden.
- einen Projekt-abhängigen, aber Architektur-unabhängigen Teil, in dem die Regeln spezifiziert werden.

Das nachfolgende *mkfile* ist für eine Architektur-unabhängige Übersetzung geeignet.

```
# mk_1: architektur–unabhaengiges mkfile

< /$objtype/mkfile

$O.echo:    echo.$O
    $LD $CFLAGS –o $O.echo echo.$O

echo.$O:    echo.c echo.h
    $CC echo.c
```

Mit dem Befehl '<' kann man eine Datei in ein *mkfile* einfügen. Da der Dateiname einer Makroexpansion unterliegt und da *mk* Environment-Variablen als Makros liest, verwendet man die Konvention

```
< /$objtype/mkfile
```

um Architektur-spezifische Abhängigkeiten zu zentralisieren. *objtype* wird beim Start von Plan 9 auf den Wert von *cputype* gesetzt, also mit dem Namen der aktuellen Architektur, wie z.B. *spac*, initialisiert. Das kann allerdings jederzeit überdefiniert werden. *$O* steht für *O*bjekt; diese Variable wird in */$objtype/mkfile* initialisiert. Unter Verwendung dieser Variablen werden Architektur-spezifische Abhängigkeiten festgelegt. Eine Übersetzung geht wie gewohnt vonstatten:

```
% mk -f mk_1
kc echo.c
kl  -o k.echo echo.k
```

Soll *echo* für eine andere Zielarchitektur übersetzt werden, genügt es, *objtype* mit dem Namen der entsprechenden Zielarchitektur zu initialisieren.

```
% objtype=mips mk -f mk_1
vc echo.c
vl  -o v.echo echo.v
```

Man sieht, daß die Architektur-spezifischen Teile, wie z.B. der Name des Compilers bzw. Laders, vollkommen verborgen bleiben. Da die Dateinamen-Endungen der übersetzten Objekte und der Dateinamen-Anfang des Ziels an die Architektur gekoppelt sind, kann parallel für jede unterstützte Architektur übersetzt werden. So einfach kann Cross-Compilierung sein.

Die Auswahl des Architektur-spezifischen Compilernamens, Laders etc. wird in */$objtype/mkfile* festgelegt; für jede unterstützte Architektur wird eine Reihe von Variablen gesetzt:

```
% cat /$objtype/mkfile
CC=kc               # C-Compiler
ALEF=kal            # alef-Compiler
LD=kl               # Lader
O=k                 # Objekt-Kennung
RL=rl               # ranlib
AS=ka               # Assembler
OS=2kvz8            # moegliche Objektkennungen
CPUS=mips 68020 sparc 386 hobbit # unterstuetzte Architekturen
CFLAGS=             # Flaggen fuer den C-Compiler
LEX=lex
YACC=yacc
MK=/bin/mk
```

Eine Zuweisung besteht aus dem Namen, »=« und dem Wert, der an Zwischenraum in Worte zerlegt wird. Variablen werden normalerweise ins Environment der Rezepte exportiert. Dies kann man unterbinden, indem man »=U=« anstelle des normalen Zuweisungsoperators »=« verwendet.

7.2 Meta-Regeln

Eine normale Regel hat die Form:

```
target ...:[attributes:]   prereq_1 prereq_2 ...
    recipe    # using prereq_1, prereq_2 ... to build target
```

Die Ziele werden unter der Verwendung des Rezepts erzeugt, falls ein Datum einer Quelle neuer ist als das Datum eines der Ziele. Das *recipe* beginnt mit Zwischenraum, dessen erstes Zeichen ignoriert wird. Es wird als Ganzes von */bin/rc -e -I* bearbeitet. Somit können die zugehörigen Aktionen, ohne Zeilenfortsetzung, auf beliebig viele Zeilen verteilt werden.

Abgesehen vom Environment und den von *$objtype/mkfile* exportierten Variablen sind im Rezept folgende Variablen ansprechbar:

newprereq	nur die neuen *prereq*
nproc	maximale Anzahl der parallelen Prozesse, die generiert werden, um das Ziel zu erstellen. Es können nicht mehr erzwungen werden, als CPUs vorhanden sind.
pid	Prozeß-Id
prereq	alle *prereq*
stem	Ersatztext von %
stem1 ... stem9	Ersatztext von \(... \) in regulären Ausdrücken
target	das Ziel

Für viele Übersetzungsvorgänge können allgemeine Regeln angegeben werden, wie z.B.: jedes Programm wird aus einem gleichnamigen Objekt und einer Bibliothek generiert. Das Objekt ist von einer gleichnamigen *include*-Datei abhängig. Da Regeln nicht bestimmte Übersetzungsvorgänge steuern, sondern andere Arten von Regeln definieren, werden sie *Meta-Regeln* genannt. Diese Meta-Regeln können keine festen Namen enthalten, sondern nur Platzhalter für Namen. Eine Meta-Regel verwendet ein Ziel mit einem »%«, das möglichst viele Zeichen erkennt, oder »&«, das möglichst viele Zeichen, aber weder ».« noch »/« erkennt.

In den *prereq* wird für »%« der Text ersetzt, der im Ziel für »%« oder »&« erkannt wurde. Im *recipe* wird dieser Text für *$stem* ersetzt. Ein Beispiel:

```
# mk_2: architektur—unabhaengiges mkfile, das
#       eine parallele Uebersetzung fuer alle
#       Architekturen in einem Katalog zulaesst.

</$objtype/mkfile

$O.%:   %.$O
    $LD $LDFLAGS —o $target $prereq

%.$O:   %.c %.h
    $CC $CFLAGS $stem.c
```

Mit diesem *mkfile* können alle die Programme, *cmd*, für alle unterstützten Architekturen parallel übersetzt werden, deren Objekte, *cmd.$0*, von *cmd.[ch]* abhängen und die keine zusätzlichen Systembibliotheken benötigen.

Ein Ablauf:

```
% mk -f mk_2 k.echo
       kc echo.c
       kl  -o k.echo echo.k
```

7.3 Reguläre Ausdrücke

Für viele Probleme ist es nicht ausreichend, daß eine Zeichenkette mittels des »%«-Operators in drei Teile zerlegt werden kann. Wesentlich flexibler, allerdings auch etwas schwieriger in der Nutzung, kann eine Zeichenkette unter Verwendung regulärer Ausdrücke zerlegt werden. Das Attribut »R« zeigt an, daß das Ziel einer Meta-Regel mittels regulärer Ausdrücke definiert wird. Die abgespaltenen Zeichenketten können im *prereq* bzw. im *recipe*-Teil verwendet werden. Kontext-Klammern können verwendet werden; ihre Resultate erhält man als *$stem1* usw. Ein Beispiel:

```
# mk_7: regulaere Ausdruecke in einer Meta—Regel

< /$objtype/mkfile

([^/]*)/(.*)\.$O:R:  \1/\2.c
    cd $stem1 && $CC $CFLAGS $stem2.c
```

Ablauf:

```
% mk -f mk_7 Echo/echo.k
cd Echo; kc  echo.c
```

7.4 Regeldateien

Betrachtet man genauer, was man gewöhnlich mit einem *mkfile* generiert, dann stellt sich heraus, daß sehr oft erstellt werden:

- ein einzelnes ausführbares Programm,
- viele ausführbare Programme,
- eine Bibliothek oder
- eine Systembibliothek.

Um dies zu erleichtern, stellt Plan 9 dem Entwickler Regeldateien zur Verfügung, die diese Routineaufgaben erledigen. Eine Anforderung an eine Regeldatei besteht natürlich darin, daß sie jederzeit erweitert bzw. modifiziert werden kann. Wie müßte wohl eine Regeldatei implementiert werden, die ein Objekt generiert, das sich aus beliebigen C-Quellen zusammensetzen kann? Die Regeldatei sollte folgende Ziele ermöglichen:

all	eine lokale Version von allen Zielen wird, falls möglich, generiert. Abgespeichert werden diese in *$O.progname*, wobei *progname* der Name der einzelnen Ziele ist. Jedes Produkt kann parallel für jede Architektur übersetzt werden.
install	übersetzt und installiert jedes Produkt für die aktuelle Architektur.
installall	übersetzt und installiert jedes Produkt für jede Architektur.
clean	soll den Platzverbrauch auf ein Minimum reduzieren. Im aktuellen Katalog und allen Unterkatalogen werden die generierten Zwischendateien gelöscht.
nuke	soll eine garantierte Neuübersetzung ermöglichen. Im aktuellen Katalog und allen Unterkatalogen werden die generierten Zwischendateien und Ziele gelöscht.

Ein Nutzer der Regeldatei sollte nur noch spezifizieren müssen, wie sein Ziel und die für die Übersetzung benötigten Objekte heißen; die Frage nach dem »wie wird das Objekt erstellt« erledigt die Regeldatei. Damit die Regeldatei benutzt werden kann, muß man sich über Variablennamen einigen, in denen das Ziel und die zu generierenden Objekte abgelegt werden. Unter Plan 9 lauten die Variablen *TARG* und *OFILES*.

Der zentrale Teil der Regeldatei wird aus *mk_2* bestehen. Man sieht allerdings an dieser Stelle sehr deutlich, daß die Abhängigkeit der Objekte von beliebigen *include*-Dateien im allgemeinen nicht aufgeschrieben werden kann. Aus diesem Grunde wird vorgeschrieben, daß zu jedem Objekt eine gleichnamige *include*-Datei existiert. Die nachfolgende Regeldatei ist das Resultat:

```
# mkone_hp:

all:V:  $O.$TARG        # default Ziel, architektur—abhaengig

install:V:  $BIN/$TARG  # fuer $cputype
                        # :V: wird in 7.6 erklaert

$O.$TARG:   $OFILES
    $LD —o $target $prereq

%.$O:   %.c %.h         # .c & .h
    $CC $stem.c

$BIN/$TARG: $O.$TARG
    cp $prereq $BIN/$TARG

installall:             # fuer alle Architekturen
    for(objtype in $CPUS)
        mk install

nuke:                   # rm
    rm —f *.[$OS] [$OS].$TARG $TARG

clean:                  # rm
    rm —f *.[$OS] [$OS].$TARG
```

Damit die generierten Ziele für die unterschiedlichen Architekturen installiert werden können, muß *BIN* korrekt initialisiert sein. In aller Regel ist diese Variable, gemäß Plan 9 Konvention, mit dem Wert von *$objtype/bin* initialisiert. Mit dieser Regeldatei kann *mk_2* wesentlich vereinfacht werden.

```
# mkfile: Verbesserung vom mk_2

< /$objtype/mkfile

TARG=frodo
OFILES=frodo.$O

< mkone_hp
```

Ablauf:

```
% mk install
kc frodo.c
kl -o k.frodo frodo.k
cp k.frodo /sparc/bin/frodo
```

```
% mk installall
for(objtype in mips 68020 sparc 386 hobbit)
        mk install
vc frodo.c
vl -o v.frodo frodo.v
cp v.frodo /mips/bin/frodo
2c frodo.c
2l -o 2.frodo frodo.2
...
```

Besteht der Wunsch nach einem veränderten Rezept für eine Regel, so kann dies nach der '<'-Zeile im *mkfile* erfolgen, weil *mk* zuerst die kompletten Definitionen von Variablen und Zielen mit ihren zugehörigen Regeln einliest und erst dann ausführt. Nachfolgende Definitionen überschreiben demzufolge vorangegangene.

mkone_hp erfüllt nicht alle Wünsche eines Entwicklers. So werden Werkzeuge wie *yacc* und *lex* genausowenig wie ein Assemblerlauf berücksichtigt.

7.5 Struktur eines Plan 9 mkfiles

Nach Plan 9-Konvention sollte ein *mkfile* folgender Form genügen:

`< /$objtype/mkfile`

Projekt und Architektur-spezifische Variablendefinitionen

lokale Definitionen

`< /sys/src/cmd/`*generic*

Projekt-spezifische Regel-Dateien

zusätzliche Regeln

Aufgrund dieser Struktur zerfällt ein *mkfile* in folgende Komponenten:

- Architektur-spezifische Variablendefinitonen, die in der Regel vorgegeben sind.
- Architektur- und Projekt-spezifische Variablendefinitonen, die für das Projekt entwickelt worden sind.
- sehr spezielle Variablendefinitonen, die für bestimmte Teile des Projekts benötigt werden.
- eine allgemeine Regeldatei, die in der Regel vorgegeben ist.
- eine Projekt-spezifische Regeldatei, die für das Projekt entwickelt worden ist.
- sehr spezielle Regeln, die für bestimmte Teile des Projekts benötigt werden.

7.6 Attribute

Die Ausführung einer Regel kann durch die Angabe von Attributen gesteuert werden. Es können mehrere Attribute zwischen den Zielen folgenden Doppelpunkten angegeben werden. Folgende Attribute sind von Interesse:

V Die Ziele dieser Regel sind als virtuell markiert. In aller Regel werden künstliche Ziele, wie z.B. *install*, benutzt, um einheitliche Namen für ähnliche Aktionen verwenden zu können. Das Ziel *install* sollte ein Produkt installieren, welches und wie ist normalerweise nicht von Interesse.

D Endet das Erstellen des Ziels mit einem *non-null* Exit-Status, wird das Ziel, falls möglich, gelöscht. Angenommen, ein Ziel hängt von einer Zwischendatei ab:

```
$O.to_generate:D:   to_generate.c
    $CC to_generate.c
    $LD -o $O.to_generate.c to_generate.$O

to_generate.c:D:    first second
    cat first second > to_generate.c
```

Verläuft *cat* fehlerhaft, würde *to_generate.c* übrigbleiben, da diese Datei mittels Standardausgabe außerhalb von *cat* angelegt wird. Ändert man *first* und *second* nicht ab, wird *cat* beim nächsten Aufruf von *mk* nicht ausgeführt, und das Ziel entsteht aus einer fehlerhaften Version von *to_generate.c*.

```
# mk_D: das Ziel wird entweder korrekt oder nicht erzeugt.

to_generate:    to_generate.c
    $CC to_generate.c
    $LD -o $O.to_generate.c to_generate.$O

to_generate.c:  first second
    cat first second > to_generate.c
```

E Falls der Generierungsversuch eines Ziels fehlschlägt, wird das *mkfile* weiter abgearbeitet. Dieses Attribut würde man für Ziele verwenden, deren Generierung fehlschlagen kann, ohne daß das böse Konsequenzen hat.

R Eine Meta-Regel verwendet reguläre Ausdrücke. Unter Verwendung von regulären Ausdrücken (*regexp(6)*) kann der Name des Ziels in Fragmente zerlegt werden, die im Rezept mit *$stem*i ($1 \le i \le 9$) verwendet werden können.

Q Das Rezept wird vor der Ausführung nicht ausgedruckt.

P Die zwischen dem »P« und »:« stehenden Zeichen werden als Kommando interpretiert und durch *rc -c prog 'targ_i' 'prereq_j'* für alle Kombinationsmöglichkeiten von *i* und *j* aktiviert. Terminiert das Kommando mit einem Fehlerstatus, wird das zugehörige Rezept ausgeführt. Im nachfolgenden Beispiel wird dieses Attribut ausgenutzt.

Unter Unix wie unter Plan 9 wird ein Compiler bzw. Interpreter i.a.R. unter Verwendung von *yacc* und *lex* implementiert. Ein Lauf von *yacc* kann eine C-Quelle und eine Definitionsdatei *y.tab.h* erzeugen. Letztere enthält die symbolischen Namen der Eingabesymbole und ändert sich nur, falls man neue Eingabesymbole in die Grammatik einfügt. Um unnötige Übersetzungen zu sparen, unterhält man ein Kopie von *y.tab.h*, die man nur bei Bedarf ändert. Betrachten wir ein *mkfile*, das dieses Problem effizient löst.

```
# mk_P_yacc: ein Beispiel fuer die Anwendung des P-Attributs

< /$objtype/mkfile
```

```
lex.$O: x.tab.h
        $CC lex.c

x.tab.h:QP:cmp —s:      y.tab.h     # Psst
    cp y.tab.h x.tab.h
y.tab.c y.tab.h:    gram.y
    $YACC —d gram.y
```

Nur wenn in *gram.y* neue *token* eingeführt worden sind, wird *x.tab.h* aktualisiert und demzufolge *lex.$O* erzeugt. Der Vergleich von *x.tab.h* mit der neu erstellten *y.tab.h* löst nur dann einen Kopiervorgang aus, falls sich die Dateien unterscheiden.

N Falls kein Rezept vorhanden ist, wird das Ziel mit einem neuen Zeitstempel versehen.

< Die Standardausgabe des Rezepts wird von *mk* wie ein zusätzliches *mkfile* behandelt. Somit kann man den Ablauf zur Laufzeit durch abgearbeitete Rezepte beeinflussen.

U Das Ziel bekommt auch dann einen neuen Zeitstempel, wenn das Rezept dies nicht erledigt.

In unserer Installation sind die Attribute N, U, und < wirkungslos.

7.7 Ausführung

Die Bewertung und Ausführung eines *mkfiles* findet in folgenden Schritten statt:

- Zuerst werden alle Dateien für < ersetzt.
- Zeilenfortsetzungen, ausgelöst durch \, werden aufgelöst.
- Dann werden Leerzeilen und Kommentare (von # bis zum Zeilenende) gelöscht.
- Außerhalb von *recipe*s wird *'{ cmd }* durch die Ausgabe von *cmd* ersetzt. Innerhalb von *recipes* erfolgt dieser Ersatz später bei der Ausführung durch *rc*.
- Variablen werden ersetzt, Sonderzeichen können mit Apostroph zitiert werden.
- Regeln werden ersetzt bzw. erweitert.
- Wenn *target* und *prereq* gleich sind, ersetzt das zweite *recipe* das erste.
- Wenn nur die Ziele gleich sind und nur eine Regel ein Rezept hat, werden die *prereq* verkettet.
- Wenn auch nur ein Ziel das V-Attribut hat, wird das Rezept immer ausgeführt.

Ohne ein Ziel als Teil des Arguments von *mk* wird das erste Ziel im *mkfile* bearbeitet.

7.8 Bibliotheken und *mk*

Bibliotheken enthalten in aller Regel übersetzte C-Module. Ist ein Modul aktualisiert worden, sollte die verbesserte Version das alte Modul in der Bibliothek ersetzen. Um die Aktion anzustoßen, reicht es nicht aus zu fragen, ob das übersetzte Modul

neuer als die Bibliothek ist, sondern man muß die Frage beantworten können, ob das übersetzte Modul neuer als das entsprechende Modul in der Bibliothek ist. Der Zeitstempel eines Moduls aus einer Bibliothek kann via *libname(modulname)* erfragt werden. Mit dem nachfolgenden *mkfile* kann eine Bibliothek aktualisiert werden.

```
# mk_4: maintain a library:
# see also membername(1)

< /$objtype/mkfile

LIB=/$objtype/lib/libc.a

$LIB:V: $LIB(abs.$O) $LIB(access.$O) # ...
    echo $newprereq
    names=`{membername $newprereq}
    ar r $LIB $names && $RL $LIB && rm $names

</sys/src/cmd/mkone
```

Ein Ablauf erklärt das *mkfile* selbst:

```
% mk -f mk_4
names=`{membername $newprereq}
ar r libc.a abs.k access.k && rl libc.a && rm abs.k access.k
```

membername muß aus einer Angabe wie *libc.a((access.k)* den Namen *access.k* extrahieren. In Plan 9 ist *membername* als *rc*-Skript realisiert:

```
#!/bin/rc
echo '$*'              |
tr ' ' '\012'              | \
sed –e 's/[^(]*\(([^)]*)\).*/\1/' | \
tr '\012' ' '
```

7.9 Drucken und *mk*

Oftmals besteht ein Projekt aus einer wachsenden Anzahl von Programmen *cmd*, deren Quellen in *cmd.[hc]* abgelegt sind. Das Problem besteht nun darin, daß man i.a.R. nur die seit dem letzten Ausdruck veränderten Quellen drucken möchte. Das nachfolgende *mkfile* löst das Problem.

```
# mk_5:

< /$objtype/mkfile

TARG=      alloc       arc       bquote
HEADER=lib/alloc.h lib/arc.h
SOURCE=lib/alloc.c lib/arc.c

print:Q: $HEADER $SOURCE
    echo 'to print: ' $newprereq
    touch $newprereq
```

```
% touch Src/*
% mk -f mk_5
to print:  Src/alloc.h Src/arc.h Src/alloc.c Src/arc.c
% mk -f mk_5
mk: 'print' is up to date
```

```
% touch Src/alloc.h
% mk -f mk_5
to print:  Src/alloc.h
```

Es ist klar, daß dieses *mkfile* nicht besonders pflegefreundlich ist. Besser wäre es, wenn man statt dessen nur TARG pflegen müßte. In *mk* kann dieses Problem mittels Textersatz gelöst werden. Die Syntax lautet:

${dummy:A%B=C%D}

wobei *A, B, C* und *D* Zeichenketten sind, die auch leer sein können.

In diesem Fall wird die Variable *dummy* in drei Teile zerlegt: die Textstücke mit den Inhalten von *A* und *B*, hinter dem »%« verbirgt sich der Rest des Texts der Zeichenkette. Nach dem Gleichheitszeichen wird eine Zeichenkette, bestehend aus den Textstücken *C, D* und »%«, zusammengesetzt, diese kann wieder an eine Variable zugewiesen werden. Im nachfolgenden Beispiel wurde diese Technik verwendet, um *mk_5* pflegeleichter zu implementieren.

```
# mk_6: Textersatz, Verbesserung von mk_5

TARG=  alloc   arc   bquote
HEADER=${TARG:%=Src/%.h}
SOURCE=${TARG:%=Src/%.c}

print:Q: $HEADER $SOURCE
    echo 'to print: ' 'lib/' ^ $newprereq
    touch print
```

7.10 Zusammenfassung

Mit Plan 9s *mk* kann man elegant und adäquat alle vorkommenden Sachverhalte beschreiben die Aktionen aufgrund von älter/neuer-Relationen hervorrufen sollen. Die Programmierung der Meta-Regeln, mit denen ganze Problemkreise auf einmal bearbeitet werden können, ist unproblematisch, weil die Regeln in derselben Sprache definiert werden, in der man ein *mkfile* schreibt.

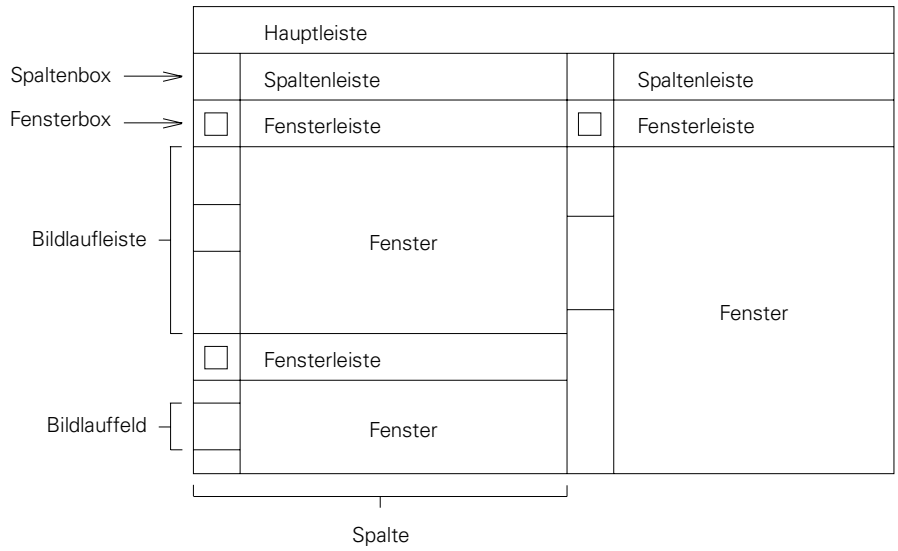
Die Entwicklung von *mkfiles* für die Übersetzung von komplexen Systemen für verschiedene Architekturen ist genauso leicht oder schwierig, wie wenn das *mkfile* nur für die Übersetzung für eine Architektur gedacht ist. Gesteuert wird der Aufruf der benötigten Programme durch nur eine Variable und die Anwendung von systematisch verteilten include-Anweisungen. Dies überrascht, wenn man weiß, wie aufwendig das gleiche Problem durch *imake* gelöst wird.

Durch Export von sinnvoll benannten Variablen und durch Übergabe aller Kommandos auf einmal harmoniert *mk* wesentlich besser mit der Plan 9-Shell *rc* als *make* mit der jweils verwendeten Unix-Shell. Dies macht die Programmierung der Regeln doch um vieles eleganter und bequemer.

Kurz gesagt, *mk* ist an sich schon ein toller Wurf.

8
acme

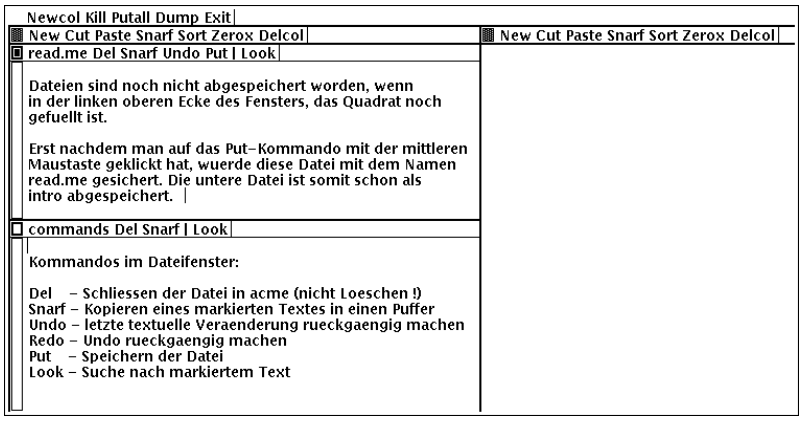
Acme ist ein interaktiver Texteditor, der mit Hilfe der Maus- und Tastatursteuerung in einer Vielzahl von Fenstern bedient wird. Nach einer kurzen Lernphase stellt sich *acme* als effizientes und leicht verwendbares Werkzeug heraus. Es dient sowohl zur Erstellung von Quellcodes und beliebigen Texten als auch zur Ausführung von Programmen in einer Shell und zum Browsen durch einzelne Verzeichnisse. Weiterhin werden *acid* und Teile von *sam* unterstützt. *Acme* ist also ideal geeignet für die Erstellung, Erprobung und Untersuchung von Programmen.



Als erstes ruft man den Editor wie gewohnt mit *acme datei* auf. Die obige Abbildung ist eine schematische Darstellung, in der die einzelnen Objekte von *acme* mit den Namen gekennzeichnet sind, die im weiteren verwendet werden. In der Hauptleiste befinden sich verschiedene generelle Kommandos zum Verlassen des Programms oder zum Abspeichern aller editierten Dateien. Darunter teilt sich die Umgebung anfangs meist in zwei Spalten auf. Jede dieser Spalten enthält wiederum eine Leiste mit Kommandos, die Spaltenleiste. Erst im unteren Teil befinden sich die einzelnen Fenster der Dateien, welche jeweils eine Fensterleiste mit speziellen Kommandos nur für die entsprechenden Dateien enthalten. Bei *acme* liegt Tiling vor, d.h., Fenster können sich nicht gegenseitig überlappen. Jedes Fenster ist auf seinen eigenen Bereich mit den zugehörigen Leisten eingeschränkt.

Der folgende Screendump zeigt die Ansicht des Editors nach dem Aufruf von *acme read.me commands*. In der linken Spalte sind die beiden Dateien, während die rechte Spalte noch leer ist. Ein Aufruf ohne Dateinamen oder nur mit Verzeichnisnamen bewirkt, daß in den Spalten nur Fenster mit den Dateinamen und den Na-

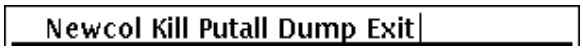
men der Unterverzeichnisse abgebildet werden. Der Inhalt des Verzeichnisses *directory* läßt sich deshalb mit *acme directory* öffnen.



Die Kontextsensitivität der Maus in *acme* ist etwas gewöhnungsbedürftig. Nur durch ein einfaches Umherfahren der Maus ohne ein Anklicken wird ein Fenster oder auch eine Kommandoleiste ausgewählt. Die Auswahl wird allerdings nicht angezeigt, so daß allein die Position der Maus ausschlaggebend ist. Die Texteingabe erscheint in demselben Fenster, in dem sich auch gerade der Mauszeiger befindet, und zwar unter dem entsprechenden Tastaturcursor. Durch ein Anklicken mit der linken Maustaste bewegt sich der Tastaturcursor an eine andere Stelle. Dies gilt insbesondere auch für den Cursor bei den Kommandos und Dateinamen in den Kommandoleisten. Auf diese Weise lassen sich neue Kommandos kurzzeitig eintragen oder die Dateinamen verändern. Eine Tabelle zur Verwendung der Maustasten befindet sich am Ende dieses Kapitels.

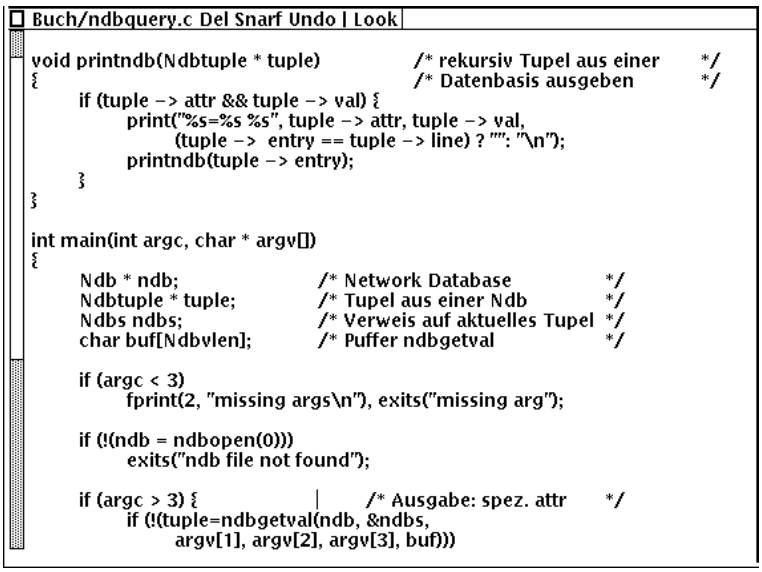
8.1 Kommandoleisten

Bei den Kommandoleisten handelt es sich um die Hauptleisten, die Spaltenleisten und um die Fensterleisten. Alle Leisten enthalten einzelne Kommandos, die Fensterleisten verfügen zudem noch über die Datei- oder Verzeichnisnamen der Fenster. Die Kommandos führt man mit der mittleren Maustaste aus. Wird zum Beispiel das Kommando *Newcol* mit der mittleren Maustaste ausgelöst, erscheint am rechten Rand eine neue Spalte. Diese Spalte enthält bereits ein neues, namenloses Dateifenster. Mit *Delcol* kann man die Spalte wieder löschen. *Putall* speichert gleichzeitig die gesamten aktuellen Dateien in allen Fenstern. Zum Schluß beendet man *acme* mit *exit*. Eine komplette Liste der *acme*-Kommandos ist ebenfalls am Kapitelende.



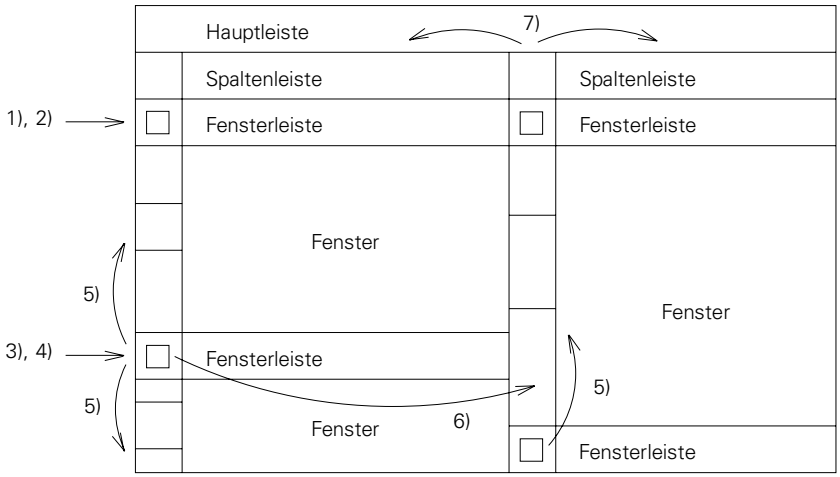
8.2 Fenster

Auf der linken Seite eines Fensters liegt die Bildlaufleiste, im oberen Teil die Fensterleiste und links davon eine kleine Fensterbox. Der Text läßt sich wie üblich mit der Tastatur eingeben, nachdem man mit dem Kommando *new* ein neues Fenster erstellt hat. Das weiße Bildlauffeld in der Bildlaufleiste kann mit Hilfe der gedrückt gehaltenen mittleren Maustaste auf dem dunklen Hintergrund hoch und runter bewegt werden. Der Text im Fenster bewegt sich analog. Durch ein Drücken auf die linke bzw. rechte Maustaste fährt der Text umso schneller in größeren Schritten nach oben bzw. nach unten, je tiefer sich der Mauszeiger auf dem Bildlauffeld befindet. Hält man die Maustaste gedrückt, wandert der Text von selbst mit hoher oder niedriger Geschwindigkeit in die gewählte Richtung.



Falls die Fensterbox auf der linken oberen Seite eines Fensters schwarz ist, deutet dies darauf hin, daß der Text geändert, aber noch nicht abgespeichert wurde. Erst wenn die Box weiß ist, ist der Text bereits gesichert, oder er wurde noch nicht editiert. Rechts neben der Fensterbox steht der Dateiname. Ein neuer Name wird eingegeben, indem man den Mauszeiger über die Fensterleiste hält, den Tastaturcursor auf den Dateinamen positioniert und den gewünschten Namen einfügt. Die Datei läßt sich dann mit dem Kommando *Put* speichern, indem man die mittlere Maustaste über dem Wort *Put* aus der Fensterleiste drückt. Der Kommandoname kann auch von einem selbst eingegeben werden, z.B. in einem Fenster oder am rechten Ende einer Kommandoleiste. Wie im weiteren noch gezeigt wird, löst das Drücken der linken, rechten oder mittleren Maustaste über demselben Wort unterschiedliche Ereignisse aus. Die mittlere Taste interpretiert das Wort als Kommandonamen, mit der rechten können Dateien geöffnet werden, und die linke dient zum Cut & Paste.

Mit *Del* schließt man das Fenster, löscht damit aber glücklicherweise nicht die Datei, wie der Kommandoname schon anzudeuten scheint. Falls man die Datei nicht speichern kann oder *Del* ausgeführt wurde, noch bevor man die Datei abgespeichert hat, erscheint in der rechten Spalte ein zusätzliches Fenster mit der entsprechenden Fehlermeldung. Diese Fenster tauchen bei allen auftretenden Fehlern und Warnungen auf und lassen sich wiederum mit *del* schließen.



Wenn sich mehr als ein Fenster in einer Spalte befinden, möchte man oftmals ein bestimmtes Fenster vergrößern oder sogar die gesamte Fläche einer Spalte einnehmen lassen. Um dies zu erreichen, werden wiederum die Boxen benötigt. Die Nummern in der schematischen Abbildung zeigen die nachfolgend beschriebenen Anwendungsmöglichkeiten der einzelnen Boxen.

1) Durch einen Mausklick mit der rechten Maustaste auf die Fensterbox eines Fensters lassen sich alle anderen Fenster unsichtbar in den Hintergrund bringen.

2) Durch einen anschließenden Mausklick mit der mittleren oder linken Maustaste auf die Fensterbox lassen sich wieder die Kommandoleisten der anderen Fenster im unteren Teil der Spalte in den Vordergrund heben.

3) Vom unteren Teil aus kann man ein Fenster langsam durch ein Drücken der linken Taste nach oben ziehen.

4) Mit der mittleren Taste läßt sich das Fenster wieder direkt unter die Kommandoleiste des anderen Fensters setzen.

5) Alternativ dazu kann man ein Fenster auch durch eine gedrückt gehaltene beliebige Maustaste hoch und runter bewegen.

6) Weiterhin ist es dadurch möglich, ein Fenster auch komplett in eine andere Spalte zu befördern, indem man mit der gedrückten Maustaste einfach in die nächste Spalte geht.

7) Die Spalten lassen sich mit der Spaltenbox nach links oder rechts bewegen.

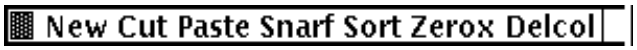
8.3 Spalten

Mit der Spaltenbox können sogar ganze Spalten, die mehrere Fenster enthalten, ausgetauscht und verschoben werden. Gleichgültig, mit welcher Maustaste man dies unternimmt, eine Spalte läßt sich direkt durch eine gedrückt gehaltene Maustaste von links nach rechts und umgekehrt vergrößern oder verkleinern. Führt die Vergrößerung über eine andere Spalte hinweg, so werden die beiden Spalten ausgetauscht. Ein Manko ist, daß man die erste linke Spalte nicht mit einer der anderen Spalten austauschen kann. Nur die Spalten auf der rechten Seite können verschoben und von der Position her verändert werden.

8.4 Cut & Paste

Der Text in einem Fenster läßt sich sowohl mit der Tastatur eingeben als auch mit der Maus bearbeiten. Leider werden in Plan 9 die Cursortasten nicht wie in anderen Betriebssystemen unterstützt. Folglich kann man den Cursor nur durch ein Anklicken mit der Maus positionieren.

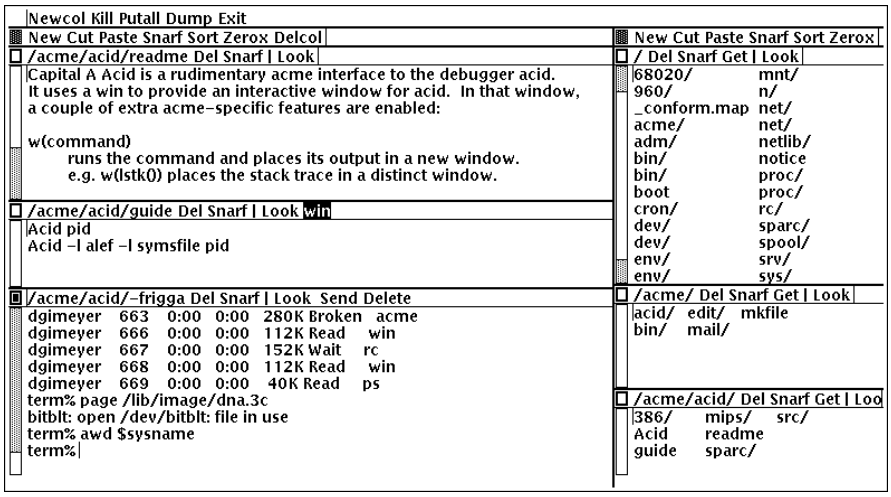
Mit der rechten Maustaste läßt sich ein Wort anklicken oder ein längerer Text markieren. Nachdem die Taste losgelassen wurde, erscheint der nächste mit dem markierten Text übereinstimmende Text. Da generell ein Text mit der linken Maustaste markiert werden kann, ist es ebenso möglich, erst den Text zu markieren und dann mit der rechten Maustaste die Suche zu beginnen. Falls das Wort nicht im Text vorhanden ist, kann das zu suchende Wort am besten im rechten Teil der Kommandoleiste des Fensters eingetragen werden, um von dort aus die Suche zu starten. Somit steht das Suchwort nicht hinderlich im Text selbst.



Um einen Textabschnitt zu kopieren oder auszuschneiden, können die Kommandos *Cut*, *Paste* und *Snarf* aus der Kommandoleiste der Spalten verwendet werden. Erst wird, wie bereits erwähnt, mit der linken Maustaste ein Teil markiert, dann mit der mittleren Maustaste das Kommando *Snarf* zum Kopieren oder *Cut* zum Ausschneiden ausgeführt. Schließlich kann der markierte Text an einer neuen Cursorposition in demselben oder in einem anderen Fenster mit *Paste* wieder eingefügt werden.

8.5 Dateibrowser und *rc*-Shell

Außer der Bearbeitung eines Texts kann *acme* auch als eine Art Dateibrowser verwendet werden. Startet man *acme* z.B. mit dem Wurzelverzeichnis, so werden alle Dateien und Verzeichnisse des Wurzelverzeichnisses in einem Fenster angezeigt. Ein Klick mit der rechten Maustaste auf eines der Verzeichnisse öffnet erneut ein Fenster mit allen dort vorhandenen Dateien. Wählt man eine Datei mit der rechten Maustaste, so wird sie in einem Fenster angezeigt und kann bearbeitet werden. Mit dieser Methode lassen sich einzelne Dateien in verschiedenen Verzeichnissen ansteuern.



Neben den von *acme* bereits vorgegebenen Kommandos können mit der mittleren Maustaste auch Plan 9-Befehle ausgeführt werden. Klickt man z.B. auf das Wort *ls*, erscheint das Resultat, die Dateiliste des augenblicklichen Verzeichnisses, in einem neuen Fenster. Im obigen Screendump befindet sich im zweiten linken Fenster das bereits dort eingegebene Wort *win*. Mit der mittleren Maustaste wurde das Wort als Kommando ausgeführt und öffnete auf diese Weise das dritte linke Fenster mit einer *rc*-Shell, die ein paar zusätzliche *acme*-spezifische Kommandos enthält.

Eigentlich ist *win* nur ein Vermittler zwischen *acme* und einer Shell, der die Standardeingabe und die Standardausgabe der Shell mit dem Text eines Fensters verbindet. Die Standardausgabe wird zur Ausgabe des Fensters weitergeleitet. Umgekehrt erhält die Shell erst dann die Standardeingabe, wenn eine Zeile im Fenster durch ein *Newline* abschlossen wird. Der Nachteil einer solchen Shell in *acme* ist jedoch, daß keine Programme gestartet werden können, die Graphiken benutzen, wie z.B. *acme* selbst oder *page*. Mit dem Kommando *fb/9v* lassen sich hingegen Bilder in einem separaten Fenster außerhalb von *acme* anzeigen.

Ein *win*-Shellfenster erhält einen Namen, bestehend aus dem aktuellen Verzeichnis und einem Label, welches standardmäßig *-rc* ist. Der Name kann mit dem Befehl *awd label* in der mit win erzeugten Shell neu eingestellt werden. So wird in der Abbildung der Systemname *$sysname=frigga* als neuer Labelname eingefügt. Es ist praktisch, den Verzeichnisnamen zusammen mit dem Systemnamen in der Fensterleiste ständig über *cd* zu aktualisieren.

```
fn cd {builtin cd $1 && awd $sysname}
```

Mit der rechten Maustaste kann eine Include-Datei automatisch geöffnet werden. In einem *alef*- oder C-Quelltext klickt man auf die entsprechende Anweisung *#include <header.h>*, und es erscheint ein Fenster mit der Datei.

Falls der markierte Text das Format *:n* besitzt, wird die *n*-te Zeile im Text angezeigt und markiert. Das Format *file:n* öffnet somit ein Fenster mit der Datei *file* an der *n*-ten Zeile. Mit *<header.h:42>* ist es beispielsweise möglich, die 42. Zeile in

der Include-Datei *header.h* zu zeigen. Fehlermeldungen, die ein Compiler für den Quellcode eines Programms liefert, besitzen immer das Format *file:n*. Führt man den Compiler also in einer Shell in *acme* aus, kann durch ein Anklicken auf diesen Fehler sofort die Datei mit der markierten Fehlerzeile geöffnet werden.

Möchte man mehrere Zeilen auf einmal in einer Datei markieren, verwendet man einfach das Format *file:n,m* für die Zeilen *n* bis *m*. Wer noch genauer vorgehen will, kann auch die einzelnen Zeichen von *x* bis *y* mit *file:#x,#y* markieren lassen.

8.6 *acme*-spezifische Kommandos

Im Verzeichnis */acme* befinden sich Unterverzeichnisse für zusätzliche *acme*-spezifische Kommandos, die nur in *acme* anzuwenden sind, z.B. in einer mit *win* erstellten Shell. In */acme/bin* liegen u.a. *win* selbst, aber auch Kommandos wie *aspell*. *Aspell* verhält sich genau wie *spell*, indem es den Text einer Datei auf seine syntaktische Korrektheit der englischen Sprache überprüft. Befindet sich in einem Text ein fehlerhaftes Wort, wird von *aspell* seine genaue Position innerhalb des Textes, zum Beispiel mit *file:#235,#247:jurisdiktion*, angegeben. Durch Anklicken der Fehlermeldung mit der rechten Maustaste läßt sich das Wort sofort im Text anzeigen.

In den anderen Unterverzeichnissen befinden sich zusätzliche Kommandos. Es ist am einfachsten, zuerst die *readme*-Datei durchzulesen und anschließend ein Beispielkommando in *guide* nach seinen Ansprüchen zu verändern und aufzurufen.

Im Gegensatz zu *sam* werden in *acme* Texte nur manuell verändert. Es gibt keine Kommandobefehle, die z.B. einen Textersatz an bestimmten Stellen automatisch durchführen. Das Verzeichnis */acme/edit* enthält jedoch in etwa die gleichen Kommandos, wie sie auch in dem *sam*-Editor angeboten werden. Unterschiede gibt es in der Syntax, da es sich um Shell-Kommandos handelt, die über eine Pipe ausgeführt werden. So kann in der Datei *file* das fehlerhafte Wort *jurisdiktion* durch *jurisdiction* mit *e 'file:0,$' | x 'jurisdiktion' | c 'jurisdiction'* gesucht und ersetzt werden. Die Buchstaben *e*, *x* und *c* sind Kommandos aus */acme/edit/sparc, mips, 386*. Mit *e* wird eine bestimmte Datei ausgewählt, in der mit *x* und *c* der Text von der ersten bis zur letzten Zeile verändert wird.

Genauso gibt es auch für die E-Mails ein *mail*-Kommando in */acme/mail* (siehe auch 5.6). Wenn *Mail* (nicht wie gewöhnlich *mail*) mit der mittleren Maustaste ausgelöst wird, entsteht ein neues Fenster, in dem alle Mails aufgelistet werden. Die Maustasten haben nun eine andere Bedeutung. Mit der rechten Maustaste läßt sich eine Mail direkt in einem weiteren Fenster darstellen. Auch die Kommandozeilen sind nun neu. Falls einer Mail eine Antwort zugeschickt werden soll, braucht nur *Reply* in der Kommandozeile mit der mittleren Maustaste gestartet werden. Die Liste der Mails wird von *acme* automatisch ständig aktualisiert, so daß der Editor auch allein als Mail-Box zu verwenden ist.

Der *acid*-Debugger, der im folgenden Kapitel erläutert wird, kann mit *Acid* um die Funktion *w(command)* erweitert werden. Ein *acid*-Kommando wie *lstk()* legt dadurch seine Ausgabe auf ein neues *acme*-Fenster ab, sofern der Debugger auch unter *acme* gestartet wurde.

8.7 Verwendung der Maustasten

	linke Maustaste	*mittlere Maustaste*	*rechte Maustaste*
Kommandos	Cursor positionieren	Kommando ausführen	
Spaltenbox	Spalten verschieben		
Fensterbox	Fenster langsam vergrößern	Fenster maximieren (andere Fenster außer den Fensterleisten in den Hintergrund bringen)	Fenster maximieren (andere Fenster in den Hintergrund bringen)
Bildlaufleiste	Bildlauffeld nach oben bewegen	Bildlauffeld mit der Maus bewegen	Bildlauffeld nach unten bewegen
Fenster	Cursor positionieren und Text mit gedrückter Taste markieren	markierten Text als Kommando ausführen	markierten Text suchen oder eine Datei mit dem Dateinamen öffnen

8.8 Kommandos

Cut Der zuletzt markierte Text wird gelöscht und im Puffer abgelegt.

Del Fenster schließen. Falls im Fenster der Text nicht abgespeichert wurde, kann es mit einem zweiten *Del* ohne Speicherung geschlossen werden.

Delcol Spalte mit allen Fenstern schließen.

Delete Fenster schließen, gleichgültig, ob der Text gespeichert wurde.

Dump Status von *acme* abspeichern, standardmäßig in *acme.dump*. Mit *acme -l acme.dump* kann später *acme* mit den gleichen Spalten, Fenstern und Fonts aufgerufen werden.

Exit Programm verlassen.

Font Ohne Argumente wird zwischen Proportionalschrift und einer Schrift mit gleichem Buchstabenabstand innerhalb des entsprechenden Fensters gewechselt. Falls eine Datei als Argument übergeben wird, wird die Datei als Schriftart für das Fenster eingesetzt.

Get Datei laden. Ohne Argumente wird die aktuelle Datei erneut geladen.

ID ID-Nummer des Fensters ausgeben.

Incl Nach einem Mausklick mit der rechten Maustaste auf eine Include-Datei in <> beginnt *acme* seine Suche in den Verzeichnissen */bjtype/include*, */sys/include* und bei *alef*-Programmen in */sys/include/alef*. *Incl* fügt neue Verzeichnisse zu dieser Liste hinzu. Ohne Argumente wird die aktuelle Liste von Verzeichnissen in einem neuen Fenster ausgegeben.

Kill Schickt ein *kill*-Kommando an alle als Argumente angegebene *acme*-Kommandos.

Local Startet ein Kommando im gleichen Namensraum und mit den gleichen Environment-Variablen. So kann mit *local bind* sogar über *acme* ein Verzeichnis eingebunden werden.

Load Status von *acme* mit Hilfe einer Datei wiederherstellen (siehe Dump).

Look Suche nach dem als Argument übergebenen Text oder nach dem bereits markierten Text.

New Neues Fenster erstellen.

Newcol Neue Spalte erstellen.

Paste Letzten markierten Text durch den Text aus dem Puffer ersetzen.

Put Text in einer Datei abspeichern. Ohne Argument wird der Text in derselben Datei wie vorher abgespeichert.

Putall Texte aus allen Fenstern abspeichern.

Redo Umkehrung von *Undo*.

Send Markierten Text oder gepufferten Text am Ende des Textes im Fenster anhängen.

Snarf Markierten Text im Puffer ablegen.

Sort Fenster in einer Spalte lexikographisch nach ihren Namen ordnen.

Undo Letzte Textänderung rückgängig machen.

Zerox Eine Kopie des aktuellen Texts in einem neuen Fenster erstellen. Beide Fenster enthalten exakt den gleichen Text. Eine Änderung in einem Fenster ergibt automatisch dieselbe Änderung im zweiten Fenster.

8.9 Zusammenfassung

Acme ist nicht allein ein Texteditor. Kommandos wie *win* und *Acid* und die direkte Anzeige von Verzeichnissen und *Include*-Dateien unterstützen die Implementierung, die Tests und das Debugging von Programmen. Darüber hinaus ist *acme* mit den vielfältigen Mausfunktionen und *acme*-spezifischen Kommandos wie *Mail* oder *aspell* sehr umfangreich, obwohl es kein Popup-Menü gibt. Die meisten denkbaren Operationen können außerdem durch einen einfachen Mausklick ausgeführt werden. Die Nachteile liegen vor allem in den *sam*-ähnlichen Befehlen wie *x* und *c*, die eine etwas umständlichere Syntax besitzen. Außerdem muß man sich zu Beginn mit der Vieltätigkeit der Mausanwendungen vertraut machen. Trotzdem bleibt *acme* insbesondere für die Programmierung der für Plan 9 zu bevorzugende Editor.

9
acid

Fast jedes Programm, welches man schreibt, enthält anfangs Fehler. Manchmal tauchen während des Programmablaufs Fehler auf, die auf den ersten Blick nicht sofort zu erkennen sind. Deshalb startet man einen Debugger, um z.B. nach Speicherlecks zu suchen. Für die Suche stellt jeder Debugger vorgegebene Kommandos zur Verfügung, mit denen sich Programmabläufe analysieren lassen. So ist ein solcher Debugger nicht von Nutzen, wenn man spezielle Untersuchungen mit den eigenen Kommandos an seinem Programm durchführen will. Wenn das Programm dann auch noch in einer Sprache geschrieben wurde, die vom Debugger nicht unterstützt wird, kann das Programm nicht untersucht werden. Zur Lösung dieser und anderer Probleme dient der Debugger *acid*, da er folgende besondere Eigenschaften besitzt:

- Programmiersprachenunabhängigkeit: Programme in *alef*, *C* und Assembler werden bisher unterstützt, aber es ist möglich, jede andere Sprache mit einzubinden.
- Programmierbarkeit: *acid* stellt Funktionen zur eigenen Programmierung bereit.

Acid scheint auf den ersten Blick relativ schwer zu bedienen, da keine grafische Benutzeroberfläche und nicht sehr viele eigene Kommandos unterstützt werden. Der Vorteil liegt dementsprechend mehr darin, daß *acid* sich selbst programmieren läßt und somit viele unterschiedliche Probleme analysiert werden können.

Acid hat ein paar eingebaute Funktionen, mit denen man eigene neue Debugger-Kommandos bauen kann. Einige dieser mit den *acid*-Funktionen entwickelten Kommandos sind bereits vorgegeben, so daß man z.B. nicht mehr das berühmte *next()*-Kommando zum schrittweisen Durchlaufen eines Programms selbst schreiben muß. Andere denkbare Kommandos wie eine Überprüfung der Speicherverwaltung mit den ANSI C-Funktionen *malloc()* und *free()* sind jedoch nur für MIPS-Rechner bisher realisiert worden.

Im folgenden wird zunächst ein einfaches C-Programm auf Fehler überprüft, und danach werden beispielhaft zwei neue Debugger-Kommandos gebaut und einige Beispiele gezeigt.

9.1 Debugging

Das Programm *search* soll die Position eines bestimmten Worts in einem Text herausfinden. Die Suche nach dem Wort *"Suche"* sollte also als Resultat die Position 2 im Text *"Die Suche nach einem Wort »\n« ist eigentlich sehr einfach."* ergeben.

```
#include <u.h>
#include <libc.h>
#include <stdio.h>
```

```
int search(char * target, char * words[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (strcmp(target, words[i]))
            return i;
    return –1;
}

void main(int argc, char * argv[])
{
    int i, n = 0;

    char * words[1024];

    if (argc < 2)
        fprint(2, "usage: %s <word>\n", argv[0]),
        exits("missing word to search for");

    while (scanf("%s", words[n++]) > 0) ;

    if ((i = search(argv[1], words, n)) >= 0)
        print("found %s as word #%d\n", argv[1], i+1);

    exits("");
}
```

Normalerweise wird der Text mit *scanf()* von der Standardeingabe Wort für Wort gelesen, die Suche begonnen und schließlich die Positionsnummer des Worts ausgegeben. Falls kein Suchwort als Argument angegeben ist, wird ein Fehler ausgegeben. Soweit ist der Ansatz. Tatsächlich ergibt sich jedoch beim Ablauf:

```
%term search
usage: search <word>
%term search Suche
Die Suche nach einem Wort
search 243: suicide: sys: trap: fault write
       addr=0x0 pc=0x0000202a
```

Nach dem Einlesen einer Zeile bricht das Programm sofort ab. Die Fehlermeldung gibt an, es würde versucht, bei der Adresse *addr=0x0* zu schreiben. Der Fehler sei dabei im Programm an der Stelle *pc=0x0000202a* aufgetreten.

Die Strategie soll sein, mit Hilfe von *acid* zuerst die Fehlerausgabe beim Programmabbruch zu analysieren. Danach soll die Suche Schritt für Schritt erfolgen und bestimmte Variablen untersucht werden. Immer wenn ein Fehler gefunden wird, soll er direkt behoben werden. Zum Schluß sollten zwei entscheidende Fehler entfernt worden sein.

Um herauszufinden, wo sich die Abbruchstelle im Quelltext befindet, startet man *acid*.

```
%term acid search
search:sparc plan 9 executable

/sys/lib/acid/port
/sys/lib/acid/sparc
```

```
Symbol renames:
    _exits=$_exits T/0x3fc0
acid: src(0x202a)
/sys/src/libstdio/vfscanf.c:296
 291            if(c==EOF){
 292                if(nn==0) return 0;
 293                else goto Done;
 294            }
 295            nn++;
>296            if(store) *s++=c;
 297            wgetc(c, f, Done);
 298        }
 299        nungetc(c, f);
 300    Done:
 301        if(store) *s='\0';
```

Acid wird mit dem lauffähigen Programm *search* aufgerufen und gibt als erstes die Dateien aus, welche standardmäßig geladen werden, um neben den *acid*-Funktionen weitere Debugger-Kommandos anzubieten. Auf diese Weise wird u.a. das Kommando *src()* in */sys/lib/acid/port* definiert. Mit Hilfe dieses Kommandos können nun direkt zehn der Quelltextzeilen ausgegeben werden, bei denen es zum Abbruch des Programms kam. Der Benutzer kann immer in den Zeilen, beginnend mit *acid:*, eine Funktion eingeben und mit *Return* ausführen. Die Adresse *0x202a* gibt die Lage des Abbruchs im Quelltext an, so daß *src()* mit dem Zeichen »>« die Stelle anzeigt. Die Pfade der Quelltexte findet *acid* automatisch heraus. Die Position der Datei *search.c* ist z.B. fest am Ende von *search* eingebrannt. Folglich darf natürlich *search.c* nicht gelöscht oder woandershin verlegt werden, falls das Programm mit *acid* bzgl. dieses Quelltexts untersucht werden soll. Da das Programm in *vfscanf.c* in der Zeile 296 abstürzt, liegt der Fehler vermutlich in der *do-while*-Schleife von *search.c*, wo *scanf()* aufgerufen wird, in dem wiederum vfscanf() aufgerufen wird. Im folgenden soll nach und nach die genaue Ursache mit Hilfe der Funktionen und Kommandos von *acid* ermittelt werden.

Als erstes wird ein Prozeß für das Programm mit *new()* gestartet unter Angabe des Worts *Suche* als Programmargument. Das Argument *"Suche"* wird der globalen Variablen *progargs* zugewiesen, welche durch die doppelten Anführungszeichen direkt als String-Format spezifiziert wird. Mit *new()* wird gleichzeitig ein Breakpoint auf die erste Programmzeile gesetzt, damit das Programm beim Ablauf gleich in der ersten Zeile gestoppt wird. Um es weiter ablaufen zu lassen, wird *cont()* aufgerufen. Eine Zeile des Textes wird eingegeben, und danach bleibt das Programm erneut an der gleichen Stelle stehen.

```
acid: progargs="Suche"
acid: new()
216: system call    _main    SUBL    $0x8,SP
216: breakpoint    main+0x6    ANDL    $0x0,n+0x100c(SP)
acid: cont()
Die Suche nach einem Wort
216: page fault    icvt_s+0xc6    MOVB    DL,0x0(AX)
Notes pending:
    sys: trap: fault write addr=0x0
acid: src(*PC)
```

Hierbei liefert *acid* die gleiche Ausgabe wie oben. Der Programmzähler, der die Stelle anzeigt, an der das Programm anhielt, befindet sich in einem Register. Auf dieses Register verweist die Variable *PC* mit einer Adresse. Deshalb zeigt *PC* nicht den eigentlichen Wert des Registers an, sondern nur die Adresse des Registers. Um den Wert, auf den eine Adresse zeigt, zu erhalten, kann in *acid* der Stern »*« verwendet werden.

Die Adresse des Registers befindet sich also bei *PC=0xc0000fec*, wo der Wert **PC=0x0000202a* steht, der auch schon beim normalen Programmaufruf angegeben wurde. Der Quelltext kann wieder mit *src(*PC)* den Fehler anzeigen.

Das Format des Programmzählers *PC* bei der Ausgabe ist eigentlich das hexadezimale Adreßformat beginnend mit »0x«. Dies kann allerdings für die Ausgabe durch das Anhängen von »\« und einem speziellen Buchstaben geändert werden. Dezimale Zahlen werden mit »\D« ausgegeben, Strings mit »\s« usw. Interessant ist hier jedoch, daß mit »\a« die Adresse, bezogen auf die nächstgelegene Funktion, ausgegeben werden kann. Dadurch erkennt man, daß der Programmzähler in der Funktion *icvt_s* anhielt, was bereits beim Abbruch ausgegeben wurde.

Dies hilft einem allerdings noch keinen Schritt weiter. Statt mit *cont()* bis zum Abbruch durchzulaufen, kann *next()* verwendet werden, um das Programm schrittweise von einer Befehlszeile zur nächsten gehen zu lassen. Da dies jedesmal die Assemblerschritte als Breakpoints und zum Schluß mit *src* die aktuelle Position anzeigt, seien hier nur drei Schritte als Beispiel angezeigt.

```
acid: next()
183: breakpoint    main+0x16    JGE    main+0x48(SB)
183: breakpoint    main+0x48    MOVL    $.string+0x2d(SB),CX
/usr/gun/Buch/search.c:24
 19
 20        if (argc < 2)
 21            fprint(2, "usage: %s <word>\n", argv[0]),
 22            exits("missing word to search for");
 23
>24        while (scanf("%s", words[n++]) > 0) ;
 25
 26        if ((i = search(argv[1], words, n)) >= 0)
 27            print("found %s as word #%d\n", argv[1], i+1);
 28
 29        exits("");
acid: next()
183: breakpoint   main+0x4d   MOVL   CX,0x0(SP)
183: breakpoint   main+0x50   MOVL   n+0x100c(SP),DX
183: breakpoint   main+0x57   INCL   n+0x100c(SP)
183: breakpoint   main+0x5e   MOVL   words+0xc(SP)(DX*4),CX
183: breakpoint   main+0x62   MOVL   CX,0x4(SP)
183: breakpoint   main+0x66   CALL   scanf(SB)
183: breakpoint   scanf   SUBL   $0x14,SP
/sys/src/libstdio/iolib.h:577
 /sys/src/libstdio/iolib.h:577
```

```
 acid: next()
183: breakpoint     scanf+0x3     LEA     fmt+0x0(FP),AX
/sys/src/libstdio/scanf.c:7
 2      */
 3     #include "iolib.h"
 4     int scanf(const char *fmt, ...){
 5         int n;
 6         va_list args;
>7         va_start(args, fmt);
 8         n=vfscanf(stdin, fmt, args);
 9         va_end(args);
 10         return n;
 11     }
```

In der 24. Zeile des Quelltexts geht *acid* von *search* über die *header*-Datei *iolib.h* nach *scanf.c* und tiefer, bis am Ende der Fehler auftaucht. Dies deutet darauf hin, daß vermutlich etwas mit dem Argument *words*, das der Funktion *scanf()* übergeben wird, nicht stimmt, so daß eine genauere Untersuchung nötig ist.

In *acid* gibt es drei verschiedene Arten von Variablen. Variablen wie *PC*, die einen Registerwert anzeigen, oder die Adressen der einzelnen Programmfunktionen wie *main()*. Die Adressenvariablen sind von *acid* selbst definiert und enthalten Informationen über den Zustand des Programms, welches untersucht wird.

Als zweites gibt es Variablen, zu denen z.B. *progargs* gehört. Sie werden vom Benutzer festgelegt. Schließlich kann auch auf die Variablen des Programms zugegriffen werden. Handelt es sich um eine lokale Variable innerhalb einer Funktion, so muß der Funktionsname angegeben werden. Dabei entstehen Adressen: *main:n* ist zum Beispiel die Adresse der Variablen *n* in der Funktion *main*. Diese Adressen können wie in *C* üblich verwendet werden: **main:n* oder *main:n[0]* ist der aktuelle Wert von *n*. Speziell für Vektoren empfiehlt sich die zweite Syntax. Der Wert der Variablen *n* im Hauptprogramm wird mit **main:n* angegeben und die Adresse im Stack mit *main:n*. Lokale Variablen können erst angezeigt werden, wenn der Programmzähler sich schon innerhalb der Funktion befindet. Nachdem das Programm also beim Aufruf von *scanf* abbrach, kann der Wert der lokalen Variablen wie folgt ermittelt werden:

```
acid: main:n
0xbfff5fc4
acid: *main:n
0x00000001
acid: main:n[0]
0x00000001
acid: main:words
0xbfff4fc4
acid: main:words[0]
0x00000000
acid: main:words[1023]
0x00000000
acid: *(main:words[0])
<stdin>:10: (error) indir: can't translate address 0x0
```

Words sollte ein 2-dimensionaler Vektor sein, damit er mehrere Wörter, aufnehmen kann. *main:words* macht deutlich, daß eine Adresse für die Variable zur Verfügung steht. Mit *main:words[0], ..., main:words[1023]* erkennt man die Ursache des Abbruchs, denn jede Zeile *main:words[i]* sollte eine Adresse besitzen, die auf die einzelnen Buchstaben der Zeile verweist. In Wirklichkeit enthalten sie allerdings Null-Werte als Adresse, an deren Position *scanf()* nichts hineinschreiben kann, womit es zum Abbruch kommt. Deshalb muß Speicherplatz für die Zeilen bereitgestellt werden, zum Beispiel mit

```
do
    words[n] = malloc(256);
while (scanf("%s", words[n++]) > 0);
```

Beim zweiten Versuch stellt sich heraus, daß jedes zu suchende Wort die Nummer 1 zu sein scheint. Da nach der *do-while*-Schleife nur noch die Funktion *search()* aufgerufen wird, muß sich der Fehler entweder im Aufruf selbst oder in der Funktion befinden.

```
%term search Suche
Die Suche nach einem Wort
ist eigentlich sehr einfach.
(EOT)
found Suche as word #1
```

Um den zweiten Fehler zu finden, wird zunächst ein Breakpoint zu Beginn der *search()*-Funktion gesetzt, mit *cont()* bis dorthin gelaufen und dann mit *next()* bis zum *return* gegangen. Die aktuellen Variablenwerte werden diesmal durch Anhängen von »\D« und »\s« mit ihrem jeweiligen Variablentyp versehen.

```
acid: bpset(search)
acid: cont()
Die Suche nach einem Wort
ist eigentlich sehr einfach.
(EOT) 226: breakpoint    search    SUBL    $0xc,SP
acid: next() (bis return)
acid: *search:i\D
0
acid: *(*search:target\s)
Suche
acid: *((*search:words)[0]\s)
Die
acid: *((*search:words)[1]\s)
Suche
acid: *((main:words)[4]\s)
Wort
```

Wäre *target* ein lokaler String, würde man seinen Wert mit **(search:target\s)* ausgeben. Da es sich aber um einen Parameterstring handelt, muß **(*search:target)\s* angegeben werden.

Die *target*- und die *words*-Variablen besitzen also die richtigen Werte. Die *for*-Schleife hält aber verfrüht an, denn offensichtlich muß der Vergleich mit *strcmp()* auf Null überprüft werden:

```
for (i = 0; i < n; i++)
    if (strcmp(target, words[i]) == 0)
        return i;
```

Das Programm findet nun die richtige Nummer für das Wort heraus.

```
% search Suche
Die Suche nach einem Wort
ist eigentlich sehr einfach.
(EOT) found Suche as word #2
```

Die Analyse des Programms ergab selbstverständlich noch nicht alle potentiell auftretenden Fehler. Es wird z.B. nicht geprüft, ob mehr als *1024* Wörter eingelesen werden oder ein Wort aus mehr als *256* Zeichen besteht und damit die Speicherkapazität überschritten wird. Bei der Eingabe zu vieler Zeichen kann mit *acid* wiederum die genaue Fehlerposition im Quellcode, dieses Mal bei *abort.c*, ermittelt und untersucht werden.

9.2 Programmierung von Debugger-Kommandos

Beim vorhergehenden Beispiel wurde entweder das Programm mit *cont()* bis zum nächsten Breakpoint ausgeführt oder mit Hilfe von *next()* schrittweise durchlaufen. Der Breakpoint ließ sich nur auf den Anfang einer Funktion, z.B. mit *bpset(search)*, festlegen und nicht auf eine bestimmte Zeile im Quellcode. Ein Kommando hierfür kann aber selbst erstellt werden, da *acid* eine eigene Programmiersprache besitzt. Sie ähnelt von der Grammatik her *ANSI C*. Genau wie in *C* können Anweisungen in Blöcken mit geschweiften Klammern *{}* zusammengefaßt und Ausdrücke mit runden Klammern *()* eingeschlossen werden. Auch Kontrollstrukturen sind ähnlich.

```
if Ausdruck then Anweisung else Anweisung
if Ausdruck then Anweisung
while Ausdruck do Anweisung
loop Startausdruck, Endausdruck do Anweisung
```

Die Ausdrücke brauchen nicht in runde Klammern gesetzt werden, da sie durch *then* und *do* abgeschlossen werden. Die *if*-Abfrage und die *while*-Schleife sind selbsterklärend. Beginnend mit dem Startausdruck, wird mit *loop* der Wert des Ausdrucks jeweils um 1 erhöht bis der Endausdruck erreicht oder überschritten wird. Es gibt jedoch keine explizite Variable als Zähler. Die Ausdrücke werden kurz vor dem Beginn der Schleife berechnet und nicht zur Laufzeit.

Es gibt nur vier Typen von Variablen: *integer, float, string* und *list*. Bei *integer* und *float* handelt es sich um Zahlen wie in *C*. Ein *string* ist *"text"*. *List* stellt eine Liste von mehreren Werten dar, die von geschweiften Klammern umgeben und durch Kommata getrennt werden: *{a,b,c}*. Die Variablentypen ergeben sich aus den Werten, mit denen die Variablen versehen werden. Neben den vier Typen gibt es noch Formate für die Ausgabe der Variablen. Wie bereits im obigen Beispiel gezeigt wurde, kann eine Variable oder ein Wert mit einem bestimmten Format ausgegeben werden, indem ein »\« mit einem Buchstaben angehängt wird. Um ein Format einer Variablen generell zu ändern, kann *fmt()* verwendet werden. *x=fmt(x, 'D')* sorgt dafür, daß *x* immer dezimal ausgegeben wird.

Eine Funktion wird folgendermaßen definiert:

`defn` *Funktionsname* `(`*Parameterliste*`)` *Block*

Parameterliste:
Variable
Parameterliste, Variable

Eine mit *local* innerhalb einer Funktion definierte Variable steht lokal beim Funktionsaufruf zur Verfügung. Falls sie den gleichen Namen wie eine globale Variable besitzt, ersetzt sie diese so lange, bis die Funktion mit *return Ausdruck* verlassen wird. Ein *return* ohne einen Ausdruck liefert eine leere Liste *{}* als Resultat. Mit *print()* kann ein zurückgelieferter Wert schließlich auch angezeigt werden.

Beispiel

Als Beispiel soll ein Kommando definiert werden, um die Adresse einer Zeile im Programmquelltext zu erhalten. Als Hilfsfunktion dient *onemore*: Diese Funktion erhält eine Adresse und liefert nach Möglichkeit die Adresse der darauffolgenden Zeile im Quelltext.

```
defn onemore(address) {
    local line;

    if fnbound(address) == {} ||
        !(line = pcline(address)) then
        error("address not in a function");

    while pcline(address) == line do
        address = address + 1;  // Schritt fuer Schritt
    return address;
}
```

Anfangs wird kontrolliert, ob die Adresse *address*, von der aus gestartet wird, innerhalb einer Funktion liegt. Die *acid*-Funktion *fnbound()* gibt bei Erfolg ein Listenpaar, z.B. *{0x00001069, 0x0000114f}*, der aktuellen Funktion und bei Mißerfolg eine leere Liste zurück. *pcline()* hilft, die Adresse einer zusätzlichen Prüfung zu unterziehen, da die Funktion die Nummer der Quelltextzeile einer Adresse liefert oder *0*, wenn für die Adresse keine Zeile geortet werden kann. Trat ein Fehler auf, endet die Funktion mit einer Fehlermeldung durch *error()*. Andernfalls durchläuft *onemore* die übergebene Adresse so lange, bis ihre Zeile im Quelltext nicht mehr mit der zu Beginn in der lokalen Variablen *line* gespeicherten Zeile übereinstimmt.

Die neue Funktion kann natürlich in *acid* selbst eingegeben werden, es ist aber auch möglich, sie in einer Datei zu speichern und mit *include("datei")* einzufügen oder sie sogar beim Start von *acid* mit *acid -l datei search* direkt zu laden.

```
acid: include("onemore.acid")
acid: print(file("search.c")[pcline(main)-1])
int main(int argc, char * argv[])
acid: print(file("search.c")[pcline(onemore(main))-1])
    int i, n = 0;
acid: print(file("search.c")[pcline(onemore(onemore(main)))-1])
    if (argc < 2)
```

```
acid: print(onemore(0x0))
<stdin>:10: (error) address not in a function
acid: print(onemore(main-0x1))
0x00001069
```

File() liefert eine Liste der einzelnen Zeilen einer Datei, so daß die *n+1*-te Zeile mit *file(datei)[n]* gelesen werden kann. Beim näheren Betrachten der ursprünglichen Datei von *search.c* fällt auf, daß nicht immer genau eine Zeile weitergesprungen wird. Es werden auch einige ausgelassen, wenn sie von *pcline()* als Adressen nicht erkannt werden, da es sich z.B. um leere Zeilen handelt. Außerdem werden nicht alle denkbaren falschen Adreßwerte, wie *main-0x1*, abgefangen.

```
include("onemore.acid")

defn number(address, n) {
    local bound;        // Funktionsanfang und –ende

    bound = fnbound(address);
    n = n + 1;          // nur fuer src()

    if bound == {} || !pcline(address) then
        error("wrong address/line");

                        // Zeilen einzeln durchlaufen
    while pcline(address) < n do
        address = onemore(address);

    if pcline(address) > n then
        error("incorrect line number");

    return address;
}
```

Mit *number()* können die Zeilen nun durchschritten werden, bis eine gesuchte Zeile erreicht ist. Dieses Mal benötigt die Funktion die Adresse, von der aus der Durchlauf beginnen soll, und die Zeilennumer. Als Adresse gibt man am besten immer die Adresse einer Funktion wie *main()* oder *search()* an. Die Zeilennummer hingegen bezieht sich auf die gesamte Datei, so wie die Zeilen auch in *src()* angezeigt werden, und nicht auf eine Funktion. In *src()* sind die Zeilennummern immer um eins größer als im Quelltext. Deshalb wird der Zähler *n* zu Beginn auch um eins in *number()* erhöht.

Erneut werden zuerst Adresse und Zeilennummer auf ihre Korrektheit überprüft. Anschließend wird jede Zeile in einer *while-do*-Schleife mit *onemore()* durchlaufen, solange die entsprechende Zeile noch nicht erreicht ist. Auch hier kann nur die Zeile angesteuert werden, die von *pcline()* als solche im Quelltext erkannt wird, d.h., wenn sie im Code mit Befehlen gekennzeichnet ist. Deshalb können u.a. keine Leerzeilen ausgewählt werden.

```
acid: src(number(main,19))
/usr/gun/Buch/Src/8/search.c:19
 14    {
 15        int i, n = 0;
 16
 17        char * words[1024];
```

```
 18
>19         if (argc < 2)
 20             fprint(2, "usage: %s <word>\n", argv[0]),
 21             exits("missing word to search for");
 22
 23         do
 24             words[n] = malloc(256);
acid: src(number(main,18))
<stdin>:8: (error) incorrect line number
```

9.3 Strukturen

Strukturierte Variablen in *C* und *alef* können mit den von *acid* zur Verfügung gestellten Variablentypen und Formaten nicht unmittelbar verwaltet und ausgegeben werden. *Acid* verwendet die Begriffe *adt, aggr, complex* und *union* synonym, um eine etwas eigenwillige Art von Aggregaten zu definieren. Die *C*-Struktur

```
struct String {
    char * string;
    long length;
};
```

beschreibt man in *acid* mit

```
aggr String
{
    'X' 0 string;
    'D' 4 length;
};
```

Eine Komponente besteht aus dem Format, der Position in der Struktur und aus ihrem Namen. Zur Ausgabe sollte man etwa folgende Funktion konstruieren:

```
defn String(addr) {
    aggr String addr;
    print(" string  ", addr.string\X, "\n");
    print(" length  ", addr.length, "\n");
};
```

Damit diese Strukturen nicht ständig selbst für *acid* erstellt werden müssen, besitzen die Compiler von *C* und *alef* die Optionen *-a* und *-aa*. Mit Hilfe der Optionen gibt der Compiler auf der Standardausgabe für jede der von ihm im Programmtext entdeckten Strukturen die passende *acid*-Struktur und eine Funktion für die Ausgabe einer solchen Struktur aus. Die Option *-a* gibt sogar auch die aus den Include-Dateien entwickelten Strukturen umgewandelt aus, während *-aa* ausschließlich die aus dem Programm selbst berücksichtigt. Es ist am besten, die Standardausgabe in eine Datei umzulenken, welche man in *acid* mit *include()* lädt.

Die Struktur und ihre Funktion für die Ausgabe bekommen jeweils denselben Namen wie die Struktur im Programm. Zusätzlich wird noch eine Variable wie *sizeofString* erstellt, die die Länge der Struktur angibt.

Handelt es sich bei einer Komponente um einen Zeiger, so wird dieser mit dem hexadezimalen Format versehen, obwohl es auch ein Zeiger auf einen String oder

auf eine andere Struktur-Variable sein könnte. Dennoch kann eine Struktur besser mit dieser Hilfe untersucht werden. Es sei eine Variable *string* vom Typ *String* gegeben, deren Werte überprüft werden sollen.

```
acid: *(main:string+4\D)
12
acid: *(*main:string\s)
Hello World!
```

Einfacher geht dies mit der vom Compiler entwickelten Struktur und Funktion.

```
acid: include("struct.acid")
acid: String(main:string)
    string    0x0000433c
    length    12
acid: sizeofString
0x00000008
```

Eine Variable kann auch als *aggr*-Struktur deklariert werden, damit die einzelnen Komponenten anders ausgegeben oder wiederverwendet werden können.

```
acid: main:string
0xbfff5fd4
acid: aggr String main:string
acid: main:string
    string    0x0000433c
    length    12
acid: *(main:string.string\s)
Hello World!
acid: main:string.length\X
0x0000000c
```

Es fällt auf, daß dann auf den Wert der Variablen nicht mit »*« zugegriffen werden braucht. Dies folgt daraus, daß *aggr* Variablenadressen als Struktur deklariert. Eine in *acid* definierte Variable kann demnach nicht als Struktur deklariert werden, weil sie selbst als Adresse auf eine mögliche Struktur zeigen müßte. Mit *acid* lassen sich aber leider keine Variablen als Struktur definieren, sondern nur deklarieren.

9.4 Listen

Neben *integer, float* und *string* verfügt *acid* über einen sehr nützlichen vierten Variablentyp *list*. Mit ihm kann, wie bereits erwähnt, eine Liste definiert werden, bestehend aus mehreren Werten. Mit den Operatoren *head, tail, append* und *delete* kann auf eine Liste zugegriffen und diese ggf. verändert werden.

```
acid: list={1\D, "Hello", 3}
acid: list
{1, "Hello", 0x00000003}
acid: list[2]
0x00000003
acid: head list
1
acid: tail list
{"Hello", 0x00000003}
```

```
acid: append list, "weiter"
{1, "Hello", 0x00000003, "weiter"}
acid: delete list, 2
{1, "Hello", "weiter"}
```

Mit *head* bekommt man den ersten Wert der Liste und mit *tail* die übrigen. Durch *append* wird der Liste ein Wert hinzugefügt, und mit *delete* kann ein Wert an einer vorgegebenen Position wieder entfernt werden. Positionen werden als 0 gezählt.

9.5 Formate

Es gibt eine Vielzahl von unterschiedlichen Formaten, mit denen eine Variable versehen werden kann. Ein Format dient für die Ausgabe und für einige Operatoren. Der *++*-Operator erhöht eine Variable nicht unbedingt um den Wert *1*, sondern um den vom Format vorgegebenen Wert. Handelte es sich um eine mit *D* formatierte Variable, würde *4* hinzuaddiert, da *D* die Variable als *4* Bytes lang betrachtet. Mit *x=x+1* kann die Variable hingegen direkt um *1* erhöht werden.

9.6 Zusammenfassung

Zwar ist *acid* allein noch sehr simpel und an Assembler-Programmen orientiert, aber es bietet eine Vielzahl neuer Ideen zur Untersuchung von laufenden Programmen an. Es gibt fast keine Einschränkung der Möglichkeiten, da der Benutzer seine eigenen Ideen durch neue Funktionen realisieren kann und sowohl Assembler als auch C und Alef unterstützt werden.

10 *alef*

Alef wurde als eine völlig neue Programmiersprache entwickelt. Die Syntax entspricht zwar in etwa der bekannten C-Syntax, aber es wurden neue innovative Ideen als Anweisungen, Operatoren und Typen realisiert, die in C, wenn überhaupt, nur als erweiterte Funktionen in einigen Bibliotheken vorhanden sind.

Die bekannten Anweisungen *if*, *for*, *switch*, *while* etc. wurden von C übernommen. Ebenso besitzen die Funktionsköpfe die gleiche Syntax, und es gibt immer noch die meisten der Bibliotheksfunktionen wie *print()*, *malloc()* und *open()*. Deshalb werden hier nur die Neuerungen von *alef* beschrieben.

Jedes Programm kann in *alef* mit mehreren, unabhängigen Programmzählern, den sogenannten Threads, in parallel ablaufende Teile aufgeteilt werden. Wenn ein Programm also gleichzeitig die Eingabe eines Texts und die Bewegung der Maus überwachen soll, kann dies z.B. mit zwei Threads erfolgen. Der erste Thread wartet auf die Tastatureingabe und legt jedes Zeichen, das er bekommen hat, im Speicher ab, während der zweite sich die aktuelle Mausposition merkt.

Es gibt in *alef* zwei Arten von Threads. Ein Programm besteht aus mindestens einem Prozeß, welcher wiederum aus mindestens einer Task besteht. Task wie Prozeß bezeichnet man als Thread; aber ein Prozeß läuft pseudo-parallel und bei Multiprozessor-Rechnern sogar vollständig parallel zu anderen Prozessen ab, während eine Task nur eine Art Co-Routine in einem Prozeß ist. Wenn eine Task ausgeführt wird, werden die anderen Tasks in demselben Prozeß angehalten bis die erste Task selbst aufhört oder in eine Art Ruhezustand versetzt wird.

Threads können über Channels miteinander kommunizieren und sich dabei synchronisieren. Ein Channel muß dabei von einem Thread zunächst erstellt werden und kann dann von verschiedenen Threads zum Datenaustausch genutzt werden.

Als zweite herausragende Neuerung bietet *alef* einen abstrakten Datentyp *adt*, der auf den ersten Blick einer objektorientierten Variante von Klassendefinitionen wie in C++ zu entsprechen scheint. Jedoch handelt es sich bei einer *adt* nur um eine verbesserte Struktur mit eingebauten Funktionen. Wie später gezeigt wird, kann mit Hilfe dieses Datentypen strukturierter programmiert und sogar eine Art Vererbung ausgenutzt werden.

10.1 Basistypen, Struktur und Union

Hello-World schreibt man in *alef* genau wie in C. Insofern sei hier schon gleich eine kleine Variante entwickelt, die den Text *"Hello World !"* in eine *aggr*-Struktur ablegt. Sie entspricht der *struct*-Struktur in C, bestehend aus einzelnen Komponenten. Wie in C greift man auf *String s* zum Beispiel mit *s.name* und auf *String * p* mit *p->length* oder auch *(*p).length* zu. Der String wird wie üblich mit *print()* ausgegeben und das Programm schließlich mit *exits()* beendet, wobei es sich bei *nil* um einen Null-Zeiger handelt.

```
#include <alef.h>

aggr String {
    byte * name;    /* Struktur wie in C */
    int length;
};

void main(void)
{
    String s;
                    /* Zuweisungen und Ausgabe */
    s.name = "Hello World";
    s.length = strlen(s.name);
    print("'%s' is %d bytes long.\n",
        s.name, s.length);
    exits(nil);
}
```

Als Basistypen für Variablen bietet *alef* natürlich die üblichen Varianten von *int, long, float* etc. an. Darüber hinaus gibt es noch *byte* statt *char* und *chan* für die bereits erwähnten Channels zwischen Threads. Alle Typen besitzen eine fest vorgegebene Größe. *Unsigned*, *long* und *short* werden nur durch den ersten Buchstaben dargestellt, so daß *usint* in C *unsigned short int* entspricht.

Name	*Größe*	*Typ*
`byte`	8 Bits	unsigned byte
`int`	32 Bits	signed integer
`chan`	32 Bits	channel
`float`	64 Bits	floating point
`sint`	16 Bits	signed short integer
`usint`	16 Bits	unsigned short integer
`uint`	32 Bits	unsigned integer
`lint`	64 Bits	long signed integer
`ulint`	64 Bits	unsigned long integer

Struktur *aggr* und Union *union* entsprechen den C-Variablentypen *struct* und *union*. Bei beiden und bei den später verwendeten abstrakten Datentypen gibt es in *alef* die Möglichkeit, Komponentennamen auszulassen. Die unbenannten Komponenten müssen allerdings vom Typ her Unikate sein, d.h., es darf zum Beispiel nicht zwei *String*-Komponenten ohne Namen im unten stehenden *SpaceObject* geben. Das bedeutet, eine Struktur kann weitere Strukturen enthalten, die nicht durch Komponentennamen gekennzeichnet sind.

```
aggr Star {
    float temp;
    Planet * system;
};

aggr Planet {
    ulint men;
};
```

```
aggr SpaceObject {
    String;              /* namenlose Komponente */
    float radius;
    union {
        Star;            /* namenlos */
        Planet;          /* namenlos */
    };
};

SpaceObject earth;
earth.name = "Earth";    /* in String */
earth.length = 5;
earth.radius = 20000.0;
earth.men = 6000000000; /* in Planet in union */
```

In einer *SpaceObject*-Variablen kann man die Werte einer *String*-Komponente direkt mit *earth.name* und *earth.length* ermitteln. Für die *union*-Komponenten ist es noch praktischer, da kein zusätzlicher, unnötiger Komponentenname eingeführt werden muß. Statt *earth.dummy.earth.men* läßt sich einfach *earth.men* verwenden.

10.2 Tupel

Ein Tupel setzt sich aus mehreren Komponenten zusammen, insgesamt umgeben von Klammern. *("Hello World !", 11)* ist der Wert eines Tupels *tuple(byte*, int)*. Es handelt sich also um ein Aggregat oder eine Struktur, deren Komponenten im Gegensatz zu *aggr* keine Namen besitzen. Ein Tupel wird mit *tuple(...)* deklariert. Der String kann somit auch als Tupel ohne Komponentennamen dargestellt werden. Hierbei sind die Typen der einzelnen Komponenten innerhalb der runden Klammern aufgelistet.

```
tuple(byte *, int) t;
t = ("Hello World", 11);
```

Um bestimmte Werte aus einem Tupel herauszulesen, hilft es, das Tupel einem anderen Tupel zuzuweisen, in dem alle Komponenten mit *nil* versehen werden, die nicht benötigt werden.

```
byte * b;
int i;
tuple(float, byte *, int, uint) t;

t = (3.14, "Hello", 42, 6);
(nil, b, i, nil) = t;
```

Nur die Variablen *b* und *i* erhalten ihre speziellen Werte. Die *float*- und *uint*-Komponenten bleiben unberücksichtigt. Sollen mehrere Werte durch eine Funktion zurückgeliefert werden, so kann dies mit einem einzigen Tupel realisiert werden.

Tupel sind insbesondere nützlich, um einzelne Werte für Funktionsaufrufe zu bündeln und auch um gebündelte Werte von der Funktion zurückgeliefert zu bekommen.

Die Funktion *putin()* im folgenden Beispielprogramm fügt am Anfang eines Strings einen anderen String ein. Die beiden Strings werden dabei als zwei Tupel

übergeben. Zurück kommt der neue String und die Anzahl der restlichen nicht geänderten Buchstaben des ersten Stringarguments. Auch diese Werte sind ein Tupel.

```
(String, int) putin(String a, String b)
{
    byte * s;

    if (b.length > a.length)
        return ((nil,0), a.length—b.length);
    s = malloc(a.length+1);
    strcpy(s, a.name);
    return ((memcpy(s,b.name,b.length), a.length),
        a.length—b.length);
}
```

Der Wert *nil* im String-Tupel, den *putin* liefert, wenn der zweite String nicht vollständig in den ersten paßt, ist in diesem Fall ein Null-Zeiger für den Typ *byte **. Nur als L-Wert in einem Tupel bedeutet *nil*, daß die entsprechende Komponente des Tupels vernachlässigt werden soll. Hier ist ein Aufruf von *putin()*:

```
void main(void)
{
    String s;
    int i;

    (s, i) = putin(("World World",11), ("Hello",5));
    if (i > 0)
        print("%s (%d)\n", s.name, s.length);
    else
        print("second string bigger than first\n");
    exits(nil);
}
```

Ein Vorteil der Tupel ist, daß z.B. *(a,b)=(b,a)* die Variablen *a* und *b* korrekt austauscht, weil beide zuerst auf der rechten Seite ausgewertet und dann direkt separat zugewiesen werden. Die Abhängigkeit der Variablen voneinander wird automatisch erkannt.

10.3 Abstrakte Datentypen

Wie ein *aggr* besitzt auch ein abstrakter Datentyp *adt* Komponenten mit Daten. Hinzu kommen jedoch noch Operatoren, die auf die Komponenten angewendet werden können. Obwohl dies einer objektorientierten Klassendefinition zu entsprechen scheint, sind die Anwendungsmöglichkeiten in gewisser Weise eingeschränkt. Dennoch können die strukturellen Vorteile praktisch verwendet werden.

Als ein abstrakter Datentyp soll ein Vektor erstellt werden, der Strings verwaltet.

```
adt Vector {
            String * buf;
    extern  int dim;
    extern  int count;
            int inc;
```

```
            Vector*  init(int);
        intern  void     increase(*Vector);
            void     insert(*Vector, String, int);
            String   get(*Vector, int);
};
```

Der Datentyp enthält als erstes die Komponenten, gefolgt von den Funktionsdeklarationen der Operatoren. Die Reihenfolge ist hierbei beliebig. Auf die Komponenten, wie z.B. auf die Strings, kann nur über die Operatoren zugegriffen werden. Wird *extern* vor eine Komponente gesetzt, kann sie auch von außerhalb gelesen und verändert werden. Somit läßt sich die Anzahl Elemente in *main()* ausgeben.

Die Operatoren hingegen können überall aufgerufen werden. Falls vor einer Deklaration *intern* steht, kann sie ausschließlich in einem Operator desselben Datentyps genutzt werden. Der Operator *increase* zur Vergrößerung des Speicherplatzes eines Vektors sei hier lokal nur für die *Vector*-Operatoren verfügbar.

Jede Deklaration eines Operators enthält die Variablentypen der Argumente. Ein Verweis auf den *adt*-Namen wie **Vector* kennzeichnet, daß ein *adt*-Objekt für den Aufruf nötig ist.

```
Vector * Vector.init(int inc)
{
    Vector * v;

    alloc v;            /* Speicher fuer Vektor         */
    v—>count = 0;
    v—>inc   = inc;
    v—>dim   = inc;
                        /* Speicher fuer Vektorelemente */
    check v—>buf = malloc(v—>dim * sizeof(String)),
        "out of memory";
                        /* Vektorelemente initialisieren*/
    memset(v—>buf, 0, v—>dim * sizeof(String));
    return v;
}
```

Eine Funktion wird als Operator eines Datentyps durch ein Präfix wie *Vector.* gekennzeichnet. Beispielsweise wird *get()* durch *String get(*Vector, int)* im *adt* deklariert, mit *String Vector.get(Vector * v, int i) {...}* außerhalb definiert und mit *v->get(i-1)* in *main()* aufgerufen.

Zur Erzeugung eines Vektors muß anfangs die *init()*-Funktion aufgerufen werden, die einen Zeiger auf eine dynamische Instanz vom Typ *Vector* zurückliefert. Auf jede Komponente dieser Variablen kann nach dem Erzeugen mit *alloc* innerhalb eines Operators mit *v->component* zugegriffen werden.

```
void main(int argc, byte ** argv)
{
    int i;
    Vector * v;
    String s;

    if (argc < 2) exits("missing args");
```

```
        v = Vector.init(3);          /* Vektor erzeugen und  */
                                     /* Arg—Strings einfuegen*/
        v—>insert((argv[i=1::argc], strlen(argv[i])), i—1);

        print("%d elements (free: %d)\n",
            v—>count, v—>dim — v—>count);
        for (i = 1; i < argc; i++) {
            s = v—>get(i—1);         /* String—Tupel holen   */
            print("%d: %s(%d)\n", i, s.name, s.length);
        }
        exits(nil);
    }
```

Als Beispiel dienen die Argumente, die dem Programm übergeben werden. Sie werden in einen Vektor eingelesen, der mit *Vector.init()* erstellt wird, mit *v->insert()* Strings einliest und mit *v->get()* wieder ausgibt. Die Strings aus dem *argv*-Vektor werden nacheinander eingelesen.

Durch *i=1::argc* wird *insert()* von *1* bis *argc-1* durchlaufen. Wie eine *for()*-Schleife kann man mit *::* ein Kommando mehrfach nacheinander ausführen. Die Strings für *insert()* werden als Tupel, bestehend aus dem Text und der jeweiligen Textlänge, übergeben.

init() entspricht einem Konstruktor wie in C++ . Der *adt*-Operator *init()* kann allerdings einen beliebigen Namen besitzen. Es muß nur ein Vektor mit *alloc* erstellt und zurückgeliefert werden. Die anderen Operatoren können dann mit Hilfe dieses Vektors aufgerufen werden. Insofern läßt sich auch ein Operator ähnlich einem Destruktor erstellen, welches hier jedoch als Übungsaufgabe verbleibe. *Alloc* reserviert soviel Speicher wie ein Zeiger eines bestimmten Typs für einen Bereich braucht, auf den er verweist. Die Speicherreservierung des Zeigers *Vector * v* durch *Alloc v* entspricht also *v = malloc(sizeof(Vector))*. Kann nicht genügend Speicher bereitgestellt werden, wird das Programm mit *check* abgebrochen.

10.4 Iteratoren

In der *main()*-Funktion wird der sogenannte Iterator *a::b* verwendet. Mit einem Iterator kann man ein einzelnes Kommando wiederholt hintereinander ausführen, beginnend mit dem Wert *a* bis zum Wert kleiner als *b*. *i=0::10* entspricht also *for(i=0; i<10; i++)*. Ist *b<a*, kommt es zu keiner Ausführung.

Es ist auch möglich, mehrere Iteratoren in einem Kommando zu verwenden. Zum Beispiel werden durch *x[0::3][0::3]=1* alle Elemente der Maxtrix *x[3][3]* mit dem Wert *1* versehen. Die Reihenfolge der Abläufe bei mehr als einem Iterator ist dabei undefiniert.

10.5 Fehlerbehandlung

Mit *check arg1, arg2* kann in *alef* ein Fehler abgefangen werden. Ist das erste Argument vor dem Komma *0*, wird automatisch eine Fehlerroutine aufgerufen, die das zweite Argument auf der Standardfehlerausgabe anzeigt und das Programm mit

dem Status von *arg2* abbricht. Folglich terminiert das Programm, falls in *insert()* kein Speicher mehr reserviert oder in *get()* ein falscher Index angegeben wurde.

```
void Vector.increase(Vector * v)
{
    check               /* Speicher erweitern, pruefen  */
        v->buf = realloc(v->buf,
            (v->dim += v->inc) * sizeof(String)),
        "out of memory";
    memset(v->buf+v->dim-v->inc, 0, v->inc * sizeof(String));
}

void Vector.insert(Vector * v, String s, int i)
{
    while (i >= v->dim) /* Speicher ggf. vergroessern   */
        v->increase();
    if (v->buf[i].name == nil)
        v->count++;
    v->buf[i] = s;      /* neuen String einfuegen       */
}

String Vector.get(Vector * v, int i)
{
    check i >= 0 && i < v->dim, "wrong index";
    return v->buf[i];   /* String zurueckliefern        */
}
```

Die Standardfehlerroutine befindet sich in der Variablen *void (*ALEFcheck)(byte *, byte *)*, so daß die Fehlerbehandlung ggf. verändert werden kann, indem diese Routine ersetzt wird.

Mit Hilfe der letzten drei Funktionen ist das Programm lauffähig.

```
% vector Dies ist ein Test.
4 elements (free: 2)
1: Dies(4)
2: ist(3)
3: ein(3)
4: Test.(5)
```

10.6 Vererbung

Mittels namenloser Komponenten kann in *alef* sogar eine Art Vererbung von abstrakten Datentypen realisiert werden. Dies ist aber nur eine Annäherung an die objektorientierte Vererbung. Als Beispiel soll ein Stapel *Stack* von *Vector* abstammen. Der *Stack*-Datentyp soll dabei folgende Funktionalität besitzen.

```
Stack * st;
String s;

st = .Stack.init();
st->push(("Hello World", 11));
s = st->pop();
```

Mit *init()* wird ein Stapel erstellt, in den mit *push()* ein String am Ende angehängt und mit *pop()* der zuletzt gespeicherte String wieder herausgeholt werden kann. Damit die Funktionen und Variablen vom *Vector* wiederverwendet werden können, fügt man ihn als erste namenlose Komponente in *Stack* ein.

```
adt Stack {
    Vector;
    int pos;

    Stack*  init(void);
    void    push(*Stack, String);
    String  pop(*Stack);
};
```

Ein leichtes Problem tritt in der Funktion *init()* auf. In ihr muß die Initialisierungsfunktion von *Vector* aufgerufen werden. Sie reserviert aber nur für den Vektor genügend Speicher und nicht für den Stapel, der zusätzliche Operatoren und Variablen enthält. Insofern muß der reservierte Speicher mit *realloc()* erweitert werden.

```
Stack * Stack.init(void)
{
    Stack * st;

    st = (Stack *) .Vector.init(10);
    check st = realloc(st, sizeof(Stack)),
        "out of memory";
    st—>pos = 0;
    return st;
}
```

Danach können alle von *Vector* zur Verfügung gestellten Funktionen ausgenutzt werden, um *push()* und *pop()* zu erstellen.

```
void Stack.push(Stack * st, String s)
{
    st—>insert(s, st—>pos++);
}

String Stack.pop(Stack * st)
{
    String s;

    s = st—>get(——st—>pos);
    st—>insert((String) (nil, 0), st—>pos);
    return s;
}
```

Die Vorteile der Vererbung in *alef* liegen auf der Hand, wie z.B. Wiederverwendbarkeit und Verkapselung von Daten und Funktionen, jedoch muß auch auf einige Nachteile bzw. Einschränkungen geachtet werden. So sind die Funktionen eines geerbten Datentyps auch außerhalb verwendbar, also nicht vollständig im erbenden Datentyp eingekapselt. *Insert()* und *get()* von *Vector* können also über *Stack * st* mit *st->insert()* und *st->get()* aufgerufen werden. Es fehlt die Option, nur bestimmte Funktionen eines Datentyps zu veröffentlichen, also nur *push()* und *pop()*.

10.7 Polymorphe Variablentypen

Neben den Basistypen *adt*, *aggr* und *union* gibt es noch einen polymorphen Variablentyp. Die Anwendung ist sehr komplex und umfassend. Deshalb werden nur die grundlegenden Möglichkeiten gezeigt. Mit *typedef* legt man den Namen eines polymorphen Typs fest. Eine Variable eines solchen Typs besteht aus einem Zeiger auf den eigentlichen Wert und einem Etikett, das den momentan gewählten Typ angibt. Mit dem Cast *(alloc Poly)* wird eine Variable oder ein Wert eines vorgegebenen Typs in die polymorphe Form umgewandelt.

```
typedef Poly;
Poly p1, p2;
int i;
float f;

i = 10;
f = 3.1415;
p1 = (alloc Poly) i;
p2 = (alloc Poly) f;
```

Mit *sizeof(p1)* bekommt man die Speichergröße des Wertes heraus, auf den eine polymorphe Variable verweist. Die Größe von *Poly* selbst ermittelt man mit *sizeof(Poly)*.

Die Unterscheidung einzelner konkreter Variablentypen läßt sich mit *typeof*, ähnlich dem *switch*, erreichen. Hinter jeder *case*-Anweisung kann ein Typ, wie z.B. *int*, stehen. Dadurch wird die polymorphe Variable in *case* als *int*-Variable angesehen.

```
typedef Poly;

aggr String {
    byte * name;
    int length;
};

void polyprint(Poly p)
{
    typeof p {
        case int:
            print("int = %d", p);
            break;
        case float:
            print("float = %f", p);
            break;
        case String:
            print("bytes = %s (strlen: %d)",p.name,p.length);
            break;
    }
    print(" (size: %d)\n", sizeof(p));
}
```

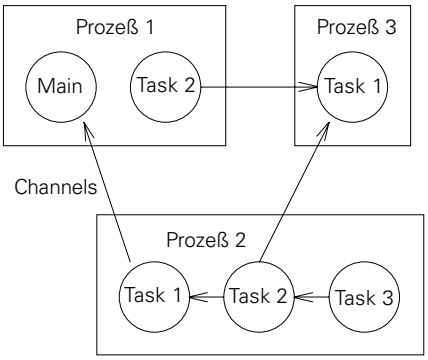
```
void main(void)
{
    polyprint((alloc Poly) 10);
    polyprint((alloc Poly) 3.1415);
    polyprint((alloc Poly) (11, "Hello World"));
    exits(nil);
}
```

Die Funktion *polyprint()* gibt mit Hilfe von *typeof* das polymorphe Argument *p* entsprechend seines jeweiligen Typs als *int*, *float* oder als Struktur *String* aus.

```
% polyprint
int = 10 (size: 4)
float = 3.141500 (size: 8)
bytes = Hello World (strlen: 11) (size: 8)
```

10.8 Prozesse und Tasks

Ein *alef*-Programm kann aus mehreren Prozessen bestehen, von denen jeder eine oder mehrere Tasks enthält. Als erstes sollen die Tasks erläutert werden, die sozusagen die Atome eines Programms sind.



Tasks

Ein *alef*-Prozeß unterteilt sich in einen oder mehrere Tasks. Unter Tasks versteht man in diesem Fall synchron geplante Threads. In einem Prozeß läuft immer genau eine Task ab. Alle anderen Tasks sind währenddessen blockiert in diesem Prozeß, und zwar so lange, bis die ausgewählte Task

- ihre Ausführung mit *terminate()* oder *return* beendet,
- eine Nachricht über einen Channel (siehe Abschnitt Channels) empfangen oder abschicken möchte,
- durch QLock (10.9) geblockt wird oder
- bis sie selbst eine andere Task startet.

Eine Task wird mit *task* ausgelöst. An *task* hängt man eine Liste mit Funktionsaufrufen, wobei jeder Funktionsaufruf durch eine Task erfolgt. Mit *terminate()* oder mit

dem Ende einer Task-Funktion werden die Tasks wieder einzeln beendet. Die Funktion *exits()* terminiert alle Tasks eines Prozesses und dadurch auch den Prozeß selbst, denn ein Prozeß besteht aus mindestens einer Task.

```
void tasking(int i)
{
    sleep(nrand(10000));
    print("leaving task #%d\n", i);
}

void main(void)
{
    srand(time());
    task tasking(1), tasking(2);
    sleep(nrand(10000));
    print("leaving main task\n");
    terminate(nil);
}
```

Die *main*-Task erzeugt anfangs zwei weitere Tasks. Die Task-Funktionen werden bzgl. ihrer Reihenfolge in der Liste von links nach rechts aufgerufen. Da Tasks nicht vollkommen parallel zueinander ablaufen können, warten die neuen Tasks erst auf die Beendigung der *main*-Task und laufen dann nacheinander ab. Aus diesem Grund ergibt sich trotz der zufallsabhängigen zeitlichen Verlängerung der Tasks mit *sleep(nrand(10000))* immer die gleiche Ausgabe:

```
% three2
leaving main task
leaving task #1
leaving task #2
```

Prozesse

Mit *proc* lassen sich neben dem *main*-Prozeß weitere Prozesse gleichzeitig starten. An *proc* wird wie bei *task* eine Liste von Funktionsaufrufen gehängt, wobei jeder Funktionsaufruf durch einen separaten Prozeß erfolgt. Die Prozesse laufen auf Multiprozessor-Rechnern vollkommen parallel, während sie ansonsten pseudo-parallel in kurzen Intervallen hintereinander ablaufen. Dabei besitzen Threads bisher in allen *alef*-Implementierungen ausschließlich *shared memory*. Alle Prozesse verfügen also über denselben Speicherbereich.

Im folgenden Programm werden neben dem *main*-Prozeß zwei weitere Prozesse mit *proc* gestartet, die beide die Funktion *process()* aufrufen.

```
void process(void)
{
    sleep(nrand(10000));
    print("leaving process %d\n", getpid());
}

void main(void)
{
    srand(time());
    proc process(), process();
    sleep(nrand(10000));
```

```
        print("leaving main process %d\n", getpid());
        exits(nil);
    }
```

Damit die Parallelität deutlich wird, erfolgt die Ausgabe der Prozeß-ID in der Funktion *process()* wiederum erst nach einer zufällig bestimmten Zeitdauer. In den meisten Fällen wird zuerst der *main*-Prozeß beendet, da die neuen Prozesse erst erzeugt werden und ihre Funktion aufrufen müssen. Die beiden anderen Prozesse geben eher willkürlich zueinander ihre PID aus. Zur gleichen Zeit kann deshalb schon wieder ein *rc*-Kommando wie *echo hello* ausgeführt werden, da die zwei anderen Prozesse immer noch im Hintergrund weiterlaufen und evtl. erst viel später terminieren.

```
% three
leaving main process 339
leaving process 341
leaving process 340

% three
leaving main process 343
% echo hello
hello
% leaving process 344
leaving process 345
```

Im Gegensatz zum *fork()*-Kommando gibt es in *alef* keine Unterteilung in Vater- und Sohnprozesse. Alle Prozesse sind gleichberechtigt, d.h., keiner muß, soweit es nicht explizit vorgesehen ist, auf einen anderen Prozeß warten. Der Vaterprozeß kann deshalb noch vor seinen Sohnprozessen beendet werden.

Tasks sind meistens schneller und effektiver als Prozesse. Sie können einfach gestartet und wieder beendet werden. Insofern sind sie besonders dann gut verwendbar, wenn eine Aufgabe nicht vollkommen parallel bearbeitet werden soll, sondern wenn es mehr auf die Geschwindigkeit kurzer Programmteile ankommt. Ansonsten sind Prozesse gut für längere parallele Abläufe einsetzbar. Deshalb kann das erste Programmbeispiel der Prozesse auch mit Tasks realisiert werden, indem *proc* durch *task* ersetzt wird. Zur Beendigung sollte *terminate()* verwendet werden. Damit wird genau eine Task beendet. Ruft man wie in C *exits()* zur Beendigung auf, so wird der komplette Prozeß mit all seinen Tasks beendet, noch bevor die Tasks vollständig ausgeführt wurden.

Channels

Das erste Programm für Prozesse soll im folgenden tatsächlich dann vollständig terminieren, wenn der *main*-Prozeß als letzter *exits()* ausführt. Hierfür werden nun Channels eingesetzt. Threads können miteinander kommunizieren, indem einer von ihnen einen Channel definiert. Danach muß er allen anderen Prozessen den Channel mitteilen, damit sie über diesen Kanal Nachrichten senden und empfangen können.

```
void process(chan(int) channel)
{
    int i;
```

```
    sleep(nrand(10000));
    i = <—channel;
    print("leaving process %d (received %d)\n",
        getpid(), i);
}

void main(void)
{
    chan(int) one, two;     /* Channels deklarieren */
    alloc one, two;         /* Channels erzeugen    */
    srand(time());
    proc process(one), process(two);
    one <—= 1;
    two <—= 2;
    print("leaving main process %d\n", getpid());
    exits(nil);
}
```

Wie in der ersten Version des Programms werden neben dem *main*-Prozeß zwei Prozesse gestartet, die *process()* aufrufen. Dieses Mal werden vorher noch die Channels *one* und *two* deklariert und Speicher für sie reserviert, damit über sie *int*-Werte geschickt werden können. Der Variablentyp *chan* wird deshalb mit *(int)* versehen. Den beiden neuen Prozessen werden dann beim Funktionsaufruf die Channels als Parameterwert mitgeteilt. Mit *one <-= 1* und *two <-= 2* sendet der *main*-Prozeß den anderen jeweils eine *int*-Zahl zu. Der *main*-Prozeß wartet so lange, bis die Daten vom anderen Prozeß mit *i = <- channel* empfangen werden. Da dies erst nach dem *sleep* stattfindet, kann es eine Weile dauern, bis das Programm terminiert.

```
% one
leaving process 430 (received 1)
leaving process 431 (received 2)
leaving main process 429
%
```

Es ist mit den Channels sichergestellt, daß der *main*-Prozeß erst dann *exits()* aufruft, wenn die neuen Prozesse ihre Nachricht über die Channels empfangen haben. Zwar erfolgt die Ausgabe in *process()* erst nach dem Empfangen der Zahlen, so daß die *main*-Ausgabe sogar noch vor einer anderen Ausgabe erscheinen könnte. Aber es ist relativ unwahrscheinlich, da zuvor ein längeres *sleep()* ausgeführt wird.

Par und Alt

Die *int*-Zahlen werden leider hintereinander abgeschickt. Folglich wartet der *main*-Prozeß bis der erste Prozeß seine Daten empfängt, und erst dann wird der zweite Wert abgeschickt. Mit *par* können diese beiden Anweisungen parallel ablaufen. Jede Anweisung innerhalb des Blocks von *par* wird parallel zu allen anderen Anweisungen ausgeführt.

```
par {
    one <—= 1;
    two <—= 2;
}
```

An die mit *proc* gestarteten Prozesse werden nun vollkommen parallel die Zahlen *1* und *2* gesendet. Da auch die Prozesse parallel ablaufen, können die Zahlen gleichzeitig empfangen werden.

```
% two
leaving process 435 (received 2)
leaving process 434 (received 1)
leaving main process 433
```

Par startet so viele Prozesse, wie es Anweisungen im Block von *Par* gibt. Die Prozesse laufen dabei völlig unabhängig voneinander. Erst wenn alle Anweisungen ausgeführt wurden, arbeitet das Programm mit der nach *par* folgenden Anweisung weiter. Es ist natürlich möglich, jede der Anweisungen mit *a[i] <-= channel[i]* auf eine Nachricht warten zu lassen. Falls aber eine Nachricht empfangen wird, werden die anderen Prozesse nicht unterbrochen.

Für die Lösung eines solchen Problems gibt es *alt*. Mit *alt* wird auf mehrere Channels gleichzeitig gewartet. Sobald eine Nachricht ankommt, wird *alt* beendet und die nachfolgende Anweisung ausgeführt. Von der Syntax her entspricht *alt* dabei einer *switch*-Anweisung mit *case* und *break*. Es gibt allerdings kein *default*. Alle Fälle müssen vom Senden oder vom Empfangen einer Nachricht über einen Channel abhängen.

```
chan(int) a;
chan(byte *) b;

alt {
    case a = <—ch1:
        print("%d\n", a);
        break;
    case b = <—ch2:
        print("%s\n", b);
        break;
}
```

Auf den ersten Blick scheinen *alt* und *par* sehr ähnlich zu sein. Im Gegensatz zu *par* besteht *alt* jedoch nur aus einem einzigen Prozeß, der auf eine Nachricht wartet. Falls mehr als ein Channel bereit ist, wird zufällig ein Channel ausgewählt. Nach dem ersten Empfangen oder Senden wird der *alt*-Block bereits verlassen.

10.9 Synchronisierung

Wie in C enthält auch *alef* Lock-Mechanismen, um mehrere Threads in bestimmten Bereichen zu synchronisieren. Vorgegeben sind in der Include-Datei *alef.h* die drei *adt*s *Lock*, *QLock* und *RWlock*.

Mit *Lock.lock()* blockiert ein Thread einen freien *Lock*, so daß kein anderer Thread, der auch *Lock.lock()* aufruft, weiterlaufen kann. Ist das *Lock* bereits von einem Vorgänger blockiert, wird der Thread in eine Warteschleife versetzt. Versuchen mehrere Threads gleichzeitig zu blockieren, wird zufällig einer von ihnen ausgewählt. Durch *Lock.unlock()* wird der *Lock* wieder freigegeben. Dabei spielt es keine Rolle, ob derselbe Thread oder ein fremder *Lock.unlock()* aufruft. Gegenüber

Lock.lock() geht die Funktion *Lock.canlock()* nicht in eine Warteschleife über, falls *Lock* bereits blockiert ist. Statt dessen wird eine Null zurückgeliefert. Ansonsten wird der freie *Lock* wieder blockiert.

QLock basiert auf *Lock* und besitzt die gleichen Funktionen. Darüber hinaus enthält *QLock* jedoch noch eine Warteschlange für die einzelnen Threads, welche bei *Qlock.lock()* warten müssen. Ist *Qlock* wieder frei, wird der nächste Thread aus der Warteschlange herausgeholt und als Nachfolger festgelegt. Die Reihenfolge ist dabei undefiniert. Vorgegeben ist bisher das Prinzip *last-in, first-out*.

Eine zusätzliche Erweiterung ist *RWlock*, die *Qlock*s verwendet. Dieser *adt* ist insbesondere für das Lesen und Schreiben von Daten nützlich. Damit ein Bereich nicht gleichzeitig von mehreren Threads überschrieben wird, ruft man *RWlock.wlock()* und *RWlock.wunlock()* auf, um einen kritischen Bereich zu schützen. Lesen dürfen hingegen mit *RWlock.rlock* und *RWlock.runlock()* viele Threads. Ein Thread kann allerdings nicht ein *RWlock* zum Lesen bzw. Schreiben blockieren, wenn *RWlock* bereits zum Schreiben bzw. Lesen blockiert wurde.

Bei allen drei *adt*s muß vorher der Speicherbereich der Struktur mit *alloc* festgelegt werden. Wird der Speicher zwischendurch wieder freigegeben, verlieren *Qlock* und *RWlock* ihre Schleifen mit den nachfolgenden Threads. Infolgedessen sollte man es vermeiden, den Speicher freizugeben, wenn ein *Qlock* noch blockiert ist.

10.10 Beispiel für Prozesse

Zum Abschluß wird nun ein kleines Programm gezeigt, welches mit Hilfe von Prozessen und Channels die Aufgabe eines *grep*-Kommandos auf eine völlig andere Art und Weise als gewohnt löst. Das Programm *newgrep [word] [word] ...* soll nur dann eine Zeile von der Standardeingabe wieder auf die Standardausgabe ausgeben, wenn sie *alle* als Argumente übergebenen Worte enthält.

Z.B.:

```
% newgrep world yours
hello world
this world is yours
this world is yours (Ausgabe)
and mine
(^D)
```

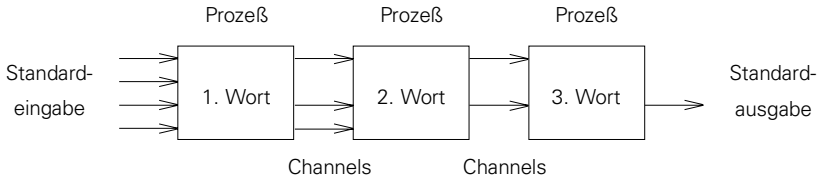
Beispiel *newgrep.l*:

```
#include <alef.h>
#include <bio.h>

/* scan ist die Eingabefunktion, liest eine Zeile von der   */
/* Standardeingabe und liefert sie zurueck.                 */
byte * scan(chan(byte*) in)
{
    if (in == nil) {            /* falls kein Channel       */
        Biobuf * bio;
        byte * input;
```

```
        alloc bio;              /* Zeile von Standardeingabe*/
        bio—>init(0, OREAD);    /* einlesen                 */
        input = bio—>rdline('\n');
        return input;           /* und zurueckliefern       */
    } else
        return <—in;            /* sonst von Channel lesen  */
}

/* newgrep sieht in einer Zeile nach text und leitet sie    */
/* ueber einen Channel weiter, falls text in der Zeile.     */
void newgrep(chan(byte*) in, chan(byte*) out,
    chan(int) end, byte * text)
{
    byte * input;
                                /* Zeilen ueber Channel oder*/
    while ((input = scan(in)) != nil) /* Standardeingabe    */
        if (strstr(input, text)) {  /* lesen und ausgeben   */
            if (out == nil)         /* falls text enthalten */
                print("%s", input); /* und Standardausgabe  */
            else
                out <—= input;  /* sonst weiterleiten       */
        }
    if (out == nil)             /* falls keine Ausgabe mehr */
        end <—= 1;              /* — also letzter Prozess — */
    else                        /* main—Prozess mitteilen   */
        out <—= input;          /* sonst weiterleiten       */
}

/* Das Hauptprogramm sucht mit mehreren Prozessen ueber     */
/* Channels nach einem Wort in der Standardeingabe und gibt */
/* die gefundenen Zeilen auf der Standardausgabe aus        */
void main(int argc, byte ** argv)
{
    if (argc > 1) {             /* mindest. ein Argument    */
        int i;
        chan(byte*) in, out;    /* Channels fuer Ein/Ausgabe*/
        chan(int) end;          /* Channel fuer Beendigung  */

        alloc end;
        in = nil;               /* Standardeingabe anfangen */
        for (i = 1; i < argc—1; i++) {
            alloc out;          /* Ausgabe ueber Channel    */
                                /* grep als neuen Prozess   */
                                /* fuer jedes Wort von argv */
            proc newgrep(in, out, end, argv[i]);
            alloc in;           /* Ein— und Ausgabe wechseln*/
            (in, out) = (out, in);
        }
        out = nil;              /* Standardausgabe setzen   */
                                /* und letztes Wort pruefen */
        proc newgrep(in, out, end, argv[argc—1]);
        <—end;                  /* letzten Prozess abwarten */
    }
    exits(nil);
}
```

Für jedes Wort, welches als Argument beim Aufruf übergeben wurde, erstellt *newgrep* einen neuen Prozeß. Die einzelnen Prozesse sind jeweils nur für ein Wort verantwortlich. Dieses Wort wird der Funktion des Prozesses als Argument übergeben, ebenso ein Channel für die Eingabe und einer für die Ausgabe.

Ein Prozeß gibt eine Zeile sofort an den nächsten Prozeß weiter, wenn sie das gesuchte Wort enthält. Danach kann der Prozeß sich weiter auf die Suche begeben, während gleichzeitig der nächste Prozeß die ihm übergebene Zeile nach seinem Wort durchsucht. Somit werden die Prozesse zwar sequentiell gestartet, aber danach prüfen sie parallel die jeweils folgenden Zeilen und teilen das Ergebnis dem Nachfolger mit. Jeder wartet nur auf das Ergebnis seines Vorgängers.



Anfangs soll der erste Prozeß von der Standardeingabe lesen. Deshalb erhält er *nil* als Argument für *in*. Außerdem wird ein Channel für die Ausgabe bereitgestellt. In *newgrep()* liest der Prozeß mit *scan()* seine Zeilen von der Standardeingabe und überprüft, ob sie das Wort enthalten. Ist dies der Fall, wird es sofort über die Channel für die Ausgabe *out* weitergeleitet zum nächsten Prozeß.

In *alef* gibt es zwar kein *scanf()*, statt dessen aber den abstrakten Datentyp *Biobuf* für eine gepufferte Ein- und Ausgabe. Da es sich um einen *adt* handelt, muß eine Variable wie beim Vektor *vec* (siehe oben) erst allokiert werden. Dann kann die Funktion *bio->init()* aufgerufen werden, um *bio* als Standardpuffer mit Leseberechtigung zu initialisieren. Erst danach ist es möglich, von der Standardeingabe eine Zeile mit *input = bio->rdline('0)* einzulesen.

Alle weiteren Prozesse erhalten ihre Zeilen über den Channel *in* vom vorherigen Prozeß und leiten sie weiter, wenn sie feststellen, daß ihr Wort sich in den Zeilen befindet. Mit Hilfe der Tupel *(in, out) = (out, in)* können die Channels für die Ein- und Ausgabe direkt ausgetauscht werden, damit der folgende Prozeß einen neuen Channel für die Ausgabe und den alten Ausgabechannel als Eingabechannel bekommt.

Mit *nil* als Argument des Ausgabechannels liefert der letzte Prozeß das Ergebnis schließlich an die Standardausgabe. Außerdem muß der *main*-Prozeß über *end* benachrichtigt werden, daß der letzte Prozeß mit der Bearbeitung fertig ist. Da es in *alef* keine Vater-und Sohn-Prozesse gibt, würde der *main*-Prozeß ansonsten als einer der ersten terminieren und zur Shell zurückkehren.

10.11 Zusammenfassung

Obwohl *alef* von der Syntax her in etwa C entspricht, werden neue Variablentypen und Anweisungen angeboten. Besonders fällt dabei *adt* auf, welches als abstrakter Datentyp eine Annäherung an die Objektorientierung ist. Eine wesentliche Neue-

rung von *alef* sind die Generierung und Verwaltung von Threads und Channels. Sie ist besonders nützlich für die Programmierung von parallel ablaufenden Programmen. Zudem lassen sich die Daten zwischen den Prozessen besser und einfacher austauschen als mit *notify()* in Plan 9 oder als Signale in UNIX, da sie als Variablentypen gehandhabt werden.

11
Systemaufrufe Bibliotheksfunktionen

Der Übergang zwischen Systemaufrufen, die der Kern bearbeitet, und Bibliotheksfunktionen, die darauf aufbauen, ist fließend. Beide Arten von Funktionen sind im Kapitel 2 des Manuals auf ca. 130 Seiten besprochen. Im vorliegenden Kapitel soll ein Überblick über die wichtigsten Funktionen und Systemaufrufe gegeben werden, wobei die Zusammenfassung nach Funktionalität erfolgt.

Plan 9 kennt nur 37 Systemaufrufe, deren Assemblerquellen sich in dem Katalog */sys/src/libc/9syscall* finden, nachdem das dortige *mkfile* zur Ausführung gebracht worden ist. In */sys/src/libc/9syscall/sys.h* sind die Konstanten definiert, die als Index in *systab* innerhalb der Kernquellen verwendet werden. Die dem Namen nach etwas ungewöhnliche Konstante *_X3* wird mit dem Systemaufruf *sysdeath()* abgebunden. Dieser kann allerdings nicht über die Bibliothek aufgerufen werden.

Für C-Programme muß man die Definitionsdateien *u.h* und *libc.h* verwenden. Die Definitionsdateien sorgen mit *#pragma*-Anweisungen, wie *#pragma lib "libc.a"*, implizit dafür, daß die richtige Bibliothek gebunden wird. Definitionsdateien hängen, im Gegensatz zu *include*-Dateien von Unix, nicht voneinander ab und korrespondieren direkt mit ihren Bibliotheken. Sie sind nicht mittels *ifndef*-Technologie gegen ein mehrfaches Ersetzen der *include*-Zeile durch den Inhalt der Datei geschützt.

»Echte« Systemaufrufe kann man mit dem Kommando *syscall(1)* ausführen.

```
% syscall write 1 hello 3
helsyscall: return 3, no error
```

schreibt drei Bytes der Zeichenkette *hello* in den Filedeskriptor 1. Ein Versuch, in Filedeskriptor 0 zu schreiben, endet, wie auch sonst, mit einer Fehlermeldung.

```
% syscall write 0 hello 4
syscall: return -1, error:inappropriate use of fd
```

11.1 Fehlertext

Systemaufrufe liefern -1 bei Fehlern und setzen in diesem Fall einen Text, den man mit *errstr()* abholen oder mit *print* unter Verwendung des Formats *%r* ausgeben kann. Bibliotheksfunktionen setzen diesen Text implizit, wenn sie Systemaufrufe verwenden.

`int errstr(char answer [ERRLEN])`	Fehlertext holen
`int print(char * format, ...)`	mit *%r* àla *printf()* ausgeben
`void perror(char * s)`	Fehlertext mit Präfix ausgeben
`void syslog(int cons, char * logn, char * fmt, ...)`	→ *syslogd*

11.2 Datei-Management

Die Systemaufrufe zum Dateimanagement selbst sind ähnlich zu UNIX.

`int access(char * path, int mode)`	Zugriff prüfen
`long read(int fd, void * buf, long len)`	lesen
`long write(int fd, void * buf, long len)`	schreiben
`int dirread(int fd, Dir * buf, long nbytes)`	Katalog lesen
`long seek(int fd, long n, int type)`	positionieren
`int open(char * path, int mode)`	Zugriff eröffnen
`int create(char * path, int omode, ulong perm)`	... dabei Datei erzeugen
`int close(int fd)`	beenden
`int dup(int old, int new)`	Deskriptor kopieren
`int pipe(int fd[2])`	bidirektionale Pipe
`int stat(char * path, char * edir)`	Informationen über eine Datei
`int wstat(char * path, char * edir)`	... ändern
`int fstat(int fd, char * edir)`	... über einen Filedeskriptor
`int fwstat(int fd, char * edir)`	... ändern
`int dirstat(char * path, Dir * dir)`	... über einen Katalog
`int dirwstat(char * path, Dir * dir)`	... ändern
`int dirfstat(int fd, Dir * dir)`	... über einen Katalog (*fd*)
`int dirfwstat(int fd, Dir * dir)`	... ändern
`int remove(char * path)`	Datei oder Katalog entfernen

Bei *open()*und *create()* gibt es OREAD, OWRITE und ORDWR als Zugriffsart. Dazu kommen OTRUNC (Abschneiden auf Länge 0), OCEXEC (implizites *close()* bei *exec()*) und ORCLOSE (Löschen beim letzten *close()*). Bei *create()* erzeugt man einen Katalog mit CHDIR, und OTRUNC ist bei einer existenten Datei impliziert. Bei *remove()* muß ein Katalog leer sein. Eine Pipe ist bidirektional und message-orientiert, das heißt, ein *read()* liest so viel, wie ein einziges *write()* geschrieben hat.

11.3 Allgemeine Information

Die folgenden Funktionen beschaffen Informationen aus dem System. Technisch gesehen sind es keine Systemaufrufe, weil sie i.a.R. nur den Inhalt einer Datei aus dem aktuellen Namensraum wiedergeben:

`int times (long t[4])`	*/dev/cputime*	ms verbrauchte Zeit
`double cputime (void)`		Summe in Sekunden

`char * getenv (char * name)` `int putenv (char * name,` `char * val)`	*/env/name*	Wert per *malloc()*
`int getpid (void)`	*/dev/pid*	Prozeßnummer
`int getppid (void)`	*/dev/ppid*	Erzeuger
`char * getuser (void)`	*/dev/user*	Benutzer
`char * getwd (char * buf,` `int size)`		aktueller Katalog
`long time (long * tp)`	*/dev/time*	Uhrzeit

11.4 Namespace-Management

Die Systemaufrufe *bind()*, *mount()* und *unmount()* werden im Kapitel 4 besprochen.

11.5 Memory-Management

Es gibt, wie unter Unix, die Funktionen der *alloc*-Familie sowie auch Funktionen, mit denen Segmente im virtuellen Speicher beschafft bzw. wieder freigegeben werden können.

```
int   segattach(int attr, char *class, void *va, ulong len)
int   segdetach(void *addr)
int   segfree(void *va, ulong len)
int   segbrk(void *saddr, void *addr)
int   segflush(void *va, ulong len)
```

11.6 Netzwerkzugriffe

Eine entscheidende Aufgabe in einem netzwerkorientierten Betriebssystem ist der Aufbau einer Verbindung und die anschließende Datenübertragung zwischen mehreren Rechnern (siehe auch Kapitel 15). Dabei sollen Prozesse auf Rechnern in demselben Netzwerk aber auch über eine Netzwerkkante hinaus miteinander kommunizieren können. Ein Prozeß, der auf einem deutschen Rechner läuft, soll sich z.B. mit einem Prozeß auf einem Rechner in den USA unterhalten können.

Damit eine Verbindung geschaffen werden kann, muß zunächst ein bestimmter Prozeß, der sogenannte Server, einen Port zur Verfügung stellen, über den andere Prozesse, die Klienten, ihn ansprechen können. Der Server öffnet unter Plan 9 einen Port mit *announce()*. Der Funktion muß er dabei seine eigene Netzwerkadresse als Argument übergeben. Das Format einer solchen Adresse *protocol!ipaddr!port* ergibt sich aus einem Protokoll *il, tcp, udp, ip, dk* oder *net* für die Datenübertragung, der IP-Adresse des Servers und dem Namen oder der Nummer des Ports. Dabei sind sowohl der Protokolltyp als auch die Angabe des Ports optional. Der Server

kann seine eigene IP-Adresse auch als * abkürzen. Falls der Protokolltyp *net* oder auch kein Typ angegeben wird, wird ein Protokoll verwendet, welches auf beiden Rechnern benutzt werden kann. Meistens ist dies unter Plan 9 das Standardprotokoll IL. Nachdem der Server seine Adresse angegeben hat, kann er mit *listen()* auf eine Anfrage eines Klienten warten, um sie schließlich mit *accept()* anzunehmen oder mit *reject()* abzulehnen. Danach können die Daten durch *read()* und *write()* untereinander ausgetauscht werden.

Ein Server kann sich auch mit mehr als einem Klienten unterhalten, solange die Klienten ihre Nachrichten hintereinander verschicken und danach die Verbindung jeweils mit *close()* beenden. Der Server wartet danach wieder mit *listen()* auf eine Verbindung mit dem nächsten Klienten. Eine parallel laufende Datenübertragung ist nur möglich, wenn der Server neue Prozesse erzeugt, die sich um die eigentlichen Antworten kümmern.

Die folgenden beiden Beispielprogramme zeigen, wie sich ein Klient und ein Server miteinander unterhalten können, indem sie jeweils eine Nachricht erhalten und eine abschicken. Der Server gibt seine Adresse als *il!*!6666* bekannt. Das bedeutet, die Verbindung soll über den Port der Nummer *6666* mit Hilfe des *il*-Protokolltyps erfolgen. Nachdem der Server die Verbindung mit einem Klienten hergestellt hat, liest er mit dem von *accept* gelieferten Filedeskriptor *fd* eine Frage des Klienten mit *read* und schickt mit *write* eine Antwort zurück. Zum Schluß wird die Verbindung mit *close* beendet.

```
#include <u.h>
#include <libc.h>
#include <auth.h>

int main (int argc, char * argv[])
{
    char adir[40], ldir[40];
    int lcfd, fd;
    char question[256];
    char answer[] = "Server: Fine, thank you client.";
    long length = strlen(answer)+1;
                              /* Server-Adresse angeben    */
    if (announce("il!*!6666", adir) < 0)
        exits("announce failed");

                             /* auf Anfrage eines Klienten  */
    print("listening ...\n");
                             /* warten und annehmen         */
    if ((lcfd = listen(adir, ldir))< 0)
        exits("listen failed");
    if ((fd = accept(lcfd, ldir)) < 0)
        exits("accept failed");
                             /* Frage empfangen und Antwort */
                             /* zuruecksenden               */
    if (read(fd, question, sizeof(question)) < 0)
        exits("read error");
    if (write(fd, answer, length) != length)
        exits("write error");
```

```
    print("fd = %d\nlcfd = %d\nadir = %s\nldir = %s\n",
        fd, lcfd, adir, ldir);

    print("%s\n", question);
    close(fd); close(lcfd); /* Verbindung fuer alle        */
    exits("");              /* Klienten beenden            */
}
```

Der Klient wählt mit *dial()* einen Server an, um eine Verbindung zu bekommen. *dial()* liefert wiederum einen Filedeskriptor, mit Hilfe dessen der Klient Nachrichten verschicken und erhalten kann. In diesem Fall schickt der Klient mit *write()* eine Frage an den Server, liest dessen Antwort mit *read()* und gibt sie aus.

```
#include <u.h>
#include <libc.h>

int main(int argc, char * argv[])
{
    int fd;
    char answer[256];
    char question[] = "Client: How are you server ?";
    long length = strlen(question)+1;
                                    /* Server anwaehlen    */
    if ((fd = dial("il!newage!6666", 0, 0, 0)) < 0)
        exits("dial failed");
                                /* Frage absenden und Antwort  */
    if (write(fd, question, length) != length)  /*empfangen*/
        exits("write error");
    if (read(fd, answer, sizeof(answer)) < 0)
        exits("read error");

    print("fd = %d\n", fd);
    print("%s\n", answer);
    close(fd);                  /* Verbindung beenden      */
    exits("");
}
```

Der Server wird zunächst gestartet, um auf dem von ihm zur Verfügung gestellten Port auf die Frage des Klienten zu horchen. Danach schickt der Klient seine Frage über den Port zum Server, und der Server antwortet über denselben Port.

Server:

```
% server
listening ...
fd = 1
lcfd = 2
adir = /net/il/0
ldir = /net/il/1
Client: How are you server ?
```

Klient:

```
% client
fd = 1
Server: Fine, thank you client.
```

11.7 Ermittlung und Ausgabe von IP- und Ethernet-Adressen

IP-Adressen setzen sich aus 4 *unsigned chars* und Ethernet-Adressen aus 6 *unsigned chars* zusammen. Die Darstellungen für IP-Adressen ergeben sich aus dezimalen Zahlen von 0 bis 255, die durch einen Punkt jeweils getrennt werden, z.B. *131.173.161.42*. Die Ethernet-Adressen werden hingegen als Strings hexadezimal ohne Trennung dargestellt (z.B. *fa0398b3844f*). Mit *fmtinstall()* kann man für einen *print()*-Aufruf neue Funktionen festlegen, die die Umwandlung der Argumente für die Ausgabe vornehmen. Für *fmtinstall()* gibt es die Funktion *eipconv()*, mit der man IP- und Ethernet-Formate durch Angabe von *'I'* und *'E'* reservieren kann. Bei den verschiedenen *print()*-Aufrufen können dann diese Formate durch *%I* und *%E* für Adressen verwendet werden, so, wie eine *int*-Variable durch *%d* ausgegeben werden kann.

Umgekehrt lassen sich Strings wieder in die *unsigned char*-Adressen mit *parseip()* und *parseether()* umwandeln. Mit *myipaddr()* und *myetheraddr()* lassen sich die Adressen des Rechners herausfinden. Zum Umgang mit IP-Adressen gibt es die Funktionen *maskip()* und *equivip()*. *maskip()* verknüpft das erste Argument mit dem zweiten durch AND und liefert als drittes Argument diese Verknüpfung zurück. *equivip()* vergleicht zwei Adressen miteinander und liefert eine Zahl ungleich Null, falls beide Adressen gleich sind.

```
#include <u.h>
#include <libc.h>
#include <ip.h>

int main(int argc, char * argv[])
{
    uchar myip[4], myip2[4], andip[4];

    if (argc <= 1)
        fprint(2, "missing arg\n"), exits("missing arg");

                                /* Format fuer IP—Adressen */
    fmtinstall('I', eipconv);
                                /* eigene IP—Adresse       */
    if (myipaddr(myip, "/net/il") < 0)
        exits("missing ip—address");
    print("my IP—address: %I\n", myip);
                                /* als Argument ueberg. IP */
    if(parseip(myip2, argv[1]) < 0)
        exits("wrong ip—address");
    print("=  IP—address: %I ?\n", myip2);

                                /* IPs mit AND vergleiche  */
    maskip(myip, myip2, andip);
    print("AND: %I\n", andip);
                                /* IPs vergleichen         */
    if (!equivip(myip, myip2))
        print("not the same address !\n");

                            /* beliebige Adresse erstellen */
    print("making an address: %s\n",
```

```
            netmkaddr("frigga", 0, "6666"));
        exits("");
}
```

Dem obigen Beispiel läßt sich als Argument eine IP-Adresse übergeben, die dann mit *equivip()* mit der Adresse des Rechners verglichen wird. Außerdem wird mit *netmkaddr()* ein String für eine Adresse vom Rechner *frigga* mit dem Port *6666* und dem Protokolltyp *net* (durch Angabe der *0* im zweiten Argument) ausgegeben.

```
% addrinfo 131.173.161.214
my IP-address: 131.173.161.213
=  IP-address: 131.173.161.214 ?
AND: 131.173.161.212
not the same address !
making an address: net!frigga!6666
```

11.8 Suche in der Network Database

In der Network Database (NDB) befinden sich alle Beschreibungen sowohl lokal als auch öffentlich bekannter Rechner. Die NDB besteht aus einer Reihe von untereinander verknüpften Dateien. In den Dateien wiederum sind Tupel auf Zeilengruppen mit beliebigen *attr=value*-Einträgen. Beginnt eine Zeile mit einem Leerzeichen, so gehört sie zum Tupel derselben Gruppe der vorherigen Zeile. Der Ursprung der NDB ist die Datei */lib/ndb/local*, in der sich ein Tupel mit einem *database*-Attribut befinden kann, welches wiederum auf andere Dateien verweist.

```
database=
        file=/lib/ndb/local
        file=/lib/ndb/global
```

Das obige Tupel in der Ursprungsdatei */lib/ndb/local* gibt an, daß als erstes */lib/ndb/local* selbst nach dem *database*-Attribut und danach */lib/ndb/global* durchsucht werden sollen. Als Standard ist *local* bereits die Datei auf der die Suche beginnt, so daß der erste Eintrag auch entfallen kann. Innerhalb eines Tupels sind die *attr-value*-Einträge auf einer Zeile vorrangig gegenüber denen, die sich über mehrere Zeilen erstrecken. Das bedeutet, *database* bekommt erst nach der Suche in den Ursprungsdateien deren Namen zugewiesen. Ansonsten sind Einträge im linken Teil einer Zeile wichtiger als im rechten und in einer oberen Zeile wichtiger als in einer unteren.

Neben *database* gibt es noch eine Vielzahl anderer Attribute zur Bestimmung der Adressen, Protokolle und Services. Ein typisches Beispiel ist:

```
ip=131.173.13.213 ether=0000c0141dab sys=newage
        dom=newage.informatik.uni-osnabrueck.de
        bootf=/386/9pccpu
        proto=il
```

Dies ist die Beschreibung des Domainservers *newage*, der das IL-Protokoll unterstützt und mit Hilfe der Datei */386/9pccpu* gestartet wird. Außerdem sind in dem Tupel die IP- und die Ethernetadresse enthalten.

NDB-Attribute

Die untere Liste gibt alle möglichen Attribute an, die bei *attr=value*-Zeilen verwendet werden können. Ebenso werden sie bei einer Abfrage mit z.B. *ndb/cs* benötigt.

9P	Parameter für 9P-Protokolle
auth	Name des Authentifizierungsservers
bootf	Datei zum Booten des Rechners
dk	Datakit-Adresse
dom	Domainname im Internet
ether	Ethernet-Adresse
fs	Fileserver des Rechners
il	IL-Services
ip	Netwerkname im Internet
ipgw	Gateway
ipmask	Name der Netzwerkmaske
port	Portnummer
proto	vom Host unterstütztes Protokoll
resticted	TCP-Service durch *port*<1024 eingeschränkt
sys	Systemname
tcp	TCP-Services
udp	UDP-Services

Durch das Attribut *proto* in einer Zeilengruppe gibt man den Protokolltyp an, der von einem Host unterstützt wird. *Restricted* legt fest, daß der zugehörige TCP-Service nur von einem Port aufgerufen werden kann, dessen Portnummer kleiner als 1024 ist.

NDB-Informationen

Mit *ipattr()* kann die Form einer Adresse festgestellt werden, also ob es sich um eine Internetnummer (*ip*), um einen Domainnamen (*dom*) oder um einen Systemnamen (*sys*) handelt. Die genauen Informationen z.B. für einen Rechner *frigga* erhält man, indem vorher eine NDB-Datei mit *ndbopen(char * file)* oder alle NDB-Dateien mit *ndbopen(0)* geöffnet werden. Dann kann mit *ipinfo(Ndb * ndb, char * ether, char * ip, char * name, Ipinfo * ip)* die *Ipinfo*-Struktur aufgefüllt werden, in der alle Informationen aus der NDB vorhanden sind. Eine kurze Übersicht aller verwendeten Befehle befindet sich am Ende dieses Abschnitts.

```
#include <u.h>
#include <libc.h>
#include <ip.h>
#include <bio.h>
#include <ndb.h>

int main(int argc, char * argv[])
{
    Ipinfo iip;
    Ndb * ndb;

    if (argc  <= 1)
        fprint(2, "missing args\n"), exits("missing args");
```

```
    fmtinstall('I', eipconv);
    fmtinstall('E', eipconv);

    if (!(ndb = ndbopen(0)))     /* Network Database oeffnen*/
        exits("ndb file not found");
    if (ipinfo(ndb, 0, 0, argv[1], &iip) < 0) {
        fprint(2, "not found\n");
        return 0;
    }
                        /* Attribute fuer IP—Systeme */
    print("ipattr=%s\n", ipattr(argv[1]));
                            /* NDB—Informationen */
    print("domain=%s ether=%E\nip=%I\nmask=%I\n",
      iip.domain, iip.etheraddr, iip.ipaddr, iip.ipmask);
    print("net=%I\nbootf=%s\n",
      iip.ipnet, iip.bootf);

    ndbclose(ndb);          /* Database wieder schliessen  */
    exits("");
}
```

Ausgabe:

```
%term ndbinfo testing
not found
%term ndbinfo frigga
ipattr=sys
domain=frigga.informatik.uni-osnabrueck.de ether=080020037e86
ip=131.173.13.42
mask=255.255.255.0
net=131.173.13.0
bootf=/sparc/9ss
```

NDB-Queries

Neben der direkten Abfrage einer Information kann man auch nach Attributen suchen lassen. Eine Zeilengruppe, die die gesuchten Attribute mit ihren Werten enthält, besitzt die folgende Struktur:

```
typedef struct Ndbtuple Ndbtuple;
struct Ndbtuple {
        char * attr;
        char * val;
        Ndbtuple * entry;
        Ndbtuple * line;
        ulong ptr;
}
```

Die ersten beiden Komponenten enthalten ein Attribut mit seinem Wert als Strings. Darauf folgen zwei Verweise auf die nächsten Tupel. *entry* zeigt dabei auf eine nullterminierte Liste aller Tupel in einer Zeilengruppe. *line* zeigt hingegen auf eine Liste einer einzigen Zeile, und der letzte Zeiger geht wieder auf das erste Tupel zurück. Auf diese Art und Weise lassen sich alle Gruppen in die einzelnen Tupel zerlegen und durch ein weiteres Leerzeichen getrennt ausgeben.

Mit dem nächsten Programm wird wiederum die NDB geöffnet und dann mittels *ndbsearch()* und *ndbnext()* nach den Zeilengruppen gesucht, die die beiden ersten übergebenen Argumente als *attr=value* enthalten. Die Ausgabe einer Zeilengruppe erfolgt durch den rekursiven Aufruf von *printndb()*, da die Tupel in einer Gruppe jeweils auf ihre Nachfolger in einer Liste verweisen.

Falls man ein drittes Argument angegeben hat, wird nur der Wert in den gefundenen Zeilengruppen ausgegeben, der zu dem Argument als Attribut gehört. Also z.B. werden mit *ndbquery il 9fs port* alle Zeilengruppen zunächst ausfindig gemacht, die als IL-Service *9fs* haben. Danach wird aus diesen Zeilengruppen der Wert des Attributs *port* ausgegeben.

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ndb.h>

void printndb(Ndbtuple * tuple) /* rekursiv Tupel aus einer*/
{                               /* Datenbasis ausgeben     */
    if (tuple -> attr && tuple -> val) {
        print("%s=%s%s", tuple -> attr, tuple -> val,
            (tuple ->  entry == tuple -> line) ? " ": "\n");
        printndb(tuple -> entry);
    }
}

int main(int argc, char * argv[])
{
    Ndb * ndb;                  /* Network Database        */
    Ndbtuple * tuple;           /* Tupel aus einer Ndb     */
    Ndbs ndbs;                  /* Verweis auf akt. Tupel  */
    char buf[Ndbvlen];          /* Puffer ndbgetval        */

    if (argc < 3)
        fprint(2, "missing args\n"), exits("missing arg");

    if (!(ndb = ndbopen(0)))
        exits("ndb file not found");

    if (argc == 3) {            /* Suche nach attr=val */
        if (!(tuple=ndbsearch(ndb, &ndbs, argv[1], argv[2])))
            print("%s=%s not found\n", argv[1], argv[2]),
            exits("");

        printndb(tuple);    /* Ausgabe: Tupel mit attr=val */
        while (tuple = ndbsnext(&ndbs, argv[1], argv[2])) {
            print("\n");
            printndb(tuple);
            ndbfree(tuple);
        }
        exits("");
    }
```

```
                                /* Suche nach speziellem Wert   */
        if (!ndbgetval(ndb, &ndbs,
                argv[1], argv[2], argv[3], buf))
            print("%s=%s with rattr=%s not found\n",
                argv[1], argv[2], argv[3]);
        else
            print("%s\n", buf);

        ndbclose(ndb);
        exits("");
    }
```

Ausgabe:

```
%next ndbquery bootf /sparc/9ss
ip=131.173.161.111 ether=080020038625 sys=garm
dom=garm.informatik.uni-osnabrueck.de
bootf=/sparc/9ss
proto=il

ip=131.173.161.42 ether=080020037e86 sys=frigga
dom=frigga.informatik.uni-osnabrueck.de
bootf=/sparc/9ss
proto=il

%next ndbquery il 9fs
il=9fs port=17008

%next ndbquery il 9fs port
17008
```

NDB-Befehle

Ndb ndbopen(char *file)*	öffnet eine NDB-Datei
*void ndbclose(Ndb *db)*	schließt die Datei wieder
char ipattr(char *name)*	liefert das Attribut, zu dem ein System gehört
*int ipinfo(Ndb *db, char *ether, char *ip, char *name, Ipinfo *iip)*	liefert eine gesuchte Zeilengruppe als *Ipinfo-Struktur* zurück
Ndbtuple ndbsearch(db *db, Ndbs *s, char *attr, char *val)*	sucht nach einem Tupel mit der ersten Übereinstimmung eines *attr-value*
Ndbtuple ndbsearch(Ndb *db, Ndbs *s, char *attr, char *val)*	sucht nach dem nächsten übereinstimmenden Tupel
Ndbtuple ndbgetval(Ndb *db, Ndbs *s, char *attr, char *val, char *rattr, char *buf)*	sucht nach einem Tupel mit einem übereinstimmenden *attr-value* und liefert den Wert des Attributs *rattr*

11.9 Abarbeitung der Kommandozeile

Die Abarbeitung des Unix-Standards für den Aufbau der Kommandozeile, wie gebündelte Flaggen, »––« zum Abbruch, »–« als Argument und zum Abbruch, wurde durch die Makros ARGBEGIN und ARGEND implementiert: Eine typische Argumentabarbeitung:

ARGF() extrahiert den Wert zu einer Option; *argv0* steht als *extern char ** zur Verfügung und wird im Makro ARGBEGIN mit *argv[0]* initialisiert.

```
/* args.c:   Argumentabarbeitung                */
#include <u.h>
#include <libc.h>
#include <stdio.h>

int      x_flag;
int      d_var;
float    f_var;
char     c_var;
char *   s_var;

void main(int argc, char ** argv)
{
    ARGBEGIN {
        case 'x':       x_flag = 1;                    break;
        case 'd':       d_var =  atoi(ARGF());         break;
        case 'f':       f_var =  (float)atof(ARGF()); break;
        case 'c':       c_var =  * ARGF();             break;
        case 's':       s_var =  ARGF();               break;
    default:    exits("illegal flag.");
    } ARGEND
}
```

11.10 Eingabe/Ausgabe

Neben den von Unix bekannten Funktionen der *stdio*-Bibliothek gibt es noch zwei weitere Funktionsgruppen, *bio* und *print*, mit denen Ein- und Ausgabe realisiert werden können. Wird eine effiziente, komfortabel zu nutzende, formatierte Ausgabe benötigt, sind die Funktionen der *bio*-Familie die richtigen. Die Nutzung dieser Funktionen zieht allerdings einen geringfügig höheren Programmieraufwand nach sich. Ist die Funktionalität der Funktionen der *stdio*-Bibliothek bis auf die eingeschränkte Formatmöglichkeit ausreichend, dann sind die Funktionen der *print*-Familie die richtigen. Mit *fmtinstall()* können neue Formate angemeldet werden. Ein Beispiel:

```
/* fmti.c: test fmtinstall (2)                    */

#include <u.h>
#include <libc.h>
#include <stdio.h>
#define  MAX_L      1024
```

```
                                /* %N soll diese F. aktivieren */
int nice(void * o, Fconv *fp  )
{
char * text = * (char **) o;
char   printed_text[MAX_L];     /* calloc waere besser     */

    sprintf(printed_text,
                "format = %c, It's nice to see you %s",
                fp->chr,  text );
    print("%s", printed_text);
    return(strlen(printed_text) );  /* # gedruckter Zeichen*/
}

main(void)
{
    fmtinstall('N', nice );     /* N als Format anmelden   */
    print("%N ...\n", "Joe" );  /* drucken                 */
    exits("");                  /* bye                     */
    return(0);

}
```

Ein Ablauf:

```
% fmti
format = N, It's nice to see you Joe ...
```

11.11 Zusammenfassung

Wie zu den frühen Tagen von Unix ist die C-Schnittstelle klein und überschaubar, trotzdem mächtig. Blickt man ins Kapitel 2 der Manualseiten, so findet man sich sofort zurecht. Dies kann man von heutigen Abkömmlingen nicht mehr behaupten.

12
Die Panel-Bibliothek

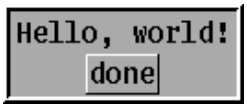
Zur benutzerfreundlichen Interaktion mit dem Anwender werden heutzutage von Applikationen graphische Objekte wie z.B. verschiedene Knöpfe, Schieberegler, Textobjekte usw. verwendet. Unter Plan 9 ist dafür die Panel-Bibliothek zuständig. Sie verbirgt in Form von einfachen C-Funktionen die mühselige Arbeit, die hinter dem Erstellen einer graphischen Benutzerschnittstelle steckt.

In der Panel-Bibliothek wird ein neuer Datentyp eingeführt. Der Typ *Panel* beschreibt das strukturelle, geometrische und funktionale Verhalten der graphischen Objekte. Jedes Panel ist fest mit einer rechteckigen Fläche verbunden, in die die Graphik des Panels gezeichnet wird. Panels sind in einer Baum-Struktur organisiert. Die Kinder eines Panels werden innerhalb seiner Fläche gezeichnet, wobei sich die Rechtecke der Kinder nicht überschneiden.

Dieses Kapitel zur Panel-Bibliothek orientiert sich stark an dem zugehörigen Abschnitt aus der Dokumentation und bildet lediglich eine kurze und unvollständige Einführung in das Thema.

12.1 Ein erstes Beispiel

Als erstes Beispiel soll eine Applikation vorgeführt werden, die aus einem kurzen Text und einem einzelnen Knopf besteht. Wird der Knopf gedrückt, so terminiert die Applikation:



Dazu wird folgender Quellcode benötigt:

```
#include <u.h>
#include <libc.h>
#include <libg.h>
#include <panel.h>

Panel *root;

void done(Panel *p, int buttons){
    USED(p, buttons);
    exits(0);
}

void main(void){
    binit(0, 0, 0);         /* Graphik-, */
    einit(Emouse);          /* Event- und */
    plinit(screen.ldepth); /* Panelbibliothek */
                            /* initialisieren  */
```

```
        root=plframe(0, 0);
        pllabel(root, 0, "Hello, world!");
        plbutton(root, 0, "done", done);

        ereshaped(screen.r);
        for(;;) plmouse(root, emouse());
    }

    void ereshaped(Rectangle r){
        screen.r=r;
        plpack(root, r);
        /* alte Graphik loeschen:  */
        bitblt(&screen, r.min, &screen, r, Zero);
        pldraw(root, &screen);
    }
```

Im Hauptprogramm werden zuerst die Initialisierungs-Funktionen zu der Graphik- (*binit*), der Event- (*einit*) und der Panel-Bibliothek (*plinit*) aufgerufen. Dabei müssen *binit* und *einit* vor *plinit* aufgerufen werden. Alle Panel-Bibliotheks-Funktionen beginnen mit dem Präfix *pl*. Die Funktion *plinit* bekommt als Argument die Bildschirmtiefe, da hiervon abhängig die Zeichnung der verschiedenen Objekte unterschiedlich ausfällt. Die Variable *screen* stammt aus *libg.h* und ist eine globale Variable vom Typ *Bitmap,* siehe *graphics*(2). Sie wird z.B. in *binit*(2) gesetzt. In den nächsten drei Zeilen wird der Panel-Baum erzeugt. *plframe* zieht einen Rahmen um seine Kinder, *pllabel* stellt einen kurzen Text dar, und *plbutton* erzeugt einen mit einer Aufschrift versehenen Knopf, der auf Maus-Aktivitäten reagieren kann.

Jede Funktion zum Kreieren graphischer Objekte liefert einen Zeiger auf das erzeugte Objekt zurück. Das erste Argument dieser Funktionen ist ein Zeiger auf das Objekt, dem es untergeordnet sein soll. Das zweite steuert die Positionierung des neuen Objekts.

Jedes Programm, welches unter Plan 9 mit Graphik arbeitet, muß eine Funktion *ereshaped* besitzen. Diese wird von der Event-Bibliothek, *event*(2), zur Neuzeichnung aufgerufen, so z.B., wenn unter 8½ die Größe des zugehörigen Fensters verändert oder das Fenster erzeugt wird.

Die Funktion *plpack* ist eine zentrale Funktion der Panel-Bibliothek. Sie berechnet die Positionierung der einzelnen Objekte des Panel-Baums, und *pldraw* zeichnet den Baum in eine Bitmap. *Bitblt* steht für **bit**-**bl**ock-**t**ransfer. Die Funktion überlagert Bits einer Bitmap mit Bits einer anderen Bitmap. Überlagern kann dabei eine von 16 möglichen Operationen sein, so z.B. eine oder-Verknüpfung der Bits, siehe *bitblt*(2). In diesem Beispiel werden alle Bits in der Bitmap *screen* gelöscht.

In einer einfachen Schleife werden am Ende des Hauptprogramms alle Maus-Aktivitäten dem *root*-Panel zur Bearbeitung weitergereicht.

Zur Verarbeitung von Maus- und Tastatur-Events stellt die Panel-Bibliothek drei Funktionen zur Verfügung:

```
void plmouse(Panel * recipient, Mouse m);
void plgrabkb(Panel * p);
void plkeyboard(rune c);
```

Das erste Argument zu *plmouse* ist das Panel, an das der Maus-Event *m* geschickt wird. *Plmouse* durchsucht den durch *recipient* spezifizierten Teilbaum nach einem maussensitiven Panel, dessen rechteckige Fläche den Event beinhaltet. Ist ein solches Panel gefunden, so wird je nach Typ des Panels verfahren. Wird z.B. ein *plbutton*-Knopf gedrückt, so ändert dieser zur Visualisierung sein Aussehen. Läßt man dann den Knopf los, so wird die im vierten Argument *plbutton* angegebene Callback-Funktion aufgerufen, und das normale Aussehen des Knopfs wird wieder hergestellt.

Tastatur-Events werden immer an ein ausgezeichnetes Panel mit *plkeyboard* geschickt. Dieses ausgezeichnete Panel wird durch Aufruf von *plgrabkbd* gesetzt.

12.2 Das Positionieren von Panels

Nachdem *plpack* das Layout des Panel-Baums bestimmt, berechnet es pro Objekt im Baum eine rechteckige Fläche, in der die Graphik des Objekts und die seiner Kinder gezeichnet wird. Die Positionierung der Kinder innerhalb dieser Fläche wird durch das zweite Argument bei deren Erzeugung gesteuert.

Plpack ist wie folgt deklariert:

```
int plpack(Panel * root, Rectangle space);
```

Die Funktion positioniert *root* innerhalb von *space*; rekursiv positioniert sie die nachgeordneten Objekte in die dann noch freie Fläche. Dabei wird in der Komponente *r* der *Panel*-Struktur eines Panel-Objekts die ihm zugeordnete Fläche hinterlegt.

Plpack startet mit einer leeren, rechteckigen Fläche und füllt diese mit den Kindern. Dabei haben früher erzeugte Panels Vorrang. Die Positionierung innerhalb der freien Fläche geschieht über die Flaggen PACKN, PACKE, PACKS, PACKW. Im folgenden zeigen die Abbildung (a) die freie Fläche vor dem Packen und die Abbildung (b) die Aufteilung nach dem Packen eines Kindes mit der Flagge PACKE.

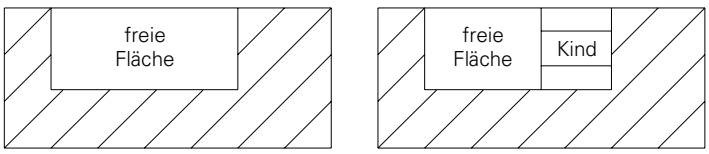
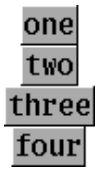


Abbildung (a) (b)

Das Positionieren von vier Knöpfen untereinander

```
root=plgroup(0, 0);
plbutton(root, PACKN, "one", 0);
plbutton(root, PACKN, "two", 0);
plbutton(root, PACKN, "three", 0);
plbutton(root, PACKN, "four", 0);
```

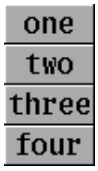
produziert folgendes Resultat:



Da die Knöpfe verschieden breit sind, sieht das Resultat unschön aus. Die Fläche innerhalb von *plgroup* ist breit genug, um das breiteste der Kinder aufzunehmen. Außerdem ist allen Knöpfen die gleiche Fläche von *plpack* zugewiesen worden, allerdings nutzen sie diese teilweise nicht komplett. Die FILLX-Flagge signalisiert einem Objekt, in horizontaler Richtung zu expandieren. Als Resultat hat

```
root=plgroup(0, 0);
plbutton(root, PACKN|FILLX, "one", 0);
plbutton(root, PACKN|FILLX, "two", 0);
plbutton(root, PACKN|FILLX, "three", 0);
plbutton(root, PACKN|FILLX, "four", 0);
```

eine gleich breite Darstellung der Knöpfe:



Analog dazu expandiert FILLY in vertikaler Richtung.

Wie bereits erwähnt, kann der von einem Panel benötigte Platz kleiner als die für dieses Panel zur Verfügung gestellte Fläche sein. Daher kann durch weitere Flaggen eine Positionierung des Panels innerhalb der Fläche erfolgen. Dabei stehen PLACEN, PLACENE, PLACEE, PLACESE, PLACES, PLACESW, PLACEW und PLACENW für die acht Kompaß-Richtungen und PLACECEN für den Mittelpunkt.

Damit erzeugt

```
root=plgroup(0,0);
plbutton(root, PACKN, "placement", 0);
plbutton(root, PACKN|PLACEE, "east", 0);
plbutton(root, PACKN|PLACEW, "west", 0);
plbutton(root, PACKN|PLACEN, "north", 0);
```

folgendes Ergebnis:

Die Flagge 0 ist äquivalent zu PACKN | PLACECEN, also Anordnung nach oben und zentriert.

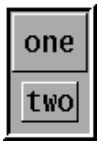
Normalerweise minimiert *plpack* die zu einem Panel gehörige Fläche. Es gibt aber vier Möglichkeiten, um diese zu vergrößern.

Erstens signalisiert die Flagge FIXED *plpack*, daß in der *fixedsize*-Komponente der *Panel*-Struktur die Dimension der Fläche vom Anwender hinterlegt ist und diese benutzt werden soll. Dabei ist FIXED eine oder-Kombination der Flaggen FIXEDX und FIXEDY. Die *fixedsize*-Komponente vom Typ *Point*, d.h., die x- und y-Komponente werden als x/y-Dimension zweckentfremdet.

Zweitens kann über die *pad*-Komponente ein zusätzlicher Freiraum außerhalb und über die *ipad*-Komponente ein zusätzlicher Freiraum innerhalb der Panel-Fläche spezifiziert werden. Auch diese beiden Struktur-Komponenten sind vom Typ *Point*. Damit hat

```
Panel *b;
root=plframe(0,0);
b=plbutton(root, PACKN, "one", 0);
b->ipad=Pt(10,10);
b=plbutton(root, PACKN, "two", 0);
b->pad=Pt(10,10);
```

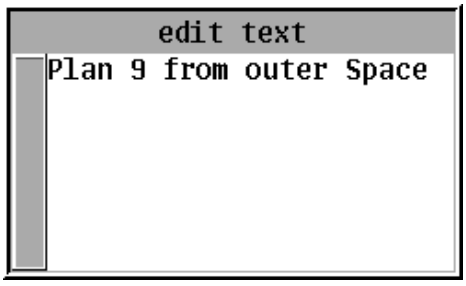
folgendes Resultat:



Drittens kann über die EXPAND-Flagge *plpack* veranlaßt werden, dem Panel die sämtliche noch freie Fläche zuzuordnen. Haben mehrere Kinder die EXPAND-Flagge gesetzt, so wird die freie Fläche gleichmäßig unter diesen aufgeteilt. In dem folgenden Beispiel wird eine editierbare Textfläche mit der EXPAND-Flagge erzeugt:

```
root=plframe(0, EXPAND);
pllabel(root, PACKN|FILLX, "edit text");
plscrollbar(root, PACKW|FILLY);
pledit(root, PACKN|EXPAND, Pt(0,0),
    L"Plan 9 from outer Space", 23, 0);
```

Dieses hat folgendes Ergebnis:



Im obigen Quelltext irritiert auf den ersten Blick die Zeile

```
pledit(root, PACKN|EXPAND, Pt(0,0),
    L"Plan 9 from outer Space", 23, 0);
```

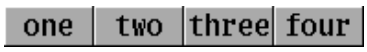
Was soll das L vor der String-Konstanten? Das L konvertiert den String nach *Rune ** und ist eine Besonderheit des Plan-9-Compilers. Ein genaue Beschreibung entnehmen Sie bitte der Dokumentation über den Plan 9 C Compiler: Rob Pike, How to Use the Plan 9 C Compiler, Plan 9 - The Documents, S. 57.

Wichtig ist, daß auch das *root*-Panel mit der EXPAND-Flagge versehen ist. Ansonsten würde das Text-Objekt zwar das *root*-Panel komplett ausfüllen, dieses würde aber weiterhin so klein wie möglich sein. Wird das *root*-Panel mit der EXPAND-Flagge versehen, so wird es die im zweiten Argument zu *plpack* übergebene Fläche ausfüllen.

Der vierte Weg ist die Benutzung der MAXX- und MAXY-Flaggen. Mit diesen Flaggen versehen wird die Breite oder Höhe eines Panels auf die jeweils größte Breite und Höhe seiner «Geschwister» gesetzt:

```
root=plgroup(0, 0);
plbutton(root, PACKW|MAXX, "one", 0);
plbutton(root, PACKW|MAXX, "two", 0);
plbutton(root, PACKW|MAXX, "three", 0);
plbutton(root, PACKW|MAXX, "four", 0);
```

Dadurch bekommen alle Knöpfe die gleiche Breite:



12.3 Der öffentliche Teil der Panel-Struktur

Die *Panel*-Struktur besteht aus einer Vielzahl von Komponenten. Die meisten sind zur internen Verwendung. Einige aber können und sollen von einem Anwender benutzt werden. Sie bilden den öffentlichen Teil der Struktur:

```
typedef struct Panel Panel;
struct Panel{
    Point ipad, pad;     /* interner und externer Freiraum */
    Point fixedsize;     /* Groesse von FIXED Panels       */
    int user;            /* zur freien Benutzung           */
    void * userp;        /* zur freien Benutzung           */
    Rectangle r;         /* wird von plpack gesetzt        */
    ...                  /* der private Teil               */
};
```

Die Komponenten *r*, *pad*, *ipad* und *fixedsize* wurden bereits erklärt. Die *user*- und *userp*-Felder werden von der Panel-Bibliothek nicht benutzt und stehen somit dem Anwender zur freien Verfügung. So kann z.B. *user* zur Identifizierung verschiedener Panels dienen, indem für jedes Panel eine andere Nummer in *user* hinterlegt wird. Bei einem graphischen Taschenrechner könnte in *user* pro Knopf ein Wert wie »0«, »1« oder »+« hinterlegt werden. *userp* könnte z. B. auf eine Funktion oder ein Objekt zur Eventbehandlung verweisen, welche dann später die Information aus *user* verwendet.

12.4 Die Panel-Grundfunktionen

```
int plinit(int ldepth);
```

Die Funktion *plinit* initialisiert die Panel-Bibliothek. Sie muß vor jeder anderen Panel-Funktion aufgerufen werden. Das einzige Argument ist normalerweise die Bildschirmtiefe. Die aktuelle Bildschirmtiefe wird beim Aufruf von *binit* in der Komponente *ldepth* der globalen Variablen *screen* hinterlegt. So benutzt die Panel-Bibliothek je nach Bildschirmtiefe vier Graustufen oder nur Schwarz und Weiß zum Zeichnen der verschiedenen Objekte.

```
int plpack(Panel * root, Rectangle space);
```

Nach jeder Änderung, die einen Einfluß auf das Layout der Applikation hat, sollte *plpack* aufgerufen werden. Daher muß die Funktion auf jeden Fall nach dem Erzeugen des Panel-Baums und innerhalb von *ereshaped* aktiviert werden. Es ist möglich, *plpack* auf einen Teilbaum des Panel-Baums anzuwenden, so z.B., wenn nur innerhalb dieses Teilbaums Änderungen stattgefunden haben. In diesem Fall sollte *plpack* mit der momentanen Fläche des Teilbaums zum Neupacken aufgerufen werden.

```
void pldraw(Panel * p,Bitmap * b);
```

Pldraw zeichnet rekursiv das Panel, auf das *p* zeigt, in die Bitmap *b*, normalerweise *&screen*. *Pldraw* muß nicht immer zum Zeichnen des gesamten Panel-Baums verwendet werden. Hat sich in einem Teilbaum etwas geändert, beispielsweise, wenn einem *pllabel*-Panel durch *plinitlabel* ein neuer Text zugewiesen wurde, so braucht nur dieser Teilbaum neu gezeichnet zu werden — *pldraw(subPanel, &screen);* —.

```
void plmove(Panel * p,Point min);
```

Ein bereits mit *plpack* gepackter Panel-Baum kann mit Hilfe von *plmove* verschoben werden. Nach dem Aufruf der Funktion ist die linke obere Ecke des Baums an der Position *min*.

```
void plfree(Panel * p);
```

Plfree gibt allen zum Panel *p* und allen zu seinen Nachfolgern gehörenden Speicher frei.

12.5 Maus, Tastatur und ein typischer Aufbau

Eine Applikation hat prinzipiell folgenden Aufbau:

```
#include <u.h>
#include <libc.h>
#include <libg.h>
#include <panel.h>

Panel *root;
Panel *mkpanels(void);

void eventloop(void);

void main(void){
    binit(0, 0, 0);
    einit(Emouse|Ekeyboard);
    plinit(screen.ldepth);
```

```
    root=mkpanels();    /* muss den Baum erzeugen */
    ereshaped(screen.r);
    eventloop();
}
void eventloop(void){
    Event e;
    for(;;){
        switch(event(&e)){
        case Ekeyboard:
            plkeyboard(e.kbdc);
            break;
        case Emouse:
            plmouse(root, e.mouse);
            break;
        }
    }
}
void ereshaped(Rectangle r){
    screen.r=r;
    plpack(root, r);
    bitblt(&screen, r.min, &screen, r, Zero);
    pldraw(root, &screen);
}
```

Die fehlende Funktion *mkpanels* muß vom Anwender implementiert werden. Ihre Aufgabe ist es, den Panel-Baum zu erzeugen und einen Zeiger auf das *root*-Panel zurückzuliefern. Im Hauptprogramm werden alle notwendigen Initialisierungsfunktionen aufgerufen, mit Hilfe von *mkpanels* und *ereshaped* der Panel-Baum erzeugt und dargestellt sowie abschließend die Funktion *eventloop* aufgerufen. Diese Funktion verarbeitet eintreffende Maus- und Tastatur-Events, indem sie diese an *plmouse* oder *plkeyboard* weiterreicht.

12.6 Die verschiedenen Panel-Typen

Bislang wurden in einigen Beispielen die Panel-Typen *plbutton*, *plframe*, *plgroup*, *pllabel* usw. vorgestellt. In diesem Abschnitt werden nun alle Panel-Typen kurz aufgeführt und erklärt.

Knöpfe

```
Panel *plbutton(Panel *parent, int flags,
    Icon *label, void (*hit)(Panel *, int));
Panel *plcheckbutton(Panel *parent, int flags,
    Icon *label, void (*hit)(Panel *, int, int));
Panel *plradiobutton(Panel *parent, int flags,
    Icon *label, void (*hit)(Panel *, int, int));
void plsetbutton(Panel *panel, int value);
Panel *plmenu(Panel *parent, int flags, Icon **items,
    int itemflags, void (*hit)(int, int));
```

Der Typ *Icon** ist ein Synonym für *void**. Das aktuelle Argument kann dabei vom Typ *char** oder vom Typ *Bitmap** sein. Im zweiten Fall muß die Flagge BITMAP in *flags* gesetzt sein.

Plbutton erzeugt ein Knopf-Panel mit der Aufschrift *label*. Ist *hit* nicht Null, so wird diese Funktion aufgerufen, wenn man den Knopf nach einem Drücken losläßt. *Hit* bekommt als erstes Argument einen Zeiger auf das zugehörige Panel, als zweites den Knopf der Maus, mit dem das Panel gedrückt worden ist.

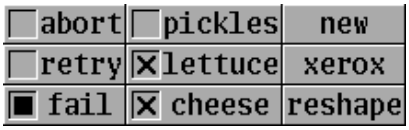
Auch die von *plcheckbutton* und *plradiobutton* erzeugten Panels sind Knöpfe mit der in *label* angegebenen Aufschrift. Drückt man diese, so bleiben sie im gedrückten Zustand und visualisieren dies durch eine Markierung. Durch ein erneutes Betätigen kehren diese Panels in den Ausgangszustand zurück. Durch *plsetbutton* kann der Zustand eines Radio- oder Check-Knopfs gesetzt werden. Dabei steht 1 für »gedrückt« und 0 für »nicht gedrückt«. Wechselt ein Radio- oder Check-Knopf seinen Zustand, so wird die Callback-Funktion *hit* aufgerufen. *Hit* bekommt als erstes Argument einen Zeiger auf das zugehörige Panel, als zweites den Maus-Knopf und als drittes den neuen Zustand des Panels. Versetzt man einen Radio-Knopf in den gedrückten Zustand, so werden eventuell alle Geschwister-Panels des gleichen Typs zurückgesetzt, ohne deren *hit*-Funktionen aufzurufen. Es kann nämlich innerhalb einer Gruppe nur ein Radio-Knopf gedrückt sein.

Die *plmenu*-Funktion erzeugt eine Matrix von Knöpfen (siehe nachfolgendes Beispiel). *Items* ist ein null-terminierter Vektor von *Icon*-Zeigern, der die Aufschriften der einzelnen Knöpfe festlegt. *Itemflag* ist die Flagge, die beim Erzeugen der verschiedenen Knöpfe benutzt wird. Jedesmal, wenn man einen der so entstandenen Menu-Knöpfe betätigt, wird die Funktion *hit* aufgerufen, und zwar mit dem verursachenden Maus-Knopf als erstem und dem Index des gedrückten Knopfs in *items* als zweitem Argument.

Der folgende Quelltext erzeugt eine Liste von Radio- und Check-Knöpfen sowie ein Menu-Panel:

```
Panel *g, *p;
char  *buttons[4];

root=plgroup(0,0);
g=plgroup(root, PACKW);
plradiobutton(g, PACKN|FILLX, "abort", 0);
plradiobutton(g, PACKN|FILLX, "retry", 0);
p=plradiobutton(g, PACKN|FILLX, "fail", 0);
plsetbutton(p, 1);
g=plgroup(root, PACKW);
plcheckbutton(g, PACKN|FILLX, "pickles", 0);
p=plcheckbutton(g, PACKN|FILLX, "lettuce", 0);
plsetbutton(p, 1);
p=plcheckbutton(g, PACKN|FILLX, "cheese", 0);
plsetbutton(p, 1);
buttons[0]="new";
buttons[1]="xerox";
buttons[2]="reshape";
buttons[3]=0;
plmenu(root, PACKW, buttons, PACKN|FILLX, 0);
```

Das Resultat sieht folgendermaßen aus:



Label- und Message-Panels

```
Panel *pllabel(Panel *parent, int flags, Icon *label);

Panel *plmessage(Panel *parent, int flags, int width,
    char *text);
```

Ein Label-Panel stellt den als Argument *label* übergebenen Text in einer Zeile dar. Daher sollte der Text kurz sein. Ist ein längerer Text darzustellen, muß man Message-Panels benutzen. Diese brechen den Text zwischen den Wörtern um. Das *width*-Argument gibt die gewünschte Panelbreite in Pixel an. Wenn der Text Wörter enthält, die breiter als *width* sind, wird das Panel so breit wie das längste dieser Wörter.

Plgroup- und Plframe-Panels

```
Panel *plgroup(Panel *parent, int flags);

Panel *plframe(Panel *parent, int flags);
```

Die Funktionen *plgroup* und *plframe* dienen der Gruppierung und Anordnung von Panels. Möchte man z.B. zwei unabhängige Mengen von Radio-Knöpfen haben, so müssen diese in zwei Gruppen aufgeteilt werden. Der einzige Unterschied zwischen *plgroup* und *plframe* ist, daß ein *plframe*-Panel einen Rahmen um seine Kinder zeichnet.

Canvas-Panel

```
Panel *plcanvas(Panel *parent, int flags,
    void (*draw)(Panel *), void (*hit)(Panel *, Mouse *));
```

Ein *canvas*-Panel ist eine leere Fläche, die dem Anwender zur freien Benutzung zur Verfügung gestellt wird. Er kann in ihr z.B. Bilder darstellen oder Linien, Kreise usw. zeichnen. Bei jedem Aufruf von *pldraw* wird die im *draw*-Argument angegebene Funktion zum Zeichnen des Canvas-Panels aufgerufen. Gibt man eine *hit*-Funktion an, wird diese bei jeder Maus-Aktivität innerhalb des Canvas-Panels aufgerufen. Dabei liegen die Maus-Koordinaten immer innerhalb der rechteckigen Fläche des Panels, oder die Maus ist gerade aus dem Panel hinausbewegt worden. In diesem Fall ist das OUT-Bit im *buttons*-Feld der *Mouse*-Struktur gesetzt. Zu beachten ist, daß, solange *ipad* Null ist oder nicht eine der Flaggen EXPAND oder FIXED gesetzt ist, die Größe des Canvas-Panels Null ist.

Wie man innerhalb eines Canvas-Panels zeichnet, zeigt ein Auszug aus dem Beispielprogramm *ball.c*:

```
Panel *root, *can;
Bitmap * canB;
int diffx,diffy;
Point pos,old;
int radius=15;

void drawCan(Panel *p){
    ...
    circle(canB,pos,radius,~0,DxorS);
    bitblt(&screen, can->r.min,canB,
                Rect(0,0,diffx,diffy),S);
    ...
    return;
}

void ereshaped(Rectangle r){
    plpack(root, r);
    diffx=can->r.max.x-can->r.min.x;
    diffy=can->r.max.y-can->r.min.y;
    if(canB)
        bfree(canB);
    if((canB = balloc(Rect(0,0,diffx,diffy),
                screen.ldepth)) == 0)
    exits("balloc");
    ...
    pldraw(root, &screen);
}

Panel * mkpanels(void)
{
    ...
    root=plframe(0, EXPAND);
    ...
    can=plcanvas(plframe(root, PACKW|EXPAND),
                EXPAND,drawCan,0);
    ...
    return root;
}
```

Die Funktion *circle*, siehe *bitblt*(2), zeichnet einen Kreis in die Bitmap *screen* und *balloc*(2) — **B**itmap **alloc** — allokiert Speicher für eine Bitmap. Das Gegenstück zu *balloc* ist *bfree*(2).

Slider-Panel

```
Panel *plslider(Panel *parent, int flags, Point size,
    void (*hit)(Panel *p, int but, int val, int len));

void plsetslider(Panel *p, int val, int range);
```

Ein *plslider*-Panel ist ein Schieberegler, der mit der Maus verstellt werden kann. *Size* ist die gewünschte Größe, welche aber aufgrund des Setzens von Flaggen, wie z.B. EXPAND, von *plpack* anders berechnet werden kann. Das Slider-Panel ist horizontal orientiert, wenn die *x*-Komponente in *size* größer ist als die *y*-Komponen-

te. Ansonsten ist die Ausrichtung vertikal. Gibt man eine *hit*-Funktion an, wird diese jedesmal aufgerufen, wenn sich die Stellung des Reglers ändert. Die Argumente zu *hit* sind das zugehörige Panel, der verursachende Maus-Knopf, der neue Wert und der maximale Wert in Pixel.

Durch *plsetslider* kann die Stellung eines Slider-Panels *p* gesetzt werden. Diese wird bei angenommener maximaler Stellung *range* auf *val* gesetzt. So setzen

```
plsetslider(s,1,2);
plsetsilder(s,12,24);
```

und

```
plsetslider(s,50,100);
```

das Slider-Panel auf die Mittelstellung.

Im folgenden ist der Quelltext zum Erzeugen einer Applikation dargestellt, die ein Slider-Panel und zwei Label-Panels beinhaltet. Eines der Label-Panels wird in der *hit*-Funktion mit einem Text gefüllt, der die momentane Stellung des Reglers reflektiert:

```
Panel *sliderlabel;
char sliderval[]=" 57/100";

void hit(Panel *p, int buttons, int val, int range){
    USED(p, buttons);
    sprint(sliderval, "%3d/100", val*100/range);
    plinitlabel(sliderlabel, PACKE, sliderval);
    pldraw(sliderlabel, &screen);
}

void slider(void){
    Panel *sl;
    root=plframe(0, 0);
    pllabel(root, PACKN|PLACEE, "aktueller Wert");
    sl=plslider(root, PACKW|FILLY, Pt(137, 0), hit);
    plsetslider(sl, 57, 100);
    sliderlabel=pllabel(root, PACKE, sliderval);
}
```

Entry-Panel

```
Panel *plentry(Panel *parent, int flags, int width,
    char *init, void (*hit)(Panel *p, char *text));

char *plentryval(Panel *);
```

Ein *plentry*-Panel stellt eine Textzeile dar, die dann vom Anwender editiert werden kann. Mit den *width* und *init* Argumenten werden die Breite des Panels und der initiale Text spezifiziert. Klickt man das Entry-Panel mit der Maus an, wird es zum für

Tastatur-Aktivitäten zuständigen Panel. Gibt man ein Zeichen an der Tastatur ein, wird dieses am Textende angefügt. Ein Control-U löscht den gesamten Text, ein Control-W löscht ein Wort, und ein Backspace löscht das letzte Zeichen. Ein Newline beendet die Eingabe, und die *hit*-Funktion wird aufgerufen. Diese bekommt über das *text*-Argument den eingegebenen Text ohne das Newline-Zeichen.

Der aktuelle Text kann mit *plentryval* abgefragt werden. Es folgt der Quelltext zu einem Beispiel:

```
root=plgroup(0,0);
pllabel(root, PACKW, "Eingabe: ");
plentry(root, PACKW, 140, "Hallo euch allen!", 0);
```

Das Ergebnis sieht folgendermaßen aus:

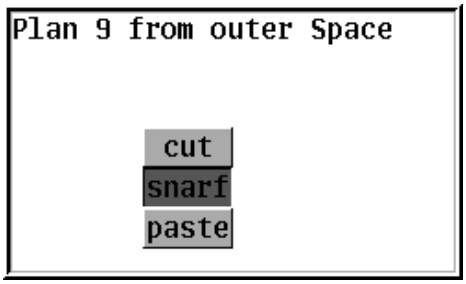
Pop-up-Panel

```
Panel *plpopup(Panel *parent, int flags,
    Panel *but1, Panel *but2, Panel *but3);
```

Empfängt ein *plpopup*-Panel einen Maus-Event, der einen gedrückten Maus-Knopf beinhaltet, so stellt es, je nachdem welcher Knopf der Maus gedrückt worden ist, den in *but1*, *but2* oder *but3* angegebenen zugehörigen Panel-Baum dar. Alle weiteren Maus-Aktivitäten werden dann an den dargestellten Panel-Baum weitergereicht. Ist eines dieser drei Argumente 0, werden die zugehörigen Maus-Events an die Kinder des Popup-Panels weitergereicht. Im nachfolgenden Beispiel wird eine Applikation mit einem Text- und zwei Popup-Menüs erzeugt. Bei Betätigung des mittleren oder rechten Maus-Knopfs wird jeweils ein *plmenu*-Panel dargestellt. Maus-Events mit der linken Maus-Taste werden an das *pledit*-Panel weitergereicht:

```
char *menu2[]={ "cut", "snarf", "paste", 0 };
char *menu3[]={ "read", "write", "exit", 0 };

void popup(void){
    Panel *m2, *m3, *pop;
    m2=plmenu(0, 0, menu2, PACKN|FILLX, 0);
    m3=plmenu(0, 0, menu3, PACKN|FILLX, 0);
    root=plframe(0, EXPAND);
    pop=plpopup(root, EXPAND, 0, m2, m3);
    pledit(pop, EXPAND, Pt(0,0),
        L"Plan9 from outer Space", 23, 0);
}
```

Wenn die mittlere Maus-Taste gedrückt wird, dann hat der Bildschirm folgenden Inhalt:



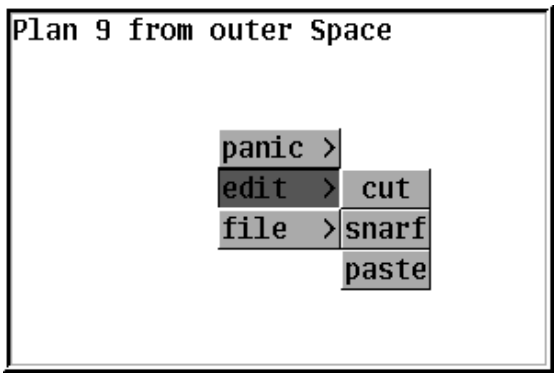
Pull-down- und Menü-Panel

```
Panel *plpulldown(Panel *parent, int flags,
    Icon *label, Panel *pull, int side);

Panel *plmenubar(Panel *parent, int flags, int itemflags,
    Icon *label1, Panel *pull1, Icon *label2, ...);
```

Ein *plpulldown*-Panel sieht wie ein normaler Knopf mit der Aufschrift *label* aus. Drückt man den Knopf, stellt das Menü den in *pull* angegeben Panel-Baum dar und reicht alle weiteren Maus-Events an *pull* weiter. Das *side*-Argument muß PACKN, PACKE, PACKS oder PACKW sein. Auf dieser Seite des Pull-down-Panels wird *pull* dargestellt. Das folgende Beispiel beinhaltet ein Pop-Up-Panel als Hauptmenü, das aus drei Pull-down-Panels besteht. Diese stellen den *pull*-Panel-Baum nach rechts dar:

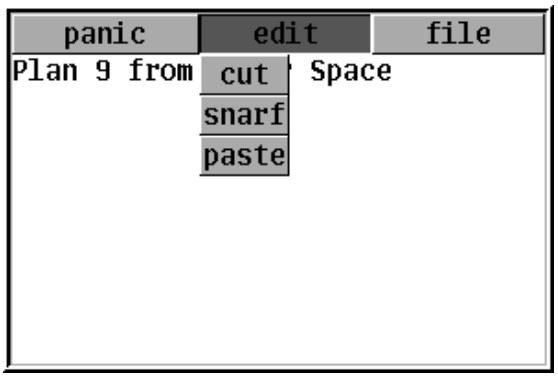
```
char *menu1[]={ "abort", "retry", "fail", 0 };
char *menu2[]={ "cut", "snarf", "paste", 0 };
char *menu3[]={ "read", "write", "exit", 0 };

void cascade(void){
    Panel *m1, *m2, *m3, *menu, *pop;
    m1=plmenu(0, 0, menu1, PACKN|FILLX, 0);
    m2=plmenu(0, 0, menu2, PACKN|FILLX, 0);
    m3=plmenu(0, 0, menu3, PACKN|FILLX, 0);
    menu=plgroup(0,0);
    plpulldown(menu, PACKN|FILLX, "panic >", m1, PACKE);
    plpulldown(menu, PACKN|FILLX, "edit  >", m2, PACKE);
    plpulldown(menu, PACKN|FILLX, "file  >", m3, PACKE);
    root=plframe(0, EXPAND);
    pop=plpopup(root, EXPAND, 0, menu, 0);
    pledit(pop, EXPAND, Pt(0,0),
        L"Plan 9 from outer Space", 23, 0);
}
```

Wenn im Hauptmenü der mittlere Eintrag ausgewählt wird, dann sieht das Resultat folgendermaßen aus:



Ein *plmenubar*-Panel ist eine Menüleiste von Pull-down-Panels. *Itemflag* ist die Flagge zum Erzeugen der einzelnen Knöpfe. Danach folgt eine null-terminierte Liste von *label*- und *pull*-Argumenten. Dabei spezifiziert *label* die Aufschrift des aktuellen Knopfs und *pull* den bei Knopfdruck zu aktivierenden Panel-Baum. Das obige Beispiel mit Pull-down-Menüs nach unten läßt sich dann einfacher schreiben:

```
char *menu1[]={ "abort", "retry", "fail", 0 };
char *menu2[]={ "cut", "snarf", "paste", 0 };
char *menu3[]={ "read", "write", "exit", 0 };

void mbar(void){
    Panel *m1, *m2, *m3;
    m1=plmenu(0, 0, menu1, PACKN|FILLX, 0);
    m2=plmenu(0, 0, menu2, PACKN|FILLX, 0);
    m3=plmenu(0, 0, menu3, PACKN|FILLX, 0);
    root=plframe(0, EXPAND);
    plmenubar(root, PACKN|FILLX, PACKW|EXPAND,
        "panic", m1, "edit", m2, "file", m3, 0);
    pledit(root, EXPAND, Pt(0,0),
        L"Plan 9 from outer Space", 23, 0);
}
```



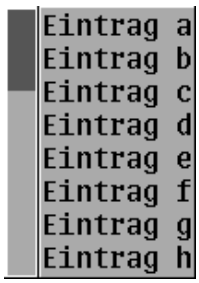
Scrollbar- und List-Panels

```
Panel *pllist(Panel *parent, int flags,
    char *(*generate)(int index), int nlist,
    void(*hit)(Panel *p, int buttons, int index));

Panel *plscrollbar(Panel *parent, int flags);

void plscroll(Panel *scrollee,
    Panel *xscroller, Panel *yscroller);
```

Ein *pllist*-Panel zeigt eine Liste von Textzeilen. Wird die *generate*-Funktion mit der Zahl *n* als Argument aufgerufen, so muß sie einen Zeiger auf die *n*-te Zeile zurückliefern. Die Indizierung beginnt dabei bei 0. Ist *n* außerhalb des zulässigen Bereichs, so muß ein Nullzeiger geliefert werden. *Hit* wird aufgerufen, wenn man eine der dargestellten Zeilen mit der Maus auswählt. Dessen Argumente sind ein Zeiger auf das List-Panel, die verursachende Maus-Taste und der Index der ausgewählten Zeile. Die minimale Anzahl dargestellter Zeilen ist *nlist*. Die Liste kann gescrollt werden, indem man ein *plscrollbar*-Panel erzeugt und dieses durch *plscroll* mit der Liste assoziiert. Die Argumente zu *plscroll* sind das Panel, das gescrollt werden soll, das *plscrollbar*-Panel für die vertikale Richtung und eines für die horizontale Richtung. Es muß nur eines der beiden letzten Argumente ungleich 0 sein. Zur Zeit gibt es allerdings noch kein Panel, welches in horizontaler Richtung scrollen kann. Daher ist *xscroller* normalerweise 0. In dem folgenden Beispiel wird eine Liste mit einem *plscrollbar*-Panel verbunden:

```
char *genlist(Panel * dummy,int which){
    static char buf[16];

    USED(dummy);
    if(which<0 || 26<=which) return 0;
    sprint(buf, "Eintrag %c", which+'a');
    return buf;
}

void scroll(void){
    Panel *l, *s;
    root=plgroup(0, 0);
    l=pllist(root, PACKE, genlist, 8, 0);
    s=plscrollbar(root, PACKW|FILLY);
    plscroll(l, 0, s);
}
```

Und so sieht das Ergebnis aus:



Text-Panels

```
Panel *pledit(Panel *parent, int flags, Point minsize,
    Rune *text, int ntext, void (*hit)(Panel *));

Panel *pltextview(Panel *parent, int flags, Point minsize,
    Rtext *text, void (*hit)(Panel *p, int buttons,
    Rtext *t));
```

Es gibt zwei Text-Panels. Ein *pledit*-Panel stellt eine editierbare Textfläche mit nur einer Schriftart dar. Man kann scrollen, markieren und natürlich tippen. Ein *pltextview*-Panel dagegen ist wesentlich mächtiger. Es kann Text in mehreren Schriftarten, Graphik usw. beinhalten. Eine ausführliche Beschreibung der beiden Panels und der zugehörigen Hilfsfunktionen würde an dieser Stelle zu weit führen. Bei Interesse ist ein Blick in die entsprechenden Manual-Seiten und in die Dokumentation zu empfehlen.

12.7 Reinitialisierungs-Funktionen

Zu jedem Paneltyp gibt es eine zugehörige Funktion, um die einzelnen Attribute zu ändern, wie z.B. die Aufschrift oder die Callback-Funktion eines Knopfs. Der Name der Funktion entspricht dem Namen der Funktion zum Kreieren des Panels, wobei das Präfix *pl* durch *plinit* ersetzt ist, so z.B. *plinitmessage* anstelle von *plmessage*. Die Argumente und deren Bedeutung sind bis auf das erste Argument gleich. Dieses ist nun nicht mehr ein Zeiger auf das Panel, dem das neue Panel untergeordnet sein soll, sondern das Panel, dessen Attribute geändert werden sollen.

Leider gibt es keine zu *plmenubar* korrespondierende *plinit*-Funktion, und der Typ eines Panels kann nicht durch Aufruf einer *plinit*-Funktion anderen Typs geändert werden.

12.8 Was blieb unerwähnt?

Bei dem Gebrauch der Panel-Bibliothek geschieht nichts automatisch. Kein Dämon bewacht den Zustand der Panels, um eventuell aufgrund von Änderungen das Layout des Panel-Baums neu zu berechnen oder den Baum neu zu zeichnen. Der Entwickler muß diese Arbeit durch Aufruf von *plpack* und *pldraw* selbst verrichten.

Die Panel-Bibliothek wurde zur Programmierung des Plan 9 »World Wide Web«-Browsers *mothra* entwickelt und ist nach Aussage der Entwickler noch zu jung, um fehlerfrei zu sein. (Anmerkung: Dieses kann von uns bestätigt werden. Allerdings befindet sich eine Vielzahl von Updates auf dem ftp-Sever der Bell Labs. Dort findet man unter anderem neben einer stabileren *mothra*-Version auch ein Update der Panel-Bibliothek.)

Normalerweise arbeitet die Panel-Bibliothek mit dem Fenster, in dem die Applikation aufgerufen wird. Wie man ein neues Fenster unter 8½ erzeugt und dort arbeitet, zeigt das nachfolgende Beispiel. In dem Hauptprogramm wird der eigentliche Prozeß als Waise angelegt, um die Shell, von der aus das Programm gestartet wurde, nicht zu blockieren. Der restliche Quelltext ist bis auf den Aufruf der Funktion *init_win* ein bereits bekanntes Beispiel aus diesem Abschnitt:

```
Panel *root;

void error(char * s)
{   fprint(2,"%s\n",s);
    exits(s);
}

void main(void)
{   switch(rfork(RFNAMEG|RFFDG|RFPROC)){
    case -1:
        error("fork");
        exits(0);
    case 0:
        init_win(0,0,200,200);
        binit(0, 0, 0);
        einit(Emouse);
        plinit(screen.ldepth);
        root=plframe(0, 0);
        pllabel(root, 0, "Hello, world!");
        plbutton(root, 0, "done", done);
        ereshaped(screen.r);
        for(;;) plmouse(root, emouse());
    }
    exits(0);
}
```

Die Funktion *init_win* bekommt als Argumente die gewünschte Position und Größe des zu erzeugenden Fensters übergeben. Zuerst wird die öffentliche Seite der Kommunikationspipe zu dem User-Server 8½ geöffnet und durch einen Aufruf von *mount*(2) das gewünschte Fenster erzeugt. Abschließend werden die von 8½ zur Verfügung gestellten virtuellen Dateien, wie z.B. *cons* oder *mouse*, an die richtige Stelle im Dateisystem gebunden und die Standardfildeskriptoren reinitialisiert:

```
void init_win(int x1,int y1,int x2,int y2)
{   int fd;
    char buf[256];
    char * env;

    env=getenv("8½srv");
    if(!env)
        error("no 8½");
    fd=open(env,ORDWR);                    /* open 8½ server*/
    if(fd<0)
        error("open 8½ failed");
    sprint(buf,"N%d %d %d %d %d",getpid(),x1,y1,x2,y2);
    if(mount(fd,"/mnt/8½",MREPL,buf)<0)/* create window */
        perror("mount output window failed");
    if(bind("/mnt/8½", "/dev", MBEFORE) < 0)
        error("win bind");
    close(0);
    close(1);
```

```
        if(open("/dev/cons", OREAD) != 0)
            error("/dev/cons 0 client open");
        if(open("/dev/cons", OWRITE) != 1)
            error("/dev/cons 1 client open");
        dup(1, 2);
    }
```

12.9 Zusammenfasssung

Die Panel-Bibliothek verbirgt viel der lästigen und umfangreichen Arbeit zur Entwicklung von graphischen und interaktiven Benutzeroberflächen. Die Modellierung und Benutzung der Bibliothek erinnern teilweise stark an *TCL/Tk*, außer, daß die Programmiersprache C ist und daß die Hintergrundoperationen von *TCL/TK* nicht automatisch stattfinden.

13
Prozesse

13.1 Prozesse unter Plan 9

Die Ausführung eines Kommandos bewirkt unter Plan 9 wie unter Unix normalerweise die Erzeugung eines neuen Prozesses. Unter Unix spiegelt sich die Existenz eines Prozesses außerhalb des Kerns in einer Prozeß-Id wider, unter Plan 9 in Dateien im Dateisystem.

Wenn man unter Unix keine expliziten Verbindungen mit Pipes, IPC oder Sockets vorbereitet, kann man mit einem Prozeß nur noch via *kill(1,2)* und einer Signalnummer verhandeln. Der Prozeß reagiert darauf mit seinem Ableben oder aktiviert eine entsprechende Funktion. Diese Art der Kommunikation hat nur wenige Wörter zur Verfügung — unter SVR4 und 4.3+BSD: die Signalnummern 0 bis 31 /Ste92/ — und ist demzufolge etwas primitiv. Dämonen wie *init* und *inetd* verwenden diese Form der Kommunikation.

In Unix wurden Prozesse durch Strukturen im Kern beschrieben, die nur über Systemaufrufe beeinflußt werden konnten. Für Linux gibt es ein Dateisystem */proc/*, über das die Prozeßinformation erreichbar ist. In Plan 9 werden die Daten eines Prozesses durch den *proc device server* »#p« in Form von Dateien offengelegt. Die Manipulation von Prozessen erfolgt mittels Schreib- und Leseoperationen auf diese Dateien. In aller Regel wird dieser Server auf */proc* durch *bind '#p' /proc* gebunden. Anzumerken ist, daß die Funktionalitäten Prozeßsteuerung und Prozeßkommunikation strikt voneinander getrennt sind. Für einen Prozeß mit der Prozeß-Id *pid* wird ein Katalog */proc/pid* mit folgenden Dateien angelegt:

ctl Durch spezielle Nachrichten, welche in aller Regel via *echo* in die Datei geschrieben werden, wird der Prozeß kontrolliert. Die wesentlichen Nachrichten sind:
stop: Die Ausführung des Prozesses wird angehalten.
start: Ein Prozeß, der sich im *stopped*-Zustand befindet, läuft weiter.
kill: Eliminiert den Prozeß.

mem Repräsentiert das aktuelle Memory-Image des Prozesses. Die Datei kann gelesen und, wenn sich der Prozeß im *stopped*-Zustand befindet, auch geschrieben werden.

note Texte, welche in diese Datei geschrieben werden, werden dem Prozeß direkt zugestellt. Der Prozeß muß, ähnlich wie unter Unix bei Signalen, Vorkehrungen treffen, um das Eintreffen einer Nachricht zu überleben.

notepg Die Nachricht wird an jedes Mitglied der Prozeßgruppe geschickt, zu der der Prozeß mit der Prozeß-Id *pid* gehört.

proc Diese Datei erlaubt Lesezugriff auf die Datenstruktur, die im Kern den Prozeß beschreibt.

status Diese nur lesbare ASCII-Datei enthält die Status-Information des Prozesses: den Namen des ausgeführten Programms, und Benutzers, den Prozeßzustand (*Open, Read, Sleep, Idle* ...), die verbrauchte CPU-Zeit und den Speicherverbrauch.

text Diese Datei repräsentiert die Datei, durch deren Aufruf der Prozeß erzeugt worden ist. Für einen *debugger* ist sie u.a. wegen der Symboltabelle von Bedeutung. Führt man sie aus, ergibt sich eine weitere Instanz des Prozesses.

Weil die Information über einen Prozeß in einer Datei, */proc/pid/status*, in ASCII-Form vorliegt, könnte ein primitives ps-*Kommando für Plan 9 in awk implementiert werden. Das Kommando*

```
% awk '{ print $1 }' /proc/*/status | sed 5q
init
awk
cs
arpd
alarm
```

liefert die Programmnamen von fünf aktuellen Prozessen im aktuellen Namensraum. Es ist offensichtlich, daß die Informationsbeschaffung über den Zustand von Prozessen im Vergleich zu Unix wesentlich vereinfacht wurde.

Wir wollen uns etwas mit dem Umgang von Prozessen vertraut machen. Unser erster Prozeß soll einen Text ausgeben und sich dann schlafen legen. Die C-Quelle, *hw.c*, sieht

```
/* hw.c                                                          */

#include <u.h>
#include <libc.h>

void main(void)
{   print("Hello World (pid = %d)\n", getpid() );
    print("Chrr chrr ....\n");
    sleep(1000 * 100 );          /* 100 Sek. schlafen        */
    exits("bye");                /* status=exits pid:bye     */
}
```

bis auf *exits("bye")* wie erwartet aus. Durch den Aufruf von *exits()* wird der Prozeß beendet und ein Text der Länge *ERRLEN - 1* (63) Zeichen in */proc/pid/status* hinterlegt. Dieser Text wird mit einem Präfix versehen und kann vom Vater-Prozeß mittels *wait()* abgeholt werden. Nach Konvention gilt ein leerer Exit-Text als Indikator, daß das Kommando ohne Fehler abgearbeitet werden konnte. Das Präfix enthält den Namen des ausgeführten Programms und die Prozeß-Id.

Schickt man in die *ctl*-Datei des Prozesses die richtigen Wörter, kann man den Prozeßzustand manipulieren.

Der Prozeß wird in den Hintergrund geschickt.

```
% hw &
% Hello World (pid = 667)
Chrr chrr ....
```

Mit *ps* bzw. *cat* kann man den Zustand des Prozesses erfragen.

```
% ps | grep hw
none        667    0:00    0:00    36K Sleep    hw
% cat /proc/667/status
hw none Sleep 0 0 18600 0 0 0 36 5
```

Mit *stop* erfragt man den Befehl, mit dem der Prozeß angehalten werden kann. Die Ausgabe von *stop* kann direkt als Eingabe für *rc* verwendet werden.

```
% stop hw
echo stop>/proc/667/ctl
% stop hw | rc
%  ps -aux | grep hw
none 667 0:00 0:00 36K Stopped  hw
```

kill liefert den Text, mit dem der Prozeß zur Terminierung gezwungen werden kann.

```
% kill hw
echo kill>/proc/270/note
% kill hw | rc
% echo $status

%
```

Ein Prozeß kann nur zur Terminierung gezwungen werden, falls er rechenbereit ist, das heißt, hier muß der *stopped*-Zustand verlassen werden, um den Prozeß zur Terminierung zwingen zu können. Das erreicht man mit:

```
% start hw
echo start>/proc/270/ctl
% echo start>/proc/270/ctl
%  ps -aux | grep hw
%
```

Weil die komplette Information über einen Prozeß in lesbaren Dateien abgelegt ist, kann man zur Laufzeit eines Programms mittels geeigneter Werkzeuge, wie z.B. *db* oder *acid*, Variablenwerte des Prozesses setzen bzw. auslesen.

Terminiert ein Prozeß aufgrund eines unerwarteten Ereignisses, wie z.B. Division durch Null oder durch Zugriff auf eine irreguläre Speicheradresse, bleibt der Prozeß in dem Zustand, der ihn zur Terminierung veranlaßte, mit *broken* als Zustand im *proc/pid/**. Ein Debugger hat es nun verhältnismäßig leicht, die Überreste des Prozesses zu analysieren, denn über Dateien wie */proc/pid/mem* und */proc/pid/text* hat er sehr einfach Zugriff zum Memory-Image und zur Symboltabelle des Programms. Das nachfolgende Programm terminiert beim Zugriff auf die Speicheradresse -1.

```
/* p_–1.c                                                    */

#include <u.h>
#include <libc.h>

void point_of_no_return(void)
{
    print("This is the point of no return %d\n",
            *((int *) –1) );
    exits("bye");                   /* status=exits pid:bye*/
}
```

```
void main(void)
{   print("Hello World (pid = %d)\n", getpid() );
    point_of_no_return();
}
```

Nach der Terminierung des Programms kann der Grund mit Hilfe des *debuggers* analysiert werden.

```
% p_-1
Hello World (pid = 400)
p_-1 400: suicide: sys: trap: unaligned address pc=0x102c
% ps -aux | grep p_-1
none        400    0:00   0:00    36K Broken   p_-1
% db 400             # 400 war die pid
                         # des zu untersuchenden Prozesses
sparc binary
last exception: unaligned address
p_-1.c:19 point_of_no_return+#c? MOVW   #ffffffff(R0), R8
$c                       # adb laesst gruessen
point_of_no_return() p_-1.c.c:17 called from main+#28 p_-1.c
.c:25
main() p_-1.c.c:23 called from _main+#14 main9.s:12
_main() at #6614
$q
% broke              # der Prozess sollte aus dem
echo kill>/proc/400/ctl  # Namensraum entfernt werden
% broke | rc         # so geht es besonders einfach
```

Plan 9s *debugger* erinnert sehr stark an Steve Bournes *adb* /Bo84/. Den Nutzer erwartet die gleiche Leistungsfähigkeit, allerdings ist die Nutzung, zumindest auf den ersten Blick, etwas archaisch.

13.2 Vermehrung

Mit *fork()* werden wie unter Unix Prozesse erzeugt, und mit *exec()* wird in einem neuen Prozeß ein neues Programm zur Ausführung gebracht. Der Prozeß kann mit *wait()* auf die Beendigung des Sohn-Prozesses warten und dadurch den *exit*-Text des Sohns erfahren. Die nachfolgende C-Quelle zeigt die Anwendung von *fork()* und *wait()* an einem Beispiel.

```
/* fork_and_wait.c                                          */

#include <u.h>
#include <libc.h>

void main(void)
{       Waitmsg msg;

        print("Daddy:  I'am here.\n");
        switch ( fork() ) {
    case -1:    print("fork failed");
                exits("fork failed");
```

```
    case 0:     print("\tSon: I'am here (pid = %d). \n",
                getpid());
                    exits("child is dead.");

    default:    print("Daddy: wait     = %d\n", wait(&msg) );
            print("Daddy: msg.msg  = %s\n", msg.msg );
            print("Daddy: msg.time = %s\n", msg.time );
            print("Daddy: msg.pid  = %s\n", msg.pid );
            exits(0);
        }
        exits("");
}
```

Falls kein Sohn-Prozeß mehr aktiv ist, kehrt *wait()* sofort zurück und liefert -1 als Resultat. Ist noch ein Sohn-Prozeß aktiv, wartet *wait()* auf desen Terminierung und liefert als Resultat die Prozeß-Id des zu Ende gegangenen Prozesses. Neben dem *exit*-Text kann der Vater-Prozeß noch die vom Sohn verbrauchte CPU-Zeit erfahren. Ein Ablauf des Programms:

```
% fork_and_wait
Daddy:  I'am here.
        Son: I'am here (pid = 128).
Daddy: wait     = 128
Daddy: msg.msg  = fork_and_wait 128:child is dead.
Daddy: msg.time =           0          40          60
Daddy: msg.pid  =         128
```

Soweit sehen Prozesse unter Plan 9 fast so aus wie Unix-Prozesse. Doch im Gegensatz zu Unix ist unter Plan 9 bis zu einem gewissen Grad wählbar, was bei einem *fork()*-Aufruf von Prozeß zu Prozeß kopiert und was gemeinsam genutzt, also gegenseitig verändert werden kann. Beim Aufruf von *rfork()* kann durch Angabe von Flaggen, wie z.B. *RFFDG* (*file*-Deskriptoren werden kopiert), festgelegt werden, womit der neue Prozeß ausgestattet wird. *fork()* ist auf *rfork(RFFDG|RFPROC)* abgebildet. *fork()* ist ein spezieller *rfork()*-Aufruf. Allgemein kontrolliert *rfork()* Erzeugung und Ressourcen der verschiedenen Flaggen, die in Abbschnitt 11.3 demonstriert werden. Zum Beispiel kann das Environment eines Prozesses kopiert oder gemeinsam genutzt werden. Ein Prozeß-Abkömmling kann dadurch in die Lage versetzt werden, Umgebungsvariablen des Ahnen-Prozesses zu ändern.

Die nachfolgende Quelle zeigt, wie ein Sohn-Prozeß den Wert der Umgebungsvariablen *home* verändert und einen Ablauf des Programms. Dieselbe Technik könnte ein Kommandoprozessor natürlich auch verwenden.

```
/* rfork.c                                                    */

#include <u.h>
#include <libc.h>

void main(void)
{   Waitmsg msg;

    print("Daddy: getenv(\"home\") = %s\n", getenv("home") );
    switch ( rfork( RFFDG | RFPROC ) )  {
    case -1:    print("fork failed");
            exits("fork failed");
```

```
        case 0:     putenv("home", "nirwana");
                exits("child is dead.");
        default:    print("Daddy: wait = %d\n", wait(&msg) );
                print("Daddy: getenv(\"home\") = %s\n",
                    getenv("home") );
                exits(0);
        }
        exits("");
}
```

Ein Ablauf:

```
% rfork
Daddy: getenv("home") = /usr/none
Daddy: wait = 494
Daddy: getenv("home") = nirwana
```

13.3 *fork* im Detail

Mit folgendem Programm *fork* kann man ein Kommando mit Argumenten unter genau kontrollierten Bedingungen ausführen.

```
% fork --
usage: fork [-e] [-w] [-f flag]... cmd
```

Die Flaggen haben folgende Bedeutung:

-e unterdrückt *exits()* im Programm und führt statt dessen */bin/rc* aus, das heißt, man kann *fork* mit *exec* an Stelle von *rc* ausführen und im gleichen Prozeß eine neue Kopie von *rc* starten.

-w unterdrückt *wait()*, das heißt, *cmd* wird parallel zu *fork* ausgeführt.

flag sind die Namen der Flaggen, die an *rfork()* übergeben werden können:

`proc`		erzeugt einen Prozeß
`envg`	`cenvg`	kopiert/löscht das Environment
`fdg`	`cfdg`	kopiert/löscht die Filedeskriptoren
`nameg`	`cnameg`	kopiert/löscht den Namensraum
`noteg`		startet eine neue *notepg* Gruppe
`nowait`		verhindert, daß man auf den neuen Prozeß warten kann
`mem`		benutzt, bis auf den Stack, alle Segmente gemeinsam

Wenn *proc* nicht angegeben ist, wird *cmd* an Stelle von *fork* ausgeführt. Damit kann man untersuchen, wie sich die anderen Flaggen auf einen Prozeß selbst auswirken. In diesem Zusammenhang ist *nowait* natürlich nicht erlaubt.

Wenn »-e« angegeben und *proc* und ein *cmd* nicht angegeben sind, kann man *fork* in *rc* mit *exec* ausführen und damit die Effekte auf *rc* selbst testen. Dies geht allerdings einfacher mit der in *rc* eingebauten Anweisung *rfork*.

Nach Voreinstellung werden Environment, Filedeskriptoren und Namensraum gemeinsam benützt. Ohne Flaggen wird *fork()* verwendet; das ist äquivalent zu *proc* und *fdg*. Die Quellen zu *fork* finden Sie im Begleitmaterial.

Mit *fork* kann man die Effekte der verschiedenen Optionen ausprobieren. Hier ist die »normale« Version von *fork*:

```
% fork /bin/date
father: 1653
son: 1654
Sun Aug 23 12:05:37 GPT 1994
fork: got pid 1654 user 20 sys 0 real 40
```

Gibt man *nowait* an, kann *wait()* den Prozeß nicht mehr finden.

```
% fork -f proc -f nowait /bin/date
father: 1655
fork: no living children
% son: 1656
Sun Aug 23 12:05:51 GPT 2026
```

Verwandte Prozesse können das Environment gemeinsam benutzen.

```
% echo hi > /env/hi          # neuer Eintrag im Environment
% fork /bin/cat /env/hi      # in neuem Prozess auslesen
father: 1658
son: 1659
hi
fork: got pid 1659 user 0 sys 0 real 20
```

Ein Eintrag im Environment kommt von einem neuen Prozeß an den Erzeugerprozeß zurück, weil das Environment von beiden Prozessen gemeinsam genutzt wird. Wir demonstrieren mit einer lokalen Datei *here*, die der neue Prozeß ins Environment kopiert, wo sie der alte Prozeß findet.

```
% echo here > here           # lokale Datei
% rm /env/hi                 # keine Datei 'hi' im Environment
% cat /env/hi
cat: can't open /env/hi: file does not exist
% fork /bin/cp here /env/hi
father: 1663
son: 1664
fork: got pid 1664 user 0 sys 80 real 80
% cat /env/hi               # 'hi' ist wieder im Environment
here
```

Ein neuer Prozeß kann aber auch seine eigene Kopie des Environments bekommen, auf die der Erzeugende keinen Zugriff hat. Ein Eintrag im Environment kommt dann von dem neuen Prozeß an den Erzeugerprozeß nicht zurück.

```
% echo here > here          # Datei 'here' enthaelt 'here'
% echo hi > /env/hi         # Environment 'hi' enthaelt 'hi'
% fork -f proc -f envg /bin/cp here /env/hi
father: 1668
son: 1669
fork: got pid 1669 user 0 sys 0 real 40
% cat /env/hi               #  Env. enthaelt immer noch 'hi'
hi
```

Ein abgeleiteter Prozeß kann mit einem neuen, leeren Environment aktiviert werden.

```
% lc /env | sed 2q
*    cflag  fn#do_cd font   ix   prompt sysname
0    cmd    fn#ll    h      n    ps1    t1
% fork -f proc -f cenvg /bin/ls /env
father: 1675
son: 1676
fork: got pid 1676 user 0 sys 0 real 20
```

Bei *fork()* kann man auch die Übergabe von offenen Filedeskriptoren kontrollieren. Die möglichen Effekte kann man mit einem Programm *close* untersuchen, das die als Argumente angegebenen Filedeskriptoren schließt und eventuelle Fehler anzeigt.

```
/* close.c                              */

#include <u.h>
#include <libc.h>

void main (int argc, char * argv [])
{   int errors = 0;
    char buf [ERRLEN];

    while (* ++ argv)                   /* fuer alle Argumente */
        if (close(atoi(* argv)))        /* Fehler akkumulieren */
            ++ errors, fprint(2, "close %s: %r\n", * argv);
    if (errors)                         /* Fehler berichten,   */
    {   sprint(buf, "%d error(s)", errors); /* falls vorh. */
        exits(buf);
    }
    exits(0);
}
```

Bei Aufruf durch *rc* sind normalerweise die Filedeskriptoren 0 bis 3 offen. Der Filedeskriptor 4 ist nicht offen, muß also beim Schließen zu einem Fehler führen.

```
% fork close 4 3 2 1
father: 1720
son: 1721
close 4: fd out of range or not open
fork: got pid 1721 user 0 sys 20 real 60
        msg "close 1721:1 error(s)"
```

Im Normalfall hat der neue Prozeß eine Kopie der Filedeskriptoren. Wenn er sie schließt, macht das dem alten Prozeß nichts:

```
% fork -f proc close 2 1 0
father: 1724
son: 1725
```

Werden die Filedeskriptoren allerdings von beiden Prozessen gemeinsam benützt, gibt es beim Schließen Ärger, weil ein Filedeskriptor nur einmal geschlossen werden kann.

Löscht man die Filedeskriptoren mit *cfdg*, kann sie der neue Prozeß nicht mehr schließen. Allerdings kann er dann auch nichts mehr ausgeben:

```
% fork -f proc -f cfdg close  2 1 0
father: 1726
fork: got pid 1727 user 0 sys 20 real 20
        msg "close 1727:3 error(s)"
```

Mit einem Programm *seek*, das *seek()* mit den Argumenten von seiner Kommandozeile aufruft, und mit Hilfe eines *rc*-Skripts kann man demonstrieren, daß auch beim Kopieren von Filedeskriptoren ein gemeinsamer Positionszeiger erhalten bleibt:

```
/* seek.c                              */
#include <u.h>
#include <libc.h>

void main (int argc, char * argv [])
{   int fd, type;
    long n, pos;

    if (argc < 4)           /* mindestens drei Argumente   */
    {   fprint(2, "usage: seek fd n type\n");
        exits("usage error");
    }

    fd = atoi(argv[1]);     /* Argumente abholen           */
    n = atol(argv[2]);
    type = atoi(argv[3]);

    if ((pos = seek(fd, n, type)) == -1L)   /*positionieren*/
    {   fprint(2, "seek %d %ld %d: %r\n", fd, n, type);
        exits("seek error");
    }
    print("seek %d %ld %d returns %ld\n", fd, n, type, pos);
    exits(0);
}
```

Das Skript illustriert dies:

```
% echo 'fork -f proc $* seek 0 3 0' > fc
% echo cat >> fc
% echo hello > hello
% rc fc -f cfdg < hello
father: 1742
fork: got pid 1743 user 0 sys 0 real 20
        msg "seek 1743:seek error"
hello
```

13.4 Nachrichten

An Stelle von Signalen gibt es in Plan 9 Nachrichten in Form von Texten. Ein Prozeß reagiert auf das Eintreffen einer Nachricht mit seinem Ableben, falls er keine Funktion hinterlegt hat, die beim Eintreffen einer Nachricht aufgerufen wird. Die Länge einer Nachricht ist auf *ERRLEN* - 1 (63) Zeichen beschränkt. Da aber Dateinamen übergeben werden können, ist die Informationsmenge praktisch keinen Beschränkungen unterworfen.

Nachrichten werden einerseits vom Betriebssystem geschickt — *sys: write on closed pipe, hangup ...* — bzw. via

```
echo Text > /proc/pid/note
```

von der Kommandozeile oder mittels

```
postnote(pid, "text")
```

aus einem C-Programm heraus. Dadurch ist die Möglichkeit geschaffen, einen Prozeß von außen jederzeit mit beliebiger Information zu versorgen.

Anders als bei Unix-Signalen kann ein Programm allerdings nur auf Nachrichten insgesamt reagieren; spezielle Nachrichten können nicht gesondert empfangen werden. Als Beispiel soll ein Programm feststellen, welche Art von Nachricht ankam, und dann entsprechend reagieren.

Ein Beispiel: *notify.c* hinterlegt mit *atnotify()* zwei Funktionen. Der Aufruf von *noted(NDFLT)* sorgt dafür, daß der Prozeß nach Eintreffen der Nachricht terminiert.

```
/* notify.c: test notify, noted, notify (2)          */

#include <u.h>
#include <libc.h>
#include <stdio.h>

int notify_0(void * u, char * msg)
{
    print("notify_0: msg = —%s—\n", msg );
    return(0);
}

int notify_1(void * u, char * msg)
{
    print("notify_1: msg = —%s—\n", msg );
    noted(NDFLT);                   /* default: RIP        */
    return(0);                      /* make compiler :—)   */
}

main(void)
{   int help = 1;

    printf("pid = %d\n", getpid() );
    atnotify(notify_0, 1 );
    atnotify(notify_1, 1 );
    print("1 / (help — 1) \n", 1 / ( help — 1 ) );
    print("....\n");
    return(0);
}
```

Ein Ablauf des Programms:

```
% notify
pid = 780
notify_0: msg = -sys: trap: unaligned address pc=0x915c-
notify_1: msg = -sys: trap: unaligned address pc=0x915c-
notify 780: suicide: sys: trap: unaligned address pc=0x915c
```

```
% db 780
sparc binary
last exception: unaligned address
muldiv.s:45 _udivmod+#14?        MOVW   R0, #ffffffff(R0)
$c
_udivmod() muldiv.s:42 called from _div+#8 muldiv.s:146
_div() muldiv.s:144 called from _div+#8 muldiv.s:146
...
$q
% echo kill>/proc/780/ctl
```

Ein anderes Beispiel: Ein Prozeß, *work_hard*, der sehr viel Rechenzeit benötigt, soll für eine bestimmte Zeit angehalten werden, um anderen Prozessen explizit Vorrang zu gewähren. Nach einer gewissen Zeitspanne oder explizit auch früher soll der Prozeß allerdings aus dem Ruhezustand geholt werden können. Eine Lösung besteht darin, daß dem Prozeß zwei Nachrichten geschickt werden können. Erreicht den Prozeß eine Nachricht der Form *s seconds* legt er sich *seconds* Sekunden lang schlafen; durch die Nachricht *wakeup* wird der Prozeß aufgeweckt. Eine Implementierung:

```
/* work_hard.c                              */

#include <u.h>
#include <libc.h>
#include <stdio.h>

/* Wird aktiviert, falls eine `note' den Prozess erreicht.
 * Resultat: 0 ——> Die 'note' wurde erkannt und das
 * Problem behoben
 */
int msg_handler(void * u, char * msg)
{   int to_sleep;

    switch ( msg[0] )                /* triviales Protokoll */
    {   case 'w':printf("moin moin\n");
             return (0);
        case 's': sscanf(msg + 1,"%d", &to_sleep );
              sleep( 1000 * to_sleep );
    }
    noted(NCONT);                    /* den Prozess nicht   */
    return(0);                       /* abbrechen           */
}

/* Diese Prozedur soll kurzfristig unterbrochen werden     */
void work_hard(void)
{   printf("pid: %d  works hard.\n", getpid() ); /* die pid*/
    while ( 1 ) ;                    /* work hard           */
}

main(void)
{   atnotify(msg_handler, 1 );       /* Initialisierung     */
    work_hard();                     /* Belastung steigt    */
    return(0);
}
```

Ein Ablauf:

```
% work_hard &
pid: 77  works hard.
% echo s 20 > /proc/77/note
% ps | grep work_hard
none 77 0:14 0:00 52K Sleep work_hard
% echo w   > /proc/77/note
% moin moin

%  ps | grep work_hard
%
```

Daß *msg_handler()* beim Eintreffen einer Nachricht aufgerufen wird, legt der Aufruf von *atnotify()* fest. Wird eine Nachricht an den Prozeß geschickt, wird *msg_handler()* mit zwei Argumenten aufgerufen: einem Zeiger auf eine *Ureg*-Struktur, welche die Werte der Register enthält, und einem Zeiger auf die geschickte Nachricht.

Es bleibt noch anzumerken, daß beliebig viele Funktionen hinterlegt werden können, die dann alle in der Reihenfolge der Registrierung aufgerufen werden. *noted(NCONT)* sorgt dafür, daß der Prozeß nicht terminiert.

13.5 Synchronisation

Wenn zwei Prozesse unabhängig voneinander aufeinander warten sollen, synchronisieren sie sich durch den Aufruf von *rendezvous()*. Die Prozesse werden auf alle Fälle blockiert, bis *beide* ihren Aufruf von *rendezvous()* erreicht haben. *rendezvous()* ist eine Schnittstelle, durch die der *scheduler* dazu veranlaßt wird, einen Prozeß bis zum Eintreffen eines zugehörigen Aufrufs zu blockieren. Zwischen den beiden Prozessen kann eine Information vom Typ *ulong* ausgetauscht werden. *rendezvous()* schickt das zweite Argument an seinen Partner. In diesem Fall schickt *client* seine Prozeß-Id an *master*. *postnote()* verwendet diese, um eine Nachricht an *client* zu schicken. Die nachfolgende Abbildung zeigt den Quelltext von *master*.

Als Beispiel sei folgendes Problem zu lösen: Zwei Prozesse, *master* und *client*, sollen sich zu einem unbestimmten Zeitpunkt synchronisieren. Es ist unbekannt, ob der *master* oder der *client* den Synchronisationspunkt zuerst erreicht. Ist das Rendezvous eingetreten, soll der *master* an den *client* eine Nachricht schicken. Nachdem die Nachricht verschickt ist, sollen sich *master* und *client* nochmals synchronisieren.

```
/* master.c                              */

#include <u.h>
#include <libc.h>
#include <stdio.h>
#include "local_tag.h"                        /* TAG          */
main(void)
{   ulong tag = TAG;
    int   rendezvous_val;
    printf("\t\t Master:\n");
    printf("\t\t tag = —%d—\n", tag );
```

```
    printf("\t\t Master: Call rendezvous and send note.\n");

    rendezvous_val = (int)rendezvous(tag, 0L );

    postnote(PNPROC, rendezvous_val,
                "Client! Here is the Master." );

    rendezvous_val = rendezvous(tag + 1, 0L );

    printf("\t\t Master: r_result = %ld\n", rendezvous_val);
    printf("\t\t Master: RIP\n");
    return(0);
}
```

Der Aufruf von *rendezvous()* hält den Prozeß so lange an, bis ein zweiter Prozeß ebenfalls *rendezvous()* mit demselben TAG aufgerufen hat. Ist dies eingetroffen, wird mit *postnote* eine Nachricht an den zweiten Prozeß geschickt. Die Prozeß-Id des zweiten Prozesses wird von *rendezvous* als Resultat geliefert. Im zweiten *rendezvous()* synchronisieren sich *master* und *client* zum zweiten Mal.

Die nachfolgende Abbildung zeigt den Quelltext von *client*.

```
/* client.c                              */

#include <u.h>
#include <libc.h>
#include <stdio.h>
#include "local_tag.h"

/* Wird aktiviert, falls eine `note' den Prozess erreicht.
 * Resultat: 0 ——> Die 'note' wurde erkannt und das
 *          Problem behoben
 */
int msg_handler(void * u, char * msg)
{   printf("    Client: msg = —%s—\n", msg );
    noted(NCONT);               /* continue forever        */
    return(0);
}

main(void)
{   atnotify(msg_handler, 1 );
    printf("    Client: pid = %d\n", getpid() );
    printf("    tag = —%ld—\n", TAG );
    printf("    Client: rendezvous is called.\n");
                                /* client pos == master pos*/
    rendezvous(TAG, (ulong) getpid() );

                                /* r. interrupted          */
    if ( rendezvous(TAG + 1, 1L) == ~0 )
                                /* client pos == master pos*/
        rendezvous(TAG + 1, 2L);
    printf("    Client: RIP\n");
    return(0);
}
```

Da der *client* eine Nachricht empfangen soll, wird mit *atnotify()* eine Funktion hinterlegt, die bei Eintreffen der Nachricht aktiviert wird. Mittels *rendezvous()* wird die

Prozeß-Id des Klienten an den Partner geschickt. Damit sich *client* und *master* in einem zweiten *rendezvous*-Punkt treffen können, richtet *client* zwei *rendezvous*-Punkte ein. Je nach Ablaufverhalten von *master* und *client* können folgende Situationen eintreten:

- *client* erhält die Nachricht, bevor er den zweiten *rendezvous*-Punkt erreicht. In diesem Fall synchronsieren sich *master* und *client* im ersten *rendezvous*-Punkt.
- *client* erhält die Nachricht, nachdem er den zweiten *rendezvous*-Punkt erreicht. Das Eintreffen der Nachricht kippt den Prozeß aus dem *rendezvous*-Punkt; *master* und *client* synchronsieren sich im zweiten *rendezvous*-Punkt. Ein Ablauf:

```
% master &
                Master:
                tag = -10-
                Master: Call rendezvous and send note.

% client
        Client: pid = 267
        tag = -10-
        Client: rendezvous is called.
        Client: msg = -Client! Here is the Master.-
        Client: RIP
                 Master: r_result = 2
                 Master: RIP
```

Mit *rendezvous()* kann keine race-freie Prozeßkommunikation erreicht werden. Dazu müßte man *spin locks* verwenden, die man in Alef finden kann.

13.6 Zusammenfassung

Die Abbildung von Prozessen in das Dateisystem ermöglicht einen einfacheren Zugang zum Prozeß. Der breitere Kommunikationskanal via *note(2)* erlaubt eine einfachere Kommunikation, als dies unter UNIX oder Windows (NT oder 95) möglich wäre.

14
Sicherheit nach UNIX

Betrachtet man die aktuelle, sehr verständliche Diskussion über die Sicherheit von und in Rechnernetzen, dann gewinnt man den Eindruck, daß sich das Hauptinteresse auf die Errichtung von hohen und mit Stacheldraht bewehrten Mauern konzentriert, um die Sicherheitslücken zu stopfen. Erhält man Briefe, die mit den Worten ''We would like to notify you of a potential security issue ...'' beginnen, greift man nach einer kurzen Schrecksekunde und einer längeren Nachdenkphase zu Mörtel und Schnellzement und kleistert die Lücken weiter zu.

Die größten Angriffsflächen, die ein UNIX-System bietet, sind in seinem ursprünglichen Design begründet. Plan 9, der UNIX-Nachfolger, hat aufgrund von zeitgemäßen Design-Entscheidungen viele Sicherheitsprobleme a priori nicht, die ein UNIX-basiertes System hat.

14.1 Daten-Spionage

Jeder Prozeß unter Plan 9 besitzt einen eigenen, vererbbaren oder teilbaren *Namensraum*. Der Namensraum eines Prozesses ist der aktuelle, komplette Dateibaum aus der Sicht des Prozesses. Er kann dynamisch und individuell von jedem Prozeß lokal geändert werden.

Der Namensraum hat i.a.R. nicht sehr viel mit dem Dateibaum auf einer Festplatte zu tun, sondern ist ein beliebig konfigurierbarer virtueller Dateibaum. Die Namensräume sind normalerweise so aufgebaut, daß die Schnittmenge der schreibbaren Bereiche zweier Prozesse unterschiedlicher Nutzer leer ist. Dadurch ist die Grundlage für eine Informationsgewinnung mittels ''Trojanischer Pferde'' in all ihren Spielarten nicht mehr möglich.

14.2 Authentifizierung

Die Authentifizierung von Terminals und Nutzern erfolgt unter Inanspruchnahme eines Authentifizierungsservers, welcher im Netz bekannt ist. Der Authentifizierungsserver kann als *stand-alone*-Rechner oder als CPU-Server betrieben werden.

Im ersten Fall kann der Rechner nur über die Konsole oder das Authentifizierungs-Protokoll angesprochen werden. Falls der Rechner in einem zugangsgesicherten Raum steht, ist gewährleistet, daß unerwünschte Modifikationen nicht vorgenommen werden können, weil das Protokoll diese Funktionalität nicht besitzt.

Die zweite Variante eröffnet die Möglichkeit der Modifikation des Authentifizierungsservers durch Dritte, weil auf dem Authentifizierungsserver beliebige Benutzerprogramme ablaufen können. Eine Manipulation der relevanten Daten ist nach unserer Ansicht nicht möglich, da keine zugängliche Schnittstelle zu diesen Daten existiert. Trotzdem ist diese Variante als ein Zugeständnis an die zur Verfügung stehenden Geldmittel anzusehen, weil nie absolut sicher ausgeschlossen werden kann, daß die Schnittstelle nicht geschaffen werden kann.

Jeder Rechner hat drei, für seine Authentifizierung benötigte Schlüssel:

- 56-Bit-DES Maschinenschlüssel,
- 28-Byte Authentifizierungs-ID,
- 48-Byte Authentifizierungs-Domainname.

Die ID ist ein Benutzername, der identifiziert, für wen der Kern läuft. Der Domainname identifiziert den Bereich, in dem die ID gültig ist. Das Tupel (ID, Domainname) identifiziert den Besitzer eines Schlüssels.

Wenn ein Plan 9-Terminal gestartet wird, wird der Nutzer nach seinem Login-Namen und seinem Paßwort gefragt. Die Authentifizierungs-ID des Terminals wird mit dem Login-Namen initialisiert. Das Paßwort wird mittels *passtokey(2)* in einen 56-Bit-DES-Schlüssel konvertiert und als Maschinenschlüssel verwendet.

Wenn ein CPU- oder File-Server startet, liest er seinen Maschinenschlüssel, ID und Domainnamen aus dem *non-volatile* RAM. Dies erlaubt einen Neustart ohne die Unterstützung durch einen Operateur.

14.3 File Service

File-Service-Verbindungen überdauern normalerweise einen längeren Zeitraum. Wenn ein Benutzerprozeß versucht, Zugriff auf einen File-Server zu bekommen, muß er sich identifizieren. Folgende vier Parteien sind daran beteiligt: der File-Server, der Kern des Klienten (Terminal), der Benutzerprozeß auf dem Klienten und der Authentifizierungsserver. Die Grundidee hinter dem Authentifizierungs-Protokoll besteht darin, den Server davon zu überzeugen, daß der Klient zu Recht für den Benutzerprozeß spricht.

Am Beginn einer Session zwischen Klient und File-Server wird globale Information in Form von zwei zufällig gewählten *Challenges* ausgetauscht.

Session	Klient an Server:	`Tsession`	*client*
	Server antwortet:	`Rsession`	*server*

client und *server* sind zufällig gewählte 8-Byte-Strings (*Challenges*) für die Session; *servername* ist der eindeutige Name des Servers.

Als nächstes muß der Klient für seinen Benutzerprozeß vom Authentifizierungsdienst ein Ticket beschaffen:

Ticket	Klient an AS:	`AuthTreq`	*server servername clientname client-user*
	AS an Klient:	`AuthOK`	*client-ticket server-ticket*
	oder	`AuthErr`	*Fehlertext*

servername ist der Name des Servers für die Session; er stammt von *Rsession*. *clientname* ist der Name des Klienten, *client-user* ist der Name des Benutzers auf dem Klienten.

Die beiden Tickets dienen dazu, beim Klienten und Server zu beweisen, daß der Authentifizierungsdienst die Zusammengehörigkeit der Identitäten kontrolliert hat.

client-ticket ist mit dem Schlüssel verschlüsselt, den der Klient kennt, *server-ticket* kann nur der Server der Session entschlüsseln. Im verschlüsselten Bereich

der Tickets ist markiert, ob es für den Klienten oder Server bestimmt ist; außerdem sind die Server-Challenge *server*, zugeordnete Benutzernamen für Klient und Server sowie ein neuer Schlüssel *key* enthalten, mit dem anschließend Klient und Server direkt verkehren:

Key	*client-ticket*:	$AuthT_c$	*server client-user server-user key*
	server-ticket:	$AuthT_s$	*server client-user server-user key*

Jetzt kann sich der Klient bei dem Server anmelden:

Attach	Klient an Server:	`Tattach`	*server-ticket*
	Server an Klient:	`Rattach`	*answer*

Das *server-ticket* bleibt so verschlüsselt, wie es vom Authentifizierungsdienst kam. Der *authenticator* ist mit *key* verschlüsselt und kann folglich vom Server entschlüsselt werden, da dieser ja den Schlüssel kennt, um das *server-ticket* zu decodieren und den *key* zu entnehmen:

Authen-*ticator*	*authenticator*:	$AuthA_c$	*server count*

Dem Server begegnet jetzt also wieder seine eigene Challenge *server* sowie ein *count*, der jeweils erhöht wird, um immer neue Werte zu garantieren.

Auch *answer* wird mit dem gemeinsam erworbenen *key* verschlüsselt, und der Klient kann kontrollieren, daß sein Server noch immer seine Challenge *client* kennt.

Geht man davon aus, daß nur der Authentifizierungsserver alle Schlüssel sowie die Zuordnung von Benutzernamen zwischen Server und Klienten kennt und daß Klient und Server ihren eigenen Schlüssel geheimhalten, läuft die Sicherheit des Verfahrens darauf hinaus, daß der Authentifizierungsserver abgesichert sein muß — die Nachrichten auf dem Netz sind verschlüsselt und bezüglich ihrer Urheber prüfbar.

Für *remote execution*, Änderung des Paßworts und Authentifizierung mittels der *SecureNet box* finden ähnliche Abfragen statt.

14.4 Paßwörter

Paßwörter werden nur verschlüsselt in Richtung Authentifizierungsserver übertragen. Die Maschinenpaßwörter selbst sind im *non-volatile* RAM des Authentifizierungsservers abgelegt, auf das kein Benutzerprozeß Zugriff bekommen kann. Modifikationen im Kern eröffnen diese Möglichkeit natürlich, aber ein Angriff à la *crack* ist somit nicht möglich. Ein Abhören der Kommunikationswege ist nicht sehr erfolgversprechend, weil die Paßwörter nur verschüsselt verschickt werden. Die wesentlich einfacheren Methoden, raten, fragen oder ''einen Blick auf die Tastatur werfen, während das Paßwort eingetippt wird'', sind natürlich immer noch gangbar.

Das Anmelden an einem Plan 9-Terminal ist mit einem *reboot* desselben verbunden. Somit ist ein Ausspionieren der Paßwörter nicht über eine simple Simulation des *login*-Panels möglich.

14.5 Administration ohne Superuser

Unter Plan 9 gibt es keinen Benutzer mit der *user id 0*, kein äquivalent zum Super-User und keinen s-Bit-Mechanismus. Die Konsequenzen sind offensichtlich. Der Zugriff auf Dateien und Kataloge wird ausschließlich durch die Zugriffsrechte gesteuert.

Es stellt sich dann allerdings die Frage, wie eigentlich das Heimatverzeichnis eines neuen Benutzers angelegt wird, wenn das dafür in Frage kommende Verzeichnis nicht für ''die Welt'' schreibbar ist? Wie werden Dateien aus dem Dateisystem entfernt, falls ein Nutzer sein Paßwort vergessen hat? Wie wird die Datensicherung arrangiert? Die Lösung ist offensichtlich, wenn man sich nochmals kurz an die Struktur eines Plan 9-Netzes erinnert.

Die File-Server halten alle Daten. Also liegt es nahe, daß dort mit sehr limitierten Kommandos die notwendigen Administrationsarbeiten durchgeführt werden.

An der normalerweise gesicherten Konsole des File-Servers können Kommandos zum Anlegen und Löschen von Dateien bzw. Katalogen, Einrichten und Löschen von Benutzer-Accounts, Testen der Platten, Sichern der Daten und Setzen der Uhrzeit eingegeben werden. Die Schnittstelle läßt allerdings keine Modifikation der Rechte einer Datei oder eines Katalogs zu. Das Vergessen eines Paßworts kann somit unangenehme Konsequenzen haben: Löschen der Daten.

Da der Kern des File-Servers nicht multi-Prozeß-fähig ist, besteht keine Möglichkeit, die Kommandos anders als über die Konsole abzusetzen.

14.6 Zusammenfassung

Plan 9 enthält ein ausgeklügeltes Verfahren, um Server und Klienten mit gemeinsamen Schüsseln zu versorgen. Davon abgesehen, attackiert Plan 9 nicht die Sicherheitsprobleme, die durch die Nutzung existenter Protokolle und Mechanismen entstehen, die nicht substituiert werden können. Klarerweise ist demzufolge z.B. IPv6 nicht realisiert. Es wurden ''nur'' die Sicherheitslücken geschlossen, die in UNIX-Systemen durch die notwendige Existenz eines ausgezeichneten Nutzers entstehen. Die s-Bit-Problematik ist eine Konsequenz, die mit der Eliminierung des Ausgangsproblems gleich mit gelöst worden ist.

15
Netzprotokolle

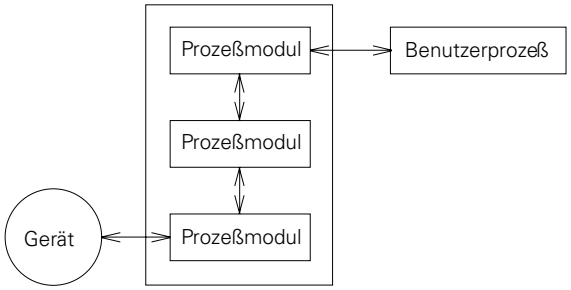
Der Aufbau von Plan 9 basiert u.a. auf der Verbindung von CPU-Servern, File-Servern und Terminals. Natürlich kann das System auch allein auf einem einzigen Rechner abgefahren werden, genau wie ein PC mit Linux das Betriebssystem Unix simulieren kann. Doch der eigentliche Sinn von Plan 9 ist es, eine Vielzahl von Rechnern und Netzwerken miteinander zum Datenaustausch zu verbinden. Insofern bezieht sich der Quellcode von Plan 9 zur einen Hälfte auf Kernels und zur anderen Hälfte auf Netzwerke und Protokolle.

Im Kernel teilt sich die Anwendung von Netzwerken in drei Teile auf: das Hardware Interface, der Ablauf von Protokollen und das Programminterface. Ein Gerätetreiber, wie für einen Drucker, ein Modem oder eine Ethernetverbindung, verknüpft mit einem Stream das Hardwareinterface mit dem Programminterface.

Die Netzwerke sind mit Hilfe der Interfaces miteinander verbunden. Die Kommunikation erfolgt jeweils über sogenannte Protokolltreiber, die in bestimmten Verzeichnissen unterhalb von */net* angesprochen werden können. Je nachdem welches Verzeichnis ausgesucht wird, erfolgt die Verbindung mit den Protokollen IL, TCP und UDP. Dabei ist IL ein neues Protokoll in Plan 9. IL ist Message-basiert wie UDP und so sicher wie TCP. Zudem ist IL schneller und effektiver als TCP. Die Funktionsweise von IL soll später etwas ausführlicher behandelt werden.

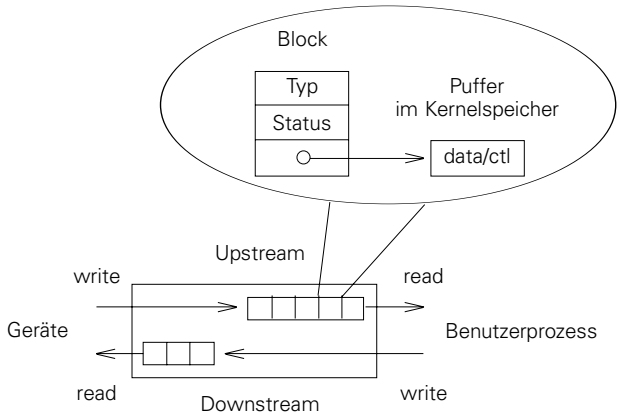
15.1 Streams

Die Verbindung zwischen Gerätetreibern und Benutzerprozessen wird von den meisten Protokollen wie TCP und IL durch sogenannte Streams realisiert. Ein Stream realisiert einen bidirektionalen Datenfluß und besteht dazu aus einer linearen Liste von Prozeßmodulen.



Jedes Prozeßmodul besitzt zwei Routinen: einen Upstream und einen Downstream. Der Upstream ist der Datenfluß vom Gerät zum Benutzerprozeß, der Downstream ist der Datenfluß in die umgekehrte Richtung. Am oberen Ende sind die Streams mit dem Benutzerprozeß und am unteren Ende mit einem Kernelprozeß als Platzhalter des Geräts verbunden.

Ein Prozeßmodul setzt sich wiederum aus zwei Warteschlangen zusammen. In die Warteschlangen werden Blöcke eingefügt und herausgeholt, die sich aus einem Typ, ihrem momentanen Status und einem Zeiger auf einen Daten- bzw. Kontrollpuffer, welche dynamisch im Kernelspeicher allokiert werden, zusammensetzen.



Aus der Benutzersicht sind die Puffer die *data*- und *ctl*-Dateien, wie z.B. in */net/il/0*. Sie können mit *open* geöffnet und mit *close* geschlossen werden. In die Datei *data* kann mit *write* geschrieben, wobei die Daten in *32 K* lange Blöcke aufgeteilt werden. Falls die Nachricht länger als *32 K* ist, wird als letzter Block ein Delimiter angehängt. In die *ctl*-Datei kann hingegen ein Kontrollblock hineingeschrieben werden, der eines der folgenden Kommandos im ASCII-Format enthält:

push name
Das Prozeßmodul *name* wird am Anfang der Warteschlange eingefügt.

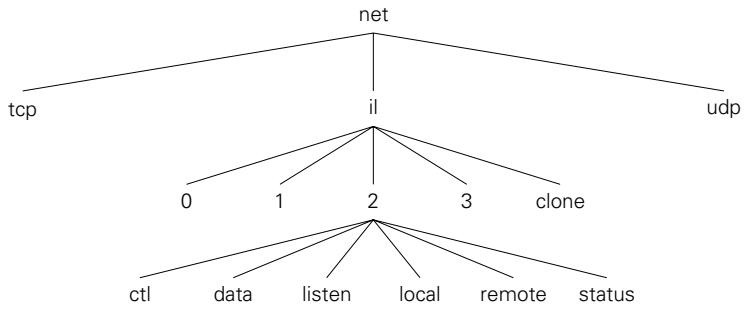
pop
Das oberste Modul wird wieder aus der Schlange geholt.

hangup
Von der Seite des Gerätetreibers aus wird ein *hangup*-Signal geschickt.

Alle weiteren Kommandos sind modulspezifisch, wie z.B. *connect*, *announce* und *backlog* bei den IP-Protokollen.

15.2 Protokolltreiber

Ein Rechner kann über Ethernetkabel, Modemverbindungen oder Datakit mit einem anderen Rechner über IL, TCP oder UDP für den Datenfluß verbunden werden. Jedes Protokoll wird in einem der Unterverzeichnisse */net/il*, */net/tcp* oder */net/udp* angesprochen.



Ein Protokolltreiber arbeitet für eines der Unterverzeichnisse und kann darüber angesprochen werden. In jedem Unterverzeichnis von */net* befinden sich eine *clone*-Datei und weitere Unterverzeichnisse *0, 1, 2, 3, ...* In *clone* befindet sich eine Nummer für das nächste Unterverzeichnis. Durch das Lesen von *clone* wird ein neues Verzeichnis mit dieser Nummer als Namen reserviert solange *clone* geöffnet ist. Wenn z.B. in *clone* die Zahl *3* steht, bedeutet dies, daß das Verzeichnis*/net/*/3* bei der nun kommenden neuen Verbindung benutzt wird. Die numerierten Unterverzeichnisse enthalten die eigentlichen Dateien für die Kommunikation und stellen einen Port dar: *data*, *status*, *local*, *remote*, *listen* und *ctl*.

Die eigentliche Datenübertragung geschieht über die Datei *data*, nachdem die Verbindung bereits aufgebaut wurde. Der augenblickliche Status einer Verbindung kann immer mit *status* festgestellt werden:

```
% cat /net/*/*/status
il/0 0 Closed rtt 104 ms 0 csum
il/1 1 Established rtt 100 ms 0 csum
tcp/0 1 Listen listen listen 1000+0
tcp/1 1 Listen listen listen 1000+0
tcp/10 1 Listen listen listen 1000+0
tcp/11 1 Listen listen listen 1000+0
tcp/12 1 Listen listen listen 1000+0
tcp/13 1 Established connect  35+18
tcp/14 0 Closed connect  0+0
udp/0 0 Datagram
```

In *local* befindet sich die komplette lokale IP-Adresse inklusive der Nummer des Ports, über den die Verbindung erstellt wurde. Falls gerade eine Verbindung zu einem anderen Rechner besteht, kann man seine IP-Adresse über *remote* ermitteln. Ansonsten enthält *remote* die Null-Adresse *0.0.0.0!0*.

```
% cd /net/tcp/13; cat status
tcp/13 1 Established connect  14+17
% cat local
131.173.161.42!513
% cat remote
131.173.17.251!1022
```

```
% cd /net/tcp/12; cat status
tcp/12 1 Listen listen listen 1000+0
% cat local
131.173.161.42!80
% cat remote
0.0.0.0!0
```

Über *listen* horcht ein Protokolltreiber auf ankommende Anfragen zum Erstellen einer neuen Verbindung. Kommt von einem Klienten eine solche Anfrage, so wird beim Server ein neues Verzeichnis für die Kommunikation erzeugt. Über *listen* läßt sich danach die Nummer des neuen Verzeichnisses herausfinden.

An *ctl* können verschiedene Nachrichten als Texte gesendet werden. Die drei wichtigsten davon sind:

connect ipaddress!port[!r]
: Die *remote*-Adresse und die Portnummer werden festgelegt. Falls zusätzlich *!r* angefügt wird und bisher keine lokale Adresse mit *announce* bestimmt wurde, kann eine lokale nicht zulässige Adresse kleiner als *1024* angegeben werden. Dies ist z.B. für Verbindungen zu Unix-Rechnern für *rlogin* nötig.

announce X
: *X* ist entweder eine dezimale Portnummer oder ***. Die lokale Portnummer wird auf *X* gesetzt. Alle ankommenden Nachrichten werden über diesen Port angenommen.

backlog n
: Mit *n* wird die Anzahl der Anfragen nach einer Verbindung eingeschränkt. Als Standardwert ist *5* vorgegeben. Wenn mehr als *n* Anfragen kommen, werden alle folgenden abgelehnt.

Für den Aufbau und den Datenfluß zwischen Klient und Server benötigt man vor allem die Dateien *clone*, *listen* und *data*. Ein Server stellt mit *announce()* einen Port für einen Klienten zur Verfügung. Dabei wird die *clone*-Datei (unter */net/il* zum Beispiel) geöffnet und ein neues Unterverzeichnis A erstellt. In dem neuen Verzeichnis A kündigt der Server mit *announce* in *ctl* an, daß er einen bestimmten Port für die Kommunikation reservieren möchte. Danach öffnet der Server die *listen*-Datei in dem Verzeichnis und wartet mit *read()* auf eine Anfrage des Klienten.

Meldet sich der Klient über den Port, wird ein neues lokales Unterverzeichnis B beim Server erstellt. Der Server kann die Verbindung mit *accept()* annehmen oder mit *reject()* ablehnen. Bei *accept()* wird der *ctl*-Datei des Verzeichnisses A einfach *accept* überliefert, um die Verbindung zu genehmigen. Mit *read()* und *write()* kann der Server nun Daten mit dem Klienten über *data* in Verzeichnis B austauschen. Zum Schluß müssen *ctl* und *data* in den Verzeichnissen A und B wieder geschlossen werden, um die Verbindung zu beenden.

Der Klient wartet nicht wie der Server, sondern wählt den Server mit *dial()* direkt an und kann danach sofort Daten abschicken und einlesen. Die Funktion *dial* öffnet ebenso wie *announce()* die *clone*-Datei, um ein neues Unterverzeichnis A zu reservieren. Über *ctl* wird mit *connect* auch hier ein zweites Verzeichnis B für die Kommunikation mit dem Server erstellt.

15.3 IL

Mit den *9P*-Nachrichten in Plan 9 kam die Suche auf nach einem passenden Protokoll zum Verschicken der Nachrichten über Ethernet und Internet. TCP besitzt keinen Delimiter, wie er für *9P* erforderlich ist, und ist für die Aufgabe zu komplex. Obgleich UDP einen Delimiter enthält, ist dieses Protokoll nicht sicher genug in bezug auf die sequentielle Datenübertragung. Um diese Nachteile zu beseitigen, wurde IL entwickelt, umgeben von IP. Es sollte ein schnelles Protokoll mit garantierter Sicherheit bei der sequentiellen Datenübertragung mit Delimitern sein.

Alle Verbindungen werden mit Hilfe von 32-Bit IP-Adressen und 16-Bit IL-Portnummern eindeutig gekennzeichnet. Zum einen gibt es die *local*-Adressen und -Ports für den Sender selbst und zum anderen die *remote*-Adressen und -Ports für die Kennzeichnung des Empfängers beim Sender.

Der Verbindungsaufbau kann auf zwei verschiedene Arten vorkommen. Wurde eine Nachricht empfangen, deren Adressen und Ports keiner der bereits geöffneten Verbindungen entspricht, so wird eine neue Verbindung aufgebaut. Deren *remote*-Adressen und -Ports sind die des Senders, während die lokalen den Zieladressen des Senders entsprechen. Möchte der Benutzer selbst eine Verbindung aufbauen, so bestimmt er die Adressen und Ports selbst, insofern sie die gleichen Bedingungen wie die der IP-Syntax erfüllen.

Jede Nachricht besteht aus einem einzigen Schreiben vom Betriebssystem. Zur Identifizierung der Nachricht wird eine ID angehängt. Zu Beginn einer Verbindung wird die ID zufallsmäßig vom Sender festgelegt und dem Empfänger übermittelt. Bei jeder neuen Nachricht erhöht der Sender die ID um eins. Wird die Nachricht noch einmal gesendet, behält sie die gleiche ID wie zuvor. Nach dem Verbindungsaufbau werden die Nachrichten über einen Port verschickt und empfangen. Am Ende wird die Verbindung wieder geschlossen, auch wenn sie irregulär unterbrochen wurde.

Einer der Vorteile von IL gegenüber IP ist seine Sicherheit bei der Datenübertragung. Sowohl beim Aufbau, bei der Übertragung selbst als auch beim Abbau müssen die Nachrichten vom Empfänger mit einer eigenen Nachricht als Antwort bestätigt werden. Die Bestätigung wird entweder direkt zusammen mit einem anderen Antwortteil oder alleine innerhalb der nächsten 200 Millisekunden zurückgeschickt. Im folgenden soll der Ablauf beim Verschicken einer Nachricht näher erläutert werden.

Der Sender versucht, seine Daten als *data*-Nachricht an den Empfänger zu schicken. Danach wartet er auf eine Bestätigung vom Empfänger als *ack*-Nachricht. Im Durchschnitt kommt sie nach einer Verzögerung von 100 Millisekunden. Der Sender wartet viermal so lange, wie der Ausfall der ersten Bestätigung brauchte. Ist dann immer noch keine Antwort vom Empfänger angekommen, werden die Daten oder die Bestätigung als verloren angesehen. Folglich werden sie erneut abgeschickt, diesmal als eine *dataquery*-Nachricht.

Der Empfänger bestätigt die Ankunft einer Nachricht mit *state*. Dabei kann es sich hierbei allerdings um die wieder verschickte alte Nachricht oder, falls die Priorität höher ist, um eine neue Nachricht handeln, die nach der verlorengegangenen

abgeschickt wurde. Bei der bisher vorhandenen Implementation des Empfängers werden bis zu 10 Nachrichten im voraus in einem Stapel gespeichert. Der Stapel wird bei einer erneuten *dataquery*-Nachricht bzgl. der Priorität durchsucht. Die Nachricht mit der höchsten Priorität wird bestätigt. Damit kommt es zumindest nicht sofort zu einem Stau durch die fehlerhaft übertragenen Nachrichten.

Die verlorengegangenen Nachrichten werden nach einer kurzen Wartezeit so lange wieder vom Sender abgeschickt, bis er eine Empfangsbestätigung erhält. Die Wartezeiten verlängern sich dabei exponentiell. Kommt es zu keiner Bestätigung, nimmt der Sender nach einer gewissen Zeit an, daß die Verbindung unterbrochen wurde und schließt sie bei sich.

Fand in einer Verbindung für längere Zeit keine Datenübertragung mehr statt, weil sie nicht nötig oder erwünscht war, schickt der Sender alle 6 Sekunden eine *query*-Nachricht. Der Empfänger antwortet darauf wiederum mit *state*. Falls auch dadurch nach 30 Sekunden keine Nachrichten empfangen wurden, schließt der Sender oder der Empfänger seine Verbindung. Auf diese Weise werden die Ressourcen für längst unterbrochene Verbindungen wieder freigegeben.

15.4 *aux/listen*

Mit *announce()*, *listen()* und *accept()* kann in C und in *alef* so lange auf eine Verbindung von einem Server gewartet werden, bis eine Anfrage von einem möglichen Klienten kommt. Es wäre sehr komfortabel, wenn genau dieses Warten bei bestimmten Ports durch ein Programm automatisiert wird, welches beim Aufbau einer Verbindung ein spezielles Shellskript als Service für einen Port aufruft. Zu diesem Zweck dient *aux/listen*. Es wird zu Beginn gestartet und stellt für jeden Port einen Prozeß her. Der Name eines Shellskripts, das für einen Port zur Verfügung stehen soll, besteht aus dem Namen des Protokolls und der Portnummer. Zum Beispiel wird die Datei *tcp23* genau dann aufgerufen, wenn eine Verbindung über ein TCP-Protokoll auf dem Port *23* erstellt wurde.

Einige Service sind in Plan 9 bereits vorgegeben. So steht das Skript *tcp23* für eine telnet-Verbindung. *tcp23* ruft dabei den Dämon *telnetd* auf. Der Start von *aux/listen* kann z.B. wie folgt aussehen, damit alle vorgegebenen Shellskripten aus */bin/service* für TCP verwendet werden:

```
% aux/listen tcp
```

Nun können alle Shellskripten, beginnend mit *tcp* und deren Ports, ausgenutzt werden. *telnet* kann man nun sowohl von einem anderen Plan 9-Rechner als auch von einem Unix-Rechner aus starten.

Von einem Plan 9-Rechner:

```
% echo $cpu
frigga
% telnet tcp!frigga
connected to tcp!frigga!telnet on /net/tcp/17
user: none
```

```
cpu% echo $cpu
newage
cpu% pwd
/usr/none
```

und von einem Unix-Rechner:

```
thor> telnet frigga
Trying 131.173.161.42 ...
Connected to frigga.
Escape character is '^]'.
user: none
cpu% echo $cpu
newage
cpu% pwd
/usr/none
```

In beiden Fällen wird der Port *23* angesteuert, so daß */bin/service/tcp23* aufgerufen wird. Diese Datei enthält jedoch nur *#!/bin/aux/telnetd*, um den *telnet*-Dämon im weiteren zu verwenden.

15.5 Vorgegebene Services

In ähnlicher Art und Weise kann eine Vielzahl von bereits vorgegebenen Services verwendet werden, die sich in */bin/service* befinden. Momentan gibt es bei unserer Installation Skripten für IL, TCP, Datakits und für Faxe. Die Namen der Shellskripten für Datakits bestehen aus *dk* und dem Namen des Service, z.B. *dkexportfs*. Für Faxgeräte gibt es ausschließlich *telcodata* und *telcofax* zum Empfang von Faxdaten. Hier sei nur eine kurze Auflistung der bedeutendsten Skripten für TCP und IL gegeben (siehe *listen(8)*).

tcp2
Verbindung bleibt für eine lange Zeit erhalten (*sleep 100000*).

il7 Alle empfangenen Daten werden mit *echo* zurückgeschickt.

il9 Alle empfangenen Daten werden nach */dev/null* geleitet.

tcp21
ftp-Verbindung

tcp23
telnet-Verbindung (siehe unten)

tcp25
smtp für Mails

tcp80
HTTP-Dämon, z.B. für *mothra* und *netscape* (siehe unten).

tcp513
rlogin-Verbindung (siehe unten)

tcp515
LP-Dämon für Drucker

tcp564
Wie *tcp17007* und *il17007*, allerdings ohne Authentifizierung. Erlaubt Unix-Systemen die Sicht auf Plan 9-Dateien.

il565 tcp565
Ausgabe der Adresse der eingegangenen Nachricht.

il17005
Server für das *cpu(1)*-Kommando.

il17007 tcp17007
Stellt einen Teil des Namensraums mit Authentifizierung zur Verfügung.

15.6 Eigene Services

Außer den vorhandenen Services kann man auch eigene Skripten einfügen. Jedes Skript erhält beim Aufruf drei Argumente: den Service-Namen, den Protokolltyp und das Verzeichnis des Ports beim Empfänger der Nachricht. Das folgende Beispiel gibt die drei Argumente aus, wenn eine Nachricht als TCP-Protokoll über den Port *1000* empfangen wurde.

```
#!/bin/rc
echo remote shell—script: $1 $2 $3
```

Erneut startet man zunächst *aux/listen*, um die einzelnen Ports abzuhorchen. Diesmal liegt das Skript jedoch in einem eigenen Verzeichnis, z.B. */usr/dgimeyer/listen*. Damit die Shellskripten in diesem Verzeichnis überwacht werden können, gibt es die Optionen *-d srvdir* und *-t trustsrvdir*. Mit *aux/listen -d /usr/dgimeyer/listen tcp* werden die sich in diesem Verzeichnis befindenden Skripten als Benutzer *none* gestartet. Mit *aux/listen -t /usr/dgimeyer/listen tcp* werden sie hingegen von dem gleichen Benutzer aus gestartet, der zuvor auch *aux/listen* aufgerufen hat.

In Unix kann man einen Port über TCP mit *telnet* ansprechen, indem man als zweites Argument die Portnummer angibt.

```
thor> telnet frigga 1000
Trying...
Connected to frigga.
Escape character is '^]'.
remote shell-script: tcp1000 tcp /net/tcp/5
Connection closed by foreign host.
```

Von einem völlig anderen Rechner mit einem anderen Betriebssystem kann man also ein Shellskript auf einem Plan 9-Rechner aufrufen.

Die Argumente der Shellskripten nützt u.a. *tcp565* (*whoami*) aus, um die Adressen des Klienten und des Servers sowie die Portnummer auszugeben (siehe */bin/service/tcp565*).

15.7 telnet und rlogin

Um einen Port direkt mit der Standardeingabe anzusprechen und die Ergebnisse in der Standardausgabe zu erhalten verwendet man einfach das *telnet*-Kommando in Plan 9 und auch in Unix. In Unix kann damit zum einen der TCP-Port *23* für das *telnet*-Protokoll angesprochen werden. Zum anderen kann ein spezieller Port verwendet werden, der als zweites Argument übergeben wird.

Auf diese Weise wird *whoami* in Plan 9 mit

```
% telnet tcp!frigga!565
```

oder in Unix mit

```
thor> telnet frigga 565
```

aufgerufen.

Um sich von Unix aus mit *rlogin* bei einem Plan 9-System anzumelden, wird einfach

```
thor> rlogin frigga -l none
cpu% pwd
/usr/none
cpu% who
bootes
dgimeyer
none
```

verwendet. Das Kommando *rlogin* spricht den Port *513* über *tcp* an. Danach ist man als Benutzer *none* von Unix aus auf dem Plan 9-Rechner *frigga*. Mit *exit* wird das *rlogin* wieder beendet.

Bei beiden Kommandos ist zu beachten, daß sowohl *telnet* als auch *rlogin* nur mit einer sogenannten *SecureNet*-Box (siehe securenet(8)) auch andere Benutzer außer *none* von einem fremden System aus zulassen. Da bei uns eine solche Box nicht zur Verfügung steht, beziehen sich alle Beispiele auf einen *none*-Benutzer.

15.8 http, *mothra* und netscape

Bisher fehlte in dem verteilten System Plan 9 noch das *http*-Protokoll. Doch auch dies wird mit dem Dämon *httpd*, welcher im Skript *tcp80* aufgerufen wird, unterstützt. So kann sogar mit *netscape* von einem anderen Rechnersystem aus eine *World Wide Web*-Seite in Plan 9 angesteuert werden. Neben *html*-Seiten können natürlich auch über *ftp* Verzeichnisse in *netscape* angezeigt und Daten übertragen werden.

In Plan 9 selbst gibt es zur Darstellung von WWW-Seiten ein Programm namens *mothra*. Anfangs war es fehleranfällig, so daß inzwischen eine verbesserte Version bereitgestellt wurde, die im Update zu bekommen ist.

Für *mothra* benötigt man genau wie bei *netscape* die URL-Notation (*Uniform Resource Locator*). Die Spezifizierung einer Datei besteht in diesem Fall nicht nur aus dem Dateinamen, sondern auch aus dem Methodennamen (*http*, *ftp*, *file*, etc.), dem Rechnernamen (*frigga*, *newage* bei uns), evtl. einer Portnummer (standardmäßig *80* für *html*-Seiten) und aus dem Pfad- und Dateinamen.

```
Methode://Maschine[:Port]/Pfad/Datei
```

In der obigen Abbildung ist mit dem Kommando *g http://thor.informatik.uni-osnabrueck.de/angew.html* ein *html*-Dokument der Osnabrücker Universität geöffnet worden. Mit *s angew.html* läßt sich das Dokument speichern. Später kann mit *mothra file:/usr/none/angew.html* dieselbe Datei wieder geöffnet werden.

In der obersten Leiste ist jeweils eines von vier Kommandos möglich:

g url Seite der *url*-Adresse holen und anzeigen.

s file Seite als Datei abspeichern.

q Programm verlassen.

h/? Hilfe anzeigen.

Das Fenster von *mothra* teilt sich in vier Teile auf. Im obersten Teil befinden sich die aktuelle Adresse und die Kommandozeile, in der die Kommandos eingegeben werden können. Darunter sind die bisher selektierten Dateien aufgelistet. Wird eine Datei mit der mittleren Maustaste selektiert, erscheint ihr Name im *URL*-Format in der obersten Zeile. Erst nachdem der Dateiname mit der linken Taste angeklickt wurde, wird die Adresse tatsächlich angewählt. Nach einiger Zeit erscheint dann die gewünschte Datei im unteren Teil des Fensters. Der Bereich zwischen dem Dokument und der Liste der Dateinamen gibt die momentane Adresse und den Dateinamen an.

Mit der rechten Maustaste erscheint ein Popup-Menü, das bei der älteren *mothra*-Version nur vier und bei einer neueren Version acht Zeilen mit Kommandos enthält. Um wieder zu den vorherigen Dateien zurückzukehren, trägt man eine Markierung mit *save back* in der Datei *$home/lib/mothra/back.html* ein. Danach kann mit *get back* einer dieser Dateinamen erneut ausgewählt werden. Bei der neueren Version gibt es hierfür die Kommandos *save hit* und *hit list*. Die Farben lassen sich mit *fix cmap*, falls nötig, anpassen. Außerdem gibt es *exit*, um das Programm zu verlassen.

15.9 Zusammenfassung

Sowohl mit IL als auch mit TCP lassen sich Verbindungen zwischen verschiedenen Rechnern herstellen. Dabei ist zwar IL in Plan 9 effektiver, aber um zu anderen Betriebssystemen zu gelangen, wird TCP benötigt. Mit Hilfe von *aux/listen* lassen sich die Vorgänge der Netzwerkkommunikation erleichtern, da ein Teil der Verbindung über einfache Shell-Skripten bearbeitet werden kann.

16
Ausblick

Was hat man nun von Plan 9?

Unix sorgte letztlich für eine Revolution, da bis dahin die Betriebssysteme proprietär, inkompatibel und unübersichtlich waren. Neue Rechnergenerationen entstanden plötzlich schneller, die alten Betriebssysteme waren definitiv nicht im Hinblick auf Portierungen entworfen oder implementiert worden, und dann gab es da dieses lästige, kleine System, mit dem begabte Leute angeblich Großes leisteten. Kurz, in den Achtzigern war die kommerzielle Welt reif für einen neuen Anfang.

Eigentlich gilt in den Neunzigern dasselbe: Die Unixe sind nahezu proprietär, inkompatibel und unübersichtlich, und sie beherrschen die heutige Aufgabe — heterogenen Netzbetrieb — eher schlecht als recht. Und wieder gibt es ein neues, kleines System, mit dem begabte Leute gerade diese Aufgabe glänzend meistern. Es gibt aber auch ein lästiges System, mit dem die meisten Leute diese Aufgabe angehen, und die neuen Rechnergenerationen entstehen so, daß sie vorwiegend mit diesem System zurechtkommen. Kurz, die Neunziger sind noch nicht reif für einen neuen Sprung.

Es lohnt sich aber trotzdem, über Plan 9 zu lernen und mit Plan 9 umzugehen. Unix wurde populär, weil es mit Altlasten Schluß machte und die Dinge entkomplizierte: Wer erinnert sich heute noch an *Data Control Blocks*, die *Indexed Sequential Access Method* oder *Partitioned Datasets*, ohne die im guten alten OS/360 der IBM absolut nichts lief, und das war damals der Nabel der Computer-Welt. Vergleicht man heute beispielsweise *dial()* und den Katalog */rc/bin/service* für *aux/listen* mit der Pflege und Fütterung von BSD-Sockets und *inetd.conf*, kommen ähnliche Gefühle auf.

Unix wurde erst nach Jahren gezwungen, mit Sockets und NFS zurechtzukommen. Die Popularität von TCP ist eine logische Konsequenz des Erfolgs der Unix-Pipes, und NFS hat ein bewundernswertes Steh-Auf-Vermögen. TCP ist aber ressourcenintensiv und NFS keineswegs transparent.

Auch bezüglich Sicherheit — Authentifizierung, Integrität und Vertraulichkeit — ist Unix eher ein Spätentwickler. Man muß seinen Super-Usern eben vertrauen, und mindestens in lokalen Netzen dürfte ein Einbruch selten schwieriger sein, als einen Stecker und eine Steckdose zu verbinden. Trennt man sich im Unfrieden von einem Super-User, verschrottet man sein Unix-Netz am besten gleich mit.

Plan 9 macht mit all den Kompromissen Schluß, allerdings um den Preis, auf Altlasten keine Rücksicht zu nehmen. Zu Unix' einfachem Dateimodell, Vektor von Bytes, kommt das Dateisystem zur homogenen Beschreibung sehr vieler, auch exotischer Ressourcen hinzu. Einheitliche, sichere Kommunikation ist der Grundbaustein des Systems, kein nachträglicher Einfall später Architekten. Plan 9 führt vor, wie man aus diesen fundamentalen Konzepten ein komplexes, heterogenes, verteiltes System aufbauen kann. Dateisysteme werden zu prozeß- und damit auch benutzerspezifischen Namensräumen verknüpft, die unabhängig von Rechnern

identisch präsentiert werden können; damit wird wirklich das Netz zum privaten Computer, und der Benutzer muß keine Hardware-Kanten berücksichtigen, egal wo er sich anmeldet.

Diese Lektionen sind *per se* schon interessant und ganz bestimmt wegweisend für die nächste Systemgeneration. Man sollte sie aber auch in existente Systeme übernehmen: Linux besitzt ein *proc*-Dateisystem und entsprechende Techniken zum Umgang mit Prozessen. Auto-montiert man genügend NFS-Systeme, bietet wenigstens ein lokales Netz einer begrenzten Benutzergruppe immer das gleiche Bild. *alex* ist ein FTP-Klient, der per NFS Zugriff auf seine Server ermöglicht. Die GNU-Compiler können cross-compilieren, *rc* gibt es in einer Unix-Implementierung, und *mk* wäre nicht schwer zu portieren; der Ansatz zum heterogenen Arbeiten aus Plan 9 ließe sich durchaus auch auf Unix-Plattformen anwenden, und er ist Techniken wie *imake* haushoch überlegen.

Den vollen Wirkungsgrad erhält man allerdings erst, wenn man alle Bausteine vollständig übernimmt. Plan 9 hat gute Aussichten, sich auf dem nächsten Schauplatz für Marketing-Kriege, nämlich im Bereich untereinander verbundener elektronischer Geräte, durchzusetzen. Hierhin zielt *Inferno*, ein leicht modifizierter Nachfolger von Plan 9, der speziell als Grundlage von Systemen konzipiert ist, bei denen der Endbenutzer höchstens ahnt, daß er eigentlich einen Computer benutzt: Bildtelefone, Boxen für digitales Fernsehen, Spielkonsolen und ähnliche Geräte.

Inferno kann man in einer Demo-Version beziehen, die als Gast unter Systemen wie Windows oder Unix betrieben werden kann. Im Vordergrund steht eine einheitliche Programmiersprache Limbo, die interpretativ und damit Hardware-unabhängig verarbeitet wird, sowie eine Oberfläche auf der Basis von TCL/TK. Wenn man sich aber ein bißchen in das System vertieft, entdeckt man schnell, daß man es hier mit dem Server-Baukasten von Plan 9, einem als Gerät ausgelagerten Sicherheitssystem und einer dafür leicht modifizierten Version von 9P zu tun hat. In Limbo entdeckt man Channels von *alef*, Threads durch Prozeßmanipulation auf der Basis von *rfork()*, Namensraummanipulation mit *bind()* und *mount()*, benannte Pipes mit *#s*, Netzverbindungen mit *dial()* und viele andere Konzepte aus Plan 9 wieder.

Inferno wird als Konkurrent zu Java angesehen. Inzwischen ist Java teilweise zu Inferno portiert, aber der Vergleich zwischen einer Programmiersprache und einer sicheren Kommunikationsplattform war schon zuvor nicht sonderlich sinnvoll. Läßt man Firmenpolitik außer acht, müßte sich Inferno eigentlich als perfekte Unterlage für Java im Bereich der ''unsichtbaren'' vernetzten Systeme erweisen, dank der Baukastenaspekte und Skalierbarkeit von Plan 9. Man kann also durchaus hoffen, daß dort Plan 9 schließlich eine ähnliche Wirkung zeigt wie seinerzeit Unix im Bereich von Timesharing. Wer sich jetzt mit Plan 9 und anschließend Inferno beschäftigt, ist dann fit als Entwickler in dieser Welt.

Literaturverzeichnis

[Bi93] Hans-Peter Bischof, *imake contra I make*, unix/mail 1/93, Carl Hanser Verlag, 1993.

[Bo84] J. F. Maranzano and S. R. Bourne, *A Tutorial Introduction to ADB*, in Bell Laboratories, UNIX Programmer's Manual, Volume 2, Holt, Rinehart and Winston, 1984.

[Du90] Tom Duff, *Rc — A Shell for Plan 9 and UNIX Systems*, The Early Papers, AT&T Bell Labs.

[Du93] Paul DuBois, *Software Portability with imake*, O'Reilly & Associates Inc., 1993.

[Fel83] S. I. Feldman, *Unix Programmer's Manual: Make — A Program for Maintaining Computer Programs*, Holt, Rinehart and Winston, 1983.

[Gre1995] Grey Rudolph, *Ed Wood*, Wilhelm Heyne Verlag München, 1995.

[Pik1995] Rob Pike, et al., *Plan 9 from Bell Labs, Plan 9 — The Documents*, Harcourt Brace & Company, 1995.

[Pik1995] Rob Pike, et al., *Plan 9 from Bell Labs, Plan 9 — The Manuals*, Harcourt Brace & Company, 1995.

[Plan9/1] Plan 9 Web Page, General information on Plan 9, http://plan9.bell-labs.com/plan9/index.html.

[Plan9/2] Plan 9 Web Page, Toronto, General information on Plan 9, http://www.ecf.utoronto.ca/plan9/.

[Sal1994] Peter Salus, *A Quarter Century of UNIX*, Addison-Wesley Publishing Company, 1994.

[Ste92] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

Sachverzeichnis

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z