5 Daemontools

5.1 Konzept der Daemontools

Die Daemontools von Qmail-Author Dan J. Bernstein sind entstanden aus den Erfahrungen mit dem Betrieb der Qmail-Software in verschiedenen Umgebungen. Wie bereits in Abschnitt 4.1.2 kurz dargestellt, haben alle Unix Varianten — und insbesonders jene die nach dem System V bzw. BSD Muster arbeiten — verschiedene Vorstellungen und Vorgaben wie ein "Daemon-Dienst" in den Start des Betriebssystems zu integrieren ist. Hinzukommt die Abhängigkeit von anderen Systemkomponenten. Hierzu zählt in erster Line der Syslog-Dienst sowie ein (fehlender) wohldefinierter und nanosekunden-genauer Zeitstempel für Log-Aktivitäten.

Mit den Daemontools löst Dan Bernstein diese Anforderung durch einen umfangreichen Neuansatz:

- Dienste werden über einen einheitlichen "Supervise" Dienst gestartet (SVC = SuperVise Control) und durch diesen administriert, wobei Standard Unix-Signale genutzt werden.
- Alle Dienste werden überwacht und bei Bedarf nachgestartet.
- Den Diensten wird ein definiertes Environment mit klar vorgebenen Ressourcen (Betriebsmitteln) bereit gestellt.
- Start/Stop-Skripte finden sich nicht wie bislang verstreut im Dateisystem, sondern werden an einer definierten Stelle — nämlich unter `/service/<dienst>/` bereit gestellt, wobei mit `<dienst>` der Name eines Dienstes gemeint ist.
- Es werden Hilfsprogramme zur Verfügung gestellt, mit denen auch "störrische" Dienste, z.B. solche die sich selbst in den Hintergrund stellen, administriert werden können, was ansonsten nicht möglich wäre.
- Die Ausgabe von Informationen auf den Standard Dateideskriptoren werden dem Programm **multilog** (als Co-Prozess) über eine Unix-Pipe zur Verfügung gestellt und dort verlässlich in eine Logdatei mit Nanosekunden genauem Zeitstempel im TAI64N-Format gestellt.

Realisiert wird dieser Ansatz durch die Abbildung und Verankerung der Dienste in einer Verzeichnisstruktur. Stellt das `/proc` Verzeichnis bzw. Dateisystem eine Abbildung für die laufenden Prozesse dar, so gilt dies für das Verzeichnis `/service` unter den Daemontools sinngemäss. Kein moderenes Unix würde heute auf ein `/proc` Dateisystem verzichten; in einigen Jahren wird dies vielleicht auch für `/service` gelten. Viele Daemaons (Apache, Samba, FreeRadius) werden in ihren neuen Releases auf die Anforderungen der Daemontools abgestimmt.

Zusätzlich handelt es sich bei den Daemontools um eine auf allen Unix-Systemen anzuwendenen (kostenlose!) Hochverfügbarkeitslösung für Prozesse.

Seit der Version 0.74 verfolgt der Autor noch einen weiteren Ansatz: Software-Pakete werden in einem "registrierten" Verfahren eingespielt. Dies haben wir bereits in Kapitel 3 vorgestellt.

5.2 Programme der Daemontools

Ich möchte hier und im folgenden die Daemontools in drei Kategorien einteilen und auf diese auch entsprechend eingehen:

1. **Supervise-Tools**: Start- und Monitorprogramme für Daemonprozesse.
   - **svscanboot** — Programm zum Starten von **svscan** mit definierter Umgebung
   - **svc** — Kommandointerface zum Starten/Stoppen/Administrieren von Programmen
   - **svscan** — Hauptprogramm zum automatischen Start und zur Überwachung der im Verzeichnis `/service/` hinterlegten Dienstprogramme (in jeweils eigenen Verzeichnissen)
   - **svstat** — Hilfsprogramm zur Anzeige des Status eines unter svcscan gestarteten Programms
   - **supervise** — Hauptprogramm zum Starten und Monitoren einzelner Prozesse
   - **svok** — Hilfsprogramm zur Überprüfung, ob **supervise** läuft
   - **fghack** — "foregroundhack" Programm, die sich selbst in den Hintergrund stellen, verbleiben unter Anwendung von **fghack** im Vordergrund
   - **pgrphack** — Hilfsprogramm zum Ausführen eines anderen Programms in einer separaten Prozessgruppe
2. **Logging-Tools**: Programme zum Aufzeichnen und Auswerten von Log-Informationen.
   - **multilog** — Hauptprogramm zum Schreiben der Log-Informationen eines mit einer Pipe verbundenen Programms
   - **readproctitle** — Hilfsprogramm zum Hinterlegen einer Log-Information

im Speicher, die mittels des Progamms **ps** ausgelesen werden kann

3. **Daemon-Helper**: Programme zur Ressourcenkontrolle von Daemon-Prozessen.
   - **envdir** — Programm zum Ausführen eines anderen Programms in einem definierten Environment
   - **envuidgid** — Programm zum Ausführen eines anderen Programms unter dem Environmet eines definierten User bzw. einer Group
   - **setlock** — Programm zum Sperren und exklusiven Schreiben auf eine Datei von einem nachgeordneten Programmes aus
   - **setuidgid** — Ausführen eines Programms unter einem definierten User/Group
   - **softlimit** — Ausführen eines Programms mit definierten Ressourcen-Grenzen
4. **Zeit-Tools**: Programme zur Erzeugung eines TAI64N-Zeitstempels.und Ausgabe in leserlicher Form
   - **tai64n** — Programm zur Erzeugung eines TAI64N-Zeitstempels
   - **tai64nlocal** — Hilfsprogramm zum on-the-fly Umwandeln des TAI64N-Datumsstempels in ein ISO Zeitformat
   - **tai64nfrac** — Hilfsprogramm zur Ausgabe des TAI64N-Zeitformats in Sekunden- und Sekundenbruchteilen

5.3 Installation der Daemontools

Die aktuelle Version 0.76 der Daemontools liegt als Tar-Archiv vor: daemontools-0.76.tar.gz (erreichbar über die URL `http://cr.yp.to/daemontools.html`). Zustätzlich gibt es hierzu angepasste man-pages von Gerrit Pape (<pape@smarden.org>): daemontools-0.76-man.tar.gz (`http://smarden.org/pape/djb/`)

Dan Bernstein empfiehlt zur Installation "registrierter" Software zunächst unter dem Root-Verzeichnis "`/`" ein High-Level Verzeichnis `/package/` zu erzeugen und hierin die Datei `daemontools-0.76.tar.gz` zu kopieren und zu entpacken.

```
# mkdir -p /package
# chmod 1755 /package
# cd /package
# tar -xzpf /path_to_file/daemontools-0.76.tar.gz
```

```
# cd admin/daemontools-0.76
```

Wichtig ist beim ent-**tar**-en des daemontools-Archiv die Angabe der Option "-p". Hierdurch werden die entpacken Dateien und Verzeichnisse mit den gleichen Attributen und Rechten wie die Originale erzeugt. Es wird ein Verzeichnis `./admin/` erstellt, das als Sammelbecken für System-Administrations-Software dient (zukünftige Qmail-Versionen werden über ein Unterverzeichnis `./mail/` installiert). Anschliessend kann das Tar-File gelöscht werden und es wird in das Verzeichnis `./admin/daemontools-0.76` (das durch das Entpacken des Tar-Archivs erzeugt wurde) gewechselt.

Durch Aufruf des Skripts

```
#  package/install
```

werden die neuen Daemontools im Verzeichnis `/package/admin/daemontools` generiert und stehen als Symlinks im Verzeichnis `/command/` zur Verfügung:

```
% ls -la /command

drwxr-xr-x   2 root  wheel   512 Nov 30  2001 .

drwxr-xr-x  24 root  wheel  1024 Sep 26 03:12 ..

lrwxr-xr-x   1 root  wheel    41 Nov 26  2001 envdir ->
/package/admin/daemontools/command/envdir

lrwxr-xr-x   1 root  wheel    44 Nov 26  2001 envuidgid ->
/package/admin/daemontools/command/envuidgid

lrwxr-xr-x   1 root  wheel    41 Nov 26  2001 fghack ->
/package/admin/daemontools/command/fghack

lrwxr-xr-x   1 root  wheel    43 Nov 26  2001 multilog ->
/package/admin/daemontools/command/multilog

lrwxr-xr-x   1 root  wheel    43 Nov 26  2001 pgrphack ->
/package/admin/daemontools/command/pgrphack

lrwxr-xr-x   1 root  wheel    48 Nov 26  2001 readproctitle
-> /package/admin/daemontools/command/readproctitle

lrwxr-xr-x   1 root  wheel    42 Nov 26  2001 setlock ->
/package/admin/daemontools/command/setlock

lrwxr-xr-x   1 root  wheel    44 Nov 26  2001 setuidgid ->
/package/admin/daemontools/command/setuidgid

lrwxr-xr-x   1 root  wheel    44 Nov 26  2001 softlimit ->
/package/admin/daemontools/command/softlimit

lrwxr-xr-x   1 root  wheel    44 Nov 26  2001 supervise ->
/package/admin/daemontools/command/supervise

lrwxr-xr-x   1 root  wheel    38 Nov 26  2001 svc ->
/package/admin/daemontools/command/svc
```

```
lrwxr-xr-x   1 root  wheel    39 Nov 26  2001 svok ->
/package/admin/daemontools/command/svok

lrwxr-xr-x   1 root  wheel    41 Nov 26  2001 svscan ->
/package/admin/daemontools/command/svscan

lrwxr-xr-x   1 root  wheel    45 Nov 26  2001 svscanboot ->
/package/admin/daemontools/command/svscanboot

lrwxr-xr-x   1 root  wheel    41 Nov 26  2001 svstat ->
/package/admin/daemontools/command/svstat

lrwxr-xr-x   1 root  wheel    45 Nov 30  2001 tai64nfrac ->
/package/admin/daemontools/command/tai64nfrac

lrwxr-xr-x   1 root  wheel    46 Nov 26  2001 tai64nlocal ->
/package/admin/daemontools/command/tai64nlocal
```

Zusätzlich werden im Verzeichnis `/usr/local/bin/` Symlinks erzeugt, sodass eine bestehende Installation der Daemontools im Kompatibilitätsmodus weiter betrieben werden kann.

```
lrwxr-xr-x   1 root  wheel       18 Nov 26  2001 supervise -
> /command/supervise

lrwxr-xr-x   1 root  wheel       12 Nov 26  2001 svc ->
/command/svc

lrwxr-xr-x   1 root  wheel       13 Nov 26  2001 svok ->
/command/svok

lrwxr-xr-x   1 root  wheel       15 Nov 26  2001 svscan ->
/command/svscan

lrwxr-xr-x   1 root  wheel       19 Nov 26  2001 svscanboot
-> /command/svscanboot

lrwxr-xr-x   1 root  wheel       15 Nov 26  2001 svstat ->
/command/svstat

lrwxr-xr-x   1 root  wheel       15 Nov 26  2001 tai64n ->
/command/tai64n

lrwxr-xr-x   1 root  wheel       20 Nov 26  2001 tai64nlocal
-> /command/tai64nlocal
```

Diese Vorgehen hat gegenüber der herkömmlichen Installation erhebliche Vorteile, da

- nicht mehr der Anwender, sondern der Entwickler die Verantwortung für einen sauberen Release-bzw. Versionswechsel trägt, indem die Sourcen der Pakete immer mit vollständigem Revision-Level in die gleiche Umgebung eingestellt wird und die aktuelle Version als Symlink des Basisnames erzeugt wird, was auch ein Downgrading ermöglicht,
- die Pakete nach ihrer Aufgabenstellung klassifiziert werden, und es so trotz ggf. gleicher Namenskonvention der Basispakete nicht so leicht zu

Überschneidungen kommt (*name qualification*),

- durch diesen Mechanismus erstmals ein einfacher, konsistenter und flexibler Installationsmechanismus gewährleistet werden kann.

```
% cd /package
% ls -la *
drwxr-xr-t  3 root  wheel  512 Nov 26 17:49 .
drwxr-xr-t  3 root  wheel  512 Nov 26 17:47 ..
lrwxr-xr-x  1 root  wheel   16 Nov 26 17:49 daemontools ->
daemontools-0.76
drwxr-xr-x  6 root  wheel  512 Nov 26 17:49 daemontools-0.76
```

Installation nach `/package/`

Aufgrund dieser stark "verlinkten" Struktur ist es nicht ratsam, die Daemontools in einem anderen Verzeichnis ausser `/package/` zu installieren. Die notwendigen ausführbaren Dateien verblieben in diesem Installationsverzeichnis, und werden nicht — wie sonst üblich per **install**-Kommando — in die Zielverzeichnisse kopiert. Gewohnheitsmässig hatte ich einmal mein Installations-Defaultverzeichnis `/usr/local/src/` zur Installation der Daemontools ausgewählt. Das hat natürlich ebenfalls geklappt — so lange, bis ich der Meinung war, das erzeugte `./admin/` Verzeichnis zu löschen ... In dem Moment standen die wichtigen Daemontool-Links quasi "nackt" da.

Abschliessend ist zu erwähnen, dass durch das Installationsskript auch das Verzeichnis `/service` mit den notwendigen Rechten angelegt wird.

5.3.1 Ergänzendes Programm: tai64nfrac

Zur Auswertung des **qmail-send** Logfiles mittels des Programmpakets Qmailanalog wird zusätzlich das Programm **tai64nfrac** benötigt. **tai64nfrac** gibt im Gegensatz zu **tai64nlocal** die "verflossene" Zeit in Sekunden an. Dieses von Russ Allbery (rra@stanford.edu) entwickelte Programm steht in Form einer E-Mail auf der Home-Page von Qmail (`http://www.qmail.org/tai64nfrac`) und muss zunächst mit einem Editor von den überflüssigen Teilen befreit werden. Hierdurch lässt sich die zu kompilierende C Quelldatei `tai64nfrac.c` extrahieren, die in das Verzeichnis `/package/admin/daemontools-0.76/src/` kopiert wird. Dieser und die weiteren Schritt lauten wie folgt:

```
# cp tai64nfrac.c /package/admin/daemontools-0.76/src/
# cd /package/admin/daemontools-0.76/compile/
# ln -s  src/tai64nfrac.c tai64nfrac.c
# make -f Makefile tai64nfrac
# install tai64nfrac ../command/tai64nfrac
```

```
# cd /command
# ln -s /package/admin/command/tai64nfrac tai64nfrac
# cd /usr/local/bin
# ln -s /commmand/tai64nfrac tai64nfrac
```

5.3.2 Die man-pages

Zunächst sollte festgestellt werden, wie der Default-Pfad für man-pages im System lautet; normalerweise `/usr/share/man/` oder `/usr/local/man/`. Ich gehe im weiteren von `/usr/share/man/` aus.

Das Tar-Archiv daemontools-0.76-man.tar.gz wird in das Verzeichnis `/package/admin/` kopiert und entpackt. Hierdurch wird ein Verzeichnis `./daemontools-man` erzeugt, in das gewechelt wird.

Anschliessend ist (entsprechend der Datei README) wie folgt zu verfahren:

```
# gzip *.8 ; cp *.8.gz /usr/share/man/man8/
```

Hiermit sind die man-pages installiert.

5.4 Aufsetzen der Supervise-Tools

Nach dem Neustart des Rechners wird über das Skript-Skript **svscanboot** der **svscan**-Prozess für das Verzeichnis `/service/` initialisiert. Dieser wiederum überprüft im Fünf-Sekundenrythmus die Untervzeichnisse von `/service/` und startet für jedes gefundene Unterverzeichnis den Prozess **supervise**. Das Starten und Monitoren (d.h. auch der Restart) sowie das Logging von Diensten geschieht hierdurch "fast wie von selbst"; vorausgesetzt

- für jeden Dienst (Daemon) wird ein Verzeichnis erstellt, das typischerweise den Namen des von **svscan** zu administrierenden Programmes (=Daemons) trägt,
- in dieses Verzeichnis wird ein geeignetes Start-Skript des Programms eingestellt, das immer den Namen `run` trägt. Im Gegensatz zu den z.B. unter Linux typischen Run-Level-Skripten benötigt das `run`-Skript keine eigene Logik. Es muss nur Sorge getragen werden, dass das zu administrierende Programm mit den richtigen Pfaden über ein **exec**-Statement aufgerufen, sowie über das notwendige Environment verfügt. Hierunter werden beispielsweise die User/Group-Rechte sowie Unix-Ressourcen verstanden. Dies kann z.B. mittels der Daemontools Hilfsprogramme erfolgen.
  Das **exec**-Statement sorgt dafür, dass das aufgerufene Programm (mit dem nun definierten Environment) die über das `run`-Skript gestartete Shell ersetzt. Erst hierdurch ist es möglich, per **supervise** den eigentlichen

Daemon-Prozess zu managen und zu monitoren.

Sollte das Programm einen Logfile schreiben, der vom Daemontools-Programm **multilog** weiter zu verarbeiten ist, muss ein Unterzeichnis **./log/** erstellt werden. In dieses Verzeichnis ist ebenfalls ein Skript namens **run** zu stellen, dass in Verbindung mit dem Programm **multilog** die Ausgabe des eigentlichen Programms über den File-Descriptor 1 entgegen nimmt und sicher in einen **multilog**-formatierten Logfile schreibt.

Es ist zu beachten, dass der zu administrierende Daemon-Prozess mit dem **multilog**-Logfile-Prozess über eine Unix-Pipe verbunden ist und nun sowohl der "Haupt"- als auch der "Co"-Prozess von **svscan** überwacht werden.
Das Schreiben eines **multilog**-Logfiles greift aber nur dann, wenn das Hauptprogramm nicht den Standard-Unix Syslog-Dienst nutzt, sondern auf den File-Descriptor 1 schreibt. Dies muss notfalls durch eine Anpassung des Hauptprogramms erreicht werden.

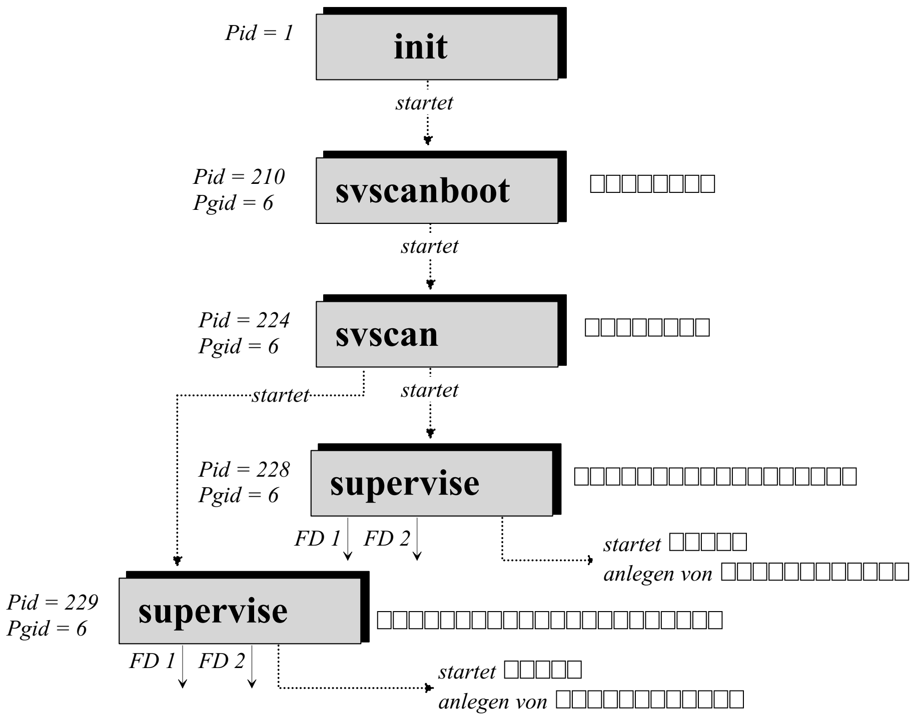

Abbildung 5-1: Verkettung der Supervise-Tools. FD = File Descriptor. ***init*** *ist der Unix-Basisprozess; Pid = Process Id, Pgid= Process Group Id. Die zugewiesenen Zahlenwerte sind systemabhängig.*

Dieser Zusammenhang ist in Abbildung 5-1 dargestellt. Während uns die Daemontools die Freiheit geben, das eigentliche Startverzeichnis — in der Regel `/service/` — sowie die Namen der Dienste über die Benennung der Unterverzeichnisse vorzugeben, herrscht bei der Benamung der Startskripte (also `./run`) und bei der des Log-Verzeichnisses strenge Sitte: Ein Abweichen ist nicht gestattet. Die Gründe hierfür sind einsichtig:

- Durch die Wahl des möglichen Service-Verzeichnisses eignet sich Supervise auch, um "private" Daemons z.B. im `$HOME`-Verzeichnis aufzusetzen, sofern keine Daemon-spezifische Beschränkung besteht, den Dienst auch unter einer beliebigen User-Kennung auszuführen.
- Die Festlegung des Names des eigentlichen Daemon-Startskriptes (also `./run`) ermöglicht es, schnell zwischen neuen und alten Versionen zu wechseln. Hierauf will ich später noch ausführlicher eingehen.

Doch zunächst soll geklärt werden, wie die Supervise-Tools etwas effizienter in den System-Startup integriert werden können.

5.4.1 svscanboot

Nach einigen Erfahrungen (und Versionen der Daemontools) hat Dan Bernstein es für nötig befunden, den Aufruf von **superivse** "konzeptionell" richtig in ein Boot-Skript einzubinden: **svscanboot**.

Vorteilhaft ist, dass nun ein definiertes und eindeutiges Boot-Skript für die Supervise-Tools vorliegt und Systemabhängigkeiten nach korrekter Einbettung des Skripte **svscanboot** reduziert sind, bzw. komplett enfallen. Wir schauen uns einfach einmal an, wie **svscanboot** aufgesetzt ist:

```
PATH=/command:/usr/local/bin:/usr/local/sbin:/bin:/sbin:/usr
/bin:/usr/sbin:/usr/X11R6/bin

exec </dev/null

exec >/dev/null

exec 2>/dev/null

/command/svc -dx /service/* /service/*/log

env - PATH=$PATH svscan /service 2>&1 | \

env - PATH=$PATH readproctitle service errors:
..........(400 Punkte)
```

*Listing 5-1: Das supervise Boot-Skript **svscanboot**, Leerzeilen und der grösste Teil der vierhundert "." wurden unterdrückt.*

Folgende Schritte sind erkennbar:

- Es wird der Default-Path gesetzt; hierdurch können Kommandos bzw. Programme, die sich in den eingebunden Verzeichnissen befinden, bequem aufgerufen werden.
- Das Environment (mit Ausnahme des gesetzten Pfades) sowie die Standardein- und -ausgabe und die Fehlerausgabe werden "geputzt".
- Eventuell laufende **supervise**-Prozesse werden (dauerhaft) gestoppt.
- **svscan** wird für das Verzeichnis `/service` gestartet und STDOUT sowie STDERR auf den Prozess **readproctitle** umgelenkt.

5.4.2 System V Unix

Das Daemontools Installationsskript übernimmt für uns den Eintrag des Starts von **svcanboot** in die `/etc/inittab`:

```
Adding svscanboot to inittab...
```

Das unter `/package/admin/daemontools/package` liegende Installationsskript **install** beinhaltet drei Abschnitt:

1. `package/compile` — Kompilieren der unter `../src/` liegenden c-Dateien.
2. `package/upgrade` — Löscht das bestehende `./daemontools/` Verzeichnis und legt dies als einen Symlink der aktuellen Version neu an, erzeugt die Symlinks der ausführbaren Dateien in `/command/` sowie von da aus nach `/usr/local/bin/`.
3. `package/run` — Testen, ob bereits eine Daemontools-Installation vorliegt, Erzeugen des `/service/` Verzeichnisses und Eintrag in die System-Bootskripte.

Im `run`-Schritt wird untersucht, ob die Datei `/etc/inittab` vorliegt. Ist dies der Fall, liegt ein Unix System V vor, ansonsten wird ein BSD-System angenommen. Im ersten Fall die Datei `inittab` modifiziert:

```
#!/bin/sh -e

if grep svscanboot /etc/inittab >/dev/null

then

  echo ’inittab contains an svscanboot line. I assume that
svscan is already running.’

else

  echo ’Adding svscanboot to inittab...’

  rm -f /etc/inittab’{new}’

  cat /etc/inittab package/boot.inittab >
/etc/inittab’{new}’

  mv -f /etc/inittab’{new}’ /etc/inittab

  kill -HUP 1

  echo ’init should start svscan now.’

fi
```

Die Datei `boot.inittab` hat folgenden Inhalt:

```
SV:123456:respawn:/command/svscanboot
```

Dies wird an das Ende der `inittab` eingetragen. Entsprechend der Syntax `id:runlevels:action:process` wird der Prozess mit "SV" beschrieben, in die Runlevels 1 bis 6 eingeklinkt und über den **respawn**-Befehl bei Bedarf neu gestartet.

Runlevel und `svscanboot`

Die Bedeutung der Runlevels haben wir bereits in Abschnitt 2.x kennengelernt. Die Vorgaben, die Dan Bernstein für die Runlevels für **svscanboot** gemacht haben, sind dann sinnvoll, wenn über **svc** Dienste in allen Runlevels gestartet werden sollen. In allen anderen Fällen ist es ratsam, durch Editieren der Datei `/etc/inittab` die Runlevels 1 (Single-User Mode) sowie vor allem 6 (System-Shutdown) zu entfernen. Speziell letzteres haben des öfteren Slackware-Linux Admins schmerzlich erfahren müssen, weil andernfalls das **umount** des (ext2) Root-Filesystems mit einer Fehlermitteilung begleitet, dass dieses "busy" sei.

Werden hingeben über die Daemontools lediglich Netz-Dienste wie Qmail, **sshd** oder **dnscache** gestartet, kann die Runlevel-Vorgabe weiter eingeschränkt werden so dass effektiv **svscanboot** lediglich in den Runlevels 3, 4 und 5 gestart wird. Welche Runlevel tatsächlich genutzt werden und welche Bedeutung sie haben, muss individuell per Aufruf von `man inittab` herausgefunden haben.

svc und **svscanboot** sind im Prinzip Runlevel-"blind", d.h. man kann nicht vorgeben, dass zwar **svc** für alle Runlevel zu initialisieren ist, aber nur bestimmte Dienste — z.B. beim Wechel des Runlevels per Aufruf von **init** — ausgeführt werden sollen. Allerdings kann man sich eines effizienten Kniffs bedienen, der hier vorgestellt werden soll:

- Ausgangsüberlegung ist, dass **svc** nur dann einen Dienst (im folgenden `<dienst>` genannt) startet, wenn im Verzeichnis `/service/<dienst>/` keine Trigger-Datei `down` existiert.
- Wir gehen davon aus, dass die Skripte zur Steuerung der Runlevel-Dienste in `/etc/rc.d/` zu finden sind.
- Wir erzeugen uns dort eine Skriptdatei **<dienst>** mit folgendem Inhalt:

```
#!/bin/sh

Runlevel start script for <dienst>.

<dienst> is under control of svc and will only be executed
in specific run levels.

#

SERVICE=/service/<dienst>

LOG=/service/<dienst>/log

case "$1" in

       start)

              rm -f $SERVICE/down 2>/dev/null

              rm -f $SERVICE/down 2>/dev/null

              svc -u $SERVICE/run
```

```
                svc -u $LOG/run
                ;;
        stop)
                svc -d $SERVICE/run
                touch $SERVICE/down
                svc -d $LOG/run
                touch $SERVICE/down
                ;;
        status)
                if test -f $SERVICE/down
                        echo "Service $SERVICE should be down."
                else
                        echo "Service $SERVICE is controlled by
svc."
                fi
                svstat $SERVICE
                if test -f $LOG/down
                        echo "Service $LOG should be down."
                else
                        echo "Service $LOG is controlled by
svc."
                fi
                svstat $LOG
                ;;
        *)
                echo "Usage: $0 {start|stop|status}"
                exit 1
                ;:
esac
exit 0
```

Listing 5-2: System V Runlevel-Skeleton Skript für Dienste unter ***svc****.*

- In diesem Skript ist natürlich der Verzeichnisnames des Dienstes `<dienst>` unter dem Verzeichnis `/service/` ersetzen. Ggf. sind die Einträge für den zusätzlichen Logging-Dienst zu entfernen.

- Diese Datei wird anschliessend mit einem geeigneten Präfix (vgl. Abschnitt 2.x) in die Runlevel-Verzeichnisse `rcXY.d` gelinkt.
- Mittels der System V gemässen Start/Stop-Initialisierung von Diensten durch die Abarbeitung der in den ./`rcXY.d/` Verzeichnissen hinterlegten Skripte, lässt sich die gewünschte Kombination von **svc** und Runlevel-Logik erzielen.

5.4.3 BSD-Systeme

Das Installationsskript der Daemontools erkennt BSD-Systeme und nutzt die Datei `/package/admin/daemontools/package/boot.rclocal` um `/etc/rc.local` mit folgendem Statement zu bevölkern:

```
csh -cf '/command/svscanboot &'
```

Typischerweise wird die Datei `rc.local` über `/etc/rc` gestartet, deren Abarbeitung über die Konfigurationsdatei `/etc/rc.conf` (`man rc.conf`) gesteuert wird.

Auf die Rolle des BSD-Startskriptes `/etc/rc` bin ich schon in Kapitel 2 eingegangen. Ich persönlich bin kein Anhänger einer `rc.local` Datei; hier wird eine weitere Ebene in der Start-Up Abarbeitung eingefügt, die das chaotische BSD-Konzept noch undurchschaubarer macht. Obwohl die z.B die FreeBSD-Dokumentation davon abrät, edititiere ich die `rc`-Datei und füge obiges Statement einfach an Stelle des (für unsere Zwecke) überflüssigen Teils "`mta_start_script`" ein. Um zudem ein wenig mehr Kontrolle empfehle ich, in der Konfigurationsdatei `/etc/rc.conf` noch folgenden Eintrag vorzunehmen:

```
svc_enable="YES"
```

und das `rc`-Startskript sieht dann bei mir in Auszügen folgendermassen aus:

```
# Now start up miscellaneous daemons that don't belong
anywhere else

#

echo -n 'Starting standard daemons:'

case ${inetd_enable} in

[Nn][Oo])

        ;;

*)

        echo -n ' inetd'; ${inetd_program:-/usr/sbin/inetd}
${inetd_flags}

        ;;

esac
```

```
case ${cron_enable} in
[Nn][Oo])
        ;;
*)
        echo -n ' cron';        ${cron_program:-
/usr/sbin/cron} ${cron_flags}
        ;;
esac
case ${lpd_enable} in
[Yy][Ee][Ss])
        echo -n ' printer';     ${lpd_program:-
/usr/sbin/lpd} ${lpd_flags}
        ;;
esac
case ${sshd_enable} in
[Yy][Ee][Ss])
        if [ -x ${sshd_program:-/usr/sbin/sshd} ]; then
                echo -n ' sshd';
                ${sshd_program:-/usr/sbin/sshd}
${sshd_flags}
        fi
        ;;
esac
case ${usbd_enable} in
[Yy][Ee][Ss])
        echo -n ' usbd';        /usr/sbin/usbd ${usbd_flags}
        ;;
esac
case ${svc_enable} in
[Yy][Ee][Ss])
        echo -n ' svc';         /command/svscanboot &
        ;;
esac
        ;;
echo '.'
```

Jetzt steht **supervise** gleichberechtigt neben den Diensten **cron** und **lpd**. Wir werden weiter unten sehen, dass es Sinn macht, auch den SSH-Dienst unter **supervise**-Kontrolle zu stellen und im letzten Schritt ganz auf den **inetd** als "Superdaemon" zu verzichten. Hierdurch wird natürlich auch diese Einträge in der Datei `rc` überflüssig, was aber durch die per `/etc/rc.conf` deklarierten Aufrufe-Flags keine Rolle spielt. Das über den rc-Mechanismus per Environment zur Verfügung stehende svc-Flag kann allerdings auch dann genutzt werden, falls der Aufruf von **svscanboot** über die Datei `/etc/rc.locals` erfolgt.

5.5 Prozess-Steuerung und -Überwachung mit den Supervise-Tools

Bei der Prozess-Steuerung und -Überwachung geht Dan Bernstein mit seinen Supervise-Tools von folgenden Paradigmen aus:

- Der Prozess darf nicht im Hintergrund laufen, d.h. sich weder automatisch im Hintergrund stellen, noch über das &-Zeichen dazu angewiesen werden.
- Der Prozess muss sich mit Standard-Unix-Signalen administrieren lassen.
- Bei einem Stop darf der zu überwachende Prozess nicht zusätzlich die Prozesse seiner Prozessgruppe beenden.
- In der `run`-Datei des zu überwachenden Dienstes darf nur genau ein Daemon-Prozess beinhaltet sein.

Um diese Zusammenhänge genauer zu verstehen, konsultieren wir noch einmal Abbildung 5-1 und stellen folgendes fest:

1. Wir sehen, dass hier **init** (Pid=1) **scanboot** (Pid=210) startet. Von diesem aus wird für das Verzeichnis `/service/` **svscan** (Pid=224) aufgerufen. **svscan** ist verantwortlich für die unter `/service/` stehenden Unterverzeichnis, die jeweils für einen Dienst stehen und nun per **supervise** (Pid=228) gemonitort werden. Existiert ein `./log/` Verzeichnis und dort die generische Datei `./run`, so wird ein weiterer **supervise** Prozess für den Co-Daemon gestartet (Pid=229) und beide über eine Pipe verbunden.

2. In den `run`-Skripten erfolgt der Aufruf des zugeordneten Daemons bekanntlich durch das **exec**-Statement. Hierdurch ersetzt der aufgerufene Prozess die ihn aufrufende Shell. **supervise** monitort aber notwendigerweise die Daemons über die Pid des ursprünglichen `run`-Skriptes. Es lassen sich zwar mehrere Prozesse über das `run`-Skript starten; der **supervise**-Kontrolle ist aber nur der durch das **exec**-Statement zugewiesene Initialprozess unterzogen. Wurde andererseits der **exec**-Aufruf "vergessen", wird zwar der zugehörige Daemon gestartet; er lässt sich aber nicht mehr per **svc**-Aufruf

stoppen, sondern muss traditionell "gekilled" werden.

3. Eine ähnliche Problematik liegt bei Prozessen vor, die unbedingt im Hintergrund ablaufen wollen (z.B. der **inetd**). Hierdurch wird in Folge eine neue Prozess-Id vergeben, sodass ebenfalls ein Monitoring bzw. Stoppen des Daemons nicht mehr möglich ist. Als "Workaround" hat uns Dan Bernstein hierfür das Programm **fghack** spendiert.

4. Als weitere Konsequenz dieser Prozessabhängigkeiten ergibt sich, dass innerhalb der `run`-Skripte keine Unix-Pipes eingesetzt werden sollen:

```
#!/bin/sh

exec process_1  | process_2
```

   Dies bedeutet natürlich ebenfalls eine "Mehr-Prozess"-Architektur, die über **supervise** nicht sinnvoll abgewickelt werden kann.

5. Letztlich muss festgehalten werden, dass alle per **svcanboot** initierten Prozesse — einschliesslich der gestarteten Daemons und evtl. des **multilog**-Prozesses — eine einheitliche Prozessgruppe (Pgid) bilden. Hierdurch ist gewährleistet, das mittels der Standard-Unix-Signale z.B. **svc** den zu monitorenden Prozess auch steuern kann, d.h. starten und beenden. Dieses Konstrukt macht die Prozesssteuerung auch unabhängig davon, ob **svscan** von *root* oder von einem beliebigen User gestartet wird.

6. Einige Daemons (Dan Bernstein führt in seinen Erläuterungen `http://cr.yp.to/daemontools` den **ppd** aus) wollen bei Beendigung die gesamte Prozessgruppe mit sich ziehen, was in Konsequenz das "killen" des gesamten Supervise-Konstrukts bedeutet. Zwar könnten diese Dienste in einem eigenen (`./service`-)Verzeichnis eingebunden werden und somit isoliert überwacht werden; als Alternative bietet sich aber das Hilfsprogramm **pgrphack** an, dass speziell für die störrischen Daemons eine neue Prozessgruppe initialisiert.

7. In allen diesen Fällen sind wir davon ausgegangen, dass **svscanboot** über **init** ins Leben gerufen wird und dass **supervise** hiervon abgeleitet die Prozesse (Daemons) initiiert kontrolliert. Es ist aber auch möglich (bei entsprechenden Rechten) **svscanboot** von einem dezidierten Account zu starten. Dann können mehrere parallele **supervise**-Environments in unterschiedlichen Programmgruppen existieren.

5.5.1 svscan

svscan ist das Monitoring-Programm für das Defaultverzeichnis `/service/` und wird seit den Daemontools 0.75 über das Skript **svscanboot** gestartet. Das aktuelle Arbeitsverzeichnis kann beim Aufruf angegeben werden:

```
svscan [ Verzeichnis ]
```

Die Aufgaben von **svscan** lauten wie folgt:

- Für jedes Unterverzeichnis von `/service/` — im folgenden <dienst> genannt — startet **svscan** den **supervise**-Prozess: `supvervise /service/<dienst>`. Dies gilt für maximal 1000 Unterverzeichnisse von `/service/` und falls deren Namen nicht 255 Byte Länge überschreitet. Beginnt ein Unterverzeichnisname mit einem Punkt (z.B. `/service/.<dienst>/`), wird es übergangen.

- Existiert hierunter ein Verzeichnis `./log/`, wird auch hierfür **supervise** initialisiert: `supervise /service/<dienst>/log/`. Beide Prozesse sind durch eine Pipe verbunden, sodass **svscan** zwei freie Datei-Deskriptoren benötigt.

- **svscan** überprüft im 5-Sekunden-Rhythmus das Verzeichnis `/service/` und verfährt wie oben, falls ein neues Unterverzeichnis hinzugekommen oder falls ein **supervise**-Prozess abbrach. War dies für das Verzeichnis `./log/` der Fall ist, übernimmt **svscan** die alte Pipe, sodass Datenverluste vermieden werden.

- Beim Auftreten von Fehlern, insbesonders dann, wenn zwar z.B. ein Verzeichnis `<dienst>` erzeugt wurde, dieses aber noch kein `run`-Skript beinhaltet, meldet dies **svscan** an die Standard-Fehlerausgabe (STDERR) weiter. Diese Ausgabe wird in der Regel von **readproctitle** entgegen genommen (siehe weiter unten).

svscan realisiert diese Aufgaben, indem es Pointer und Zustände der untergeordneten **supervise**-Prozesse in einem Vektor hält, der maximal 1000 Prozesse (und Co-Prozesse) enthalten darf. Innerhalb dieses Vektors werden für die Prozesssteuerung und Buchführung die Dateideskriptoren der Daemons und ihrer Co-Prozessen (den Programmen, die über `./log/run` angesprochen werden) hinterlegt. Daemon und Co-Prozess werden auf diese Weise über ein Pipe mit je zwei Datei-Deskriptoren gekoppelt. Bei vielen Daemons sorgt dieses Verfahren für einen verheblichen "Verbrauch" an Datei-Deskriptoren.

5.5.2 supervise

Das Programm **supervise** ist das eigentliche Arbeitspferd im Stall der Supervise-Tools. **supervise** startet und monitort einen Dienst im Verzeichnis `<dienst>/`, typischerweise also

```
supervise /service/<dienst>
```

Arbeitsweise:

- **supervise** wechselt ins Verzeichnis `[/service/]<dienst>/` und ruft automatisch das hierin liegende Skript `./run` auf; falls dieses existiert.

- Sollte `./run` nicht exisitieren, oder sich beenden, wartet **supervise** eine Sekunde vor dem nächsten Versuch.
- Stellt **supervise** Fehler beim Start von `<dienst>` fest, gibt es diese auf der Standardausgabe (STDOUT) aus; die in der Regel an **readproctitle** mit einer Unix-Pipe verbunden ist.
- **supervise** empfängt die Signale vom Kommando-Programm **svc**, um Dienste zu starten oder zu beenden.
- Exisitert im Arbeitsverzeichnis eine Datei `./down`, so startet **superivse** das Skript `./run` nicht automatisch (vgl. Listing 5-2), sondern nur per expliziter Anweisung über **svc**.
- Statusinformationen über den zu überwachenden Daemon hinterlegt **supervise** in einem (lese-)geschützten Verzeichnis `[/service/]<dienst>/supervise/`. Diese Informationen können von **svstat** ausgelesen werden. **supervise** macht sich hierbei den *selfpipe* Trick zu Nutze, den Dan Bernstein auf einer Webseite bschrieben hat (`http://cr.yp.to/selfpipe.html`).

supervise ist als "ever lasting" Daemon ausgelegt, d.h. so lange `[/service/]<dienst>/` exisitiert, ist **supervise** — einmal gestartet — für dieses Verzeichnis aktiv. Zwei Ausnahmen sind vorgesehen:

1. **supervise** endeckt, dass bereits ein anderer **supervise**-Prozess `<dienst>` monitort.
2. **supervise** für `<dienst>` wird samt dem eigentlichen `<dienst>` durch Aufruf von `svc -x` beendet.

Wir betrachten das von **supervise** geschaffene Verzeichnis `./<dienst>/supervise/`, in dem Statusinformationen hinterlegt werden:

```
# ls -la supervise
prw-------  1 root  wheel    0 Oct 24 23:59 control
-rw-------  1 root  wheel    0 Sep  6  2001 lock
prw-------  1 root  wheel    0 Sep  6  2001 ok
-rw-r--r--  1 root  wheel   18 Oct 24 23:59 status
```

Ins Auge fällt zunächst, dass die Dateien `lock` und `ok` ein altes Datum tragen; die Dateien `control` und `status` jedoch wesentlich neueren Datums sind. Grund hierfür ist, dass sowohl `lock` als auch `ok` beim ersten Start von **supervise** angelegt werden, und dann nicht mehr verändert werden müssen, während `control` und `status` für den aktuellen Status relevant sind.

Bei jedem (Neu-)Start von **supervise** für <dienst> wird zunächst die Datei `./supervise/lock` zum Lesen exklusiv geöffnet. Dies stellt sicher, dass immer nur ein **supervise** Prozess für `<dienst>` aktiv ist.

`control` und `ok` dienen als Named Pipes für **supervise**. `control` stellen sicher, dass **supervise** in das Verzeichnis `./<dienst>/supervise/` schreiben kann. Am interessantesten ist die Datei `status`. Sie beinhaltet in Binärform folgende Informationen:

- Den (gewünschten) Status des Daemonprozesses.
- Seine Pid.
- Den Zeitstempel des letzten Start(versuch)s bzw. Shutdowns.

Wir starten und stoppen mittels des Programms **svc** einmal den Daemon **dnscache** und schauen via **hexdump** in die Datei `status` hinein:

```
# cd /service/dnscache/supervise

# svstat /service/dnscache

/service/dnscache: up (pid 20987) 62693 seconds

# svstat /service/dnscache

# hd status

00000000  40 00 00 00 3d b9 be af  33 2f 3f a4 fb 51 00 00
|@...=...3/?..Q..|

00000010  00 75
|.u|

00000012

# svc -d /service/dnscache

# svstat /service/dnscache

/service/dnscache: up (pid 91518) 199 seconds, paused

# hd status

00000000  40 00 00 00 3d ba b5 6e  17 f1 05 4c 7e 65 01 00
|@...=..n...L~e..|

00000010  01 75
|.u|

00000012

# svc -p /service/dnscache

orion# hd status

00000000  40 00 00 00 3d ba b5 6e  17 f1 05 4c 7e 65 01 00
|@...=..n...L~e..|

00000010  01 75
```

```
|.u|

00000012

# svc -d /service/dnscache

# hd status

00000000  40 00 00 00 3d ba b6 5e  19 29 cb 9c 00 00 00 00
|@...=..^.)......|

00000010  00 64
|.d|

00000012
```

Dan Bernstein hat in den ersten 12 Byte den TAI64N-Zeitstempel hineingepackt, die Bytes 16 bis 13 beinhalten die Pid des Daemonsprozesses und — wie man deutlich erkennen kann — Byte 18 ist der gewünschte Prozessstatus, also z.B. up (u) oder down (d). Byte 17 teilt mit, ob der Prozess pausiert. Im Beispiel oben steht hier nach dem Pause-Signal eine "1"; ansonsten eine "0".

Änderungen der Prozessstati werden in `status` hinterlegt, wobei zunächst die Information in eine Datei `status.new` geschrieben wird, um sie anschliessend nach `status` umzubenennen.

Aus diesem Konstrukt wird klar, dass **supervise** immer nur genau einen Prozess-Identifier bedienen kann. Dies begründet die weiter oben beschriebene Einschränkung für die Daemons. **supervise** überprüft den in `status` hinterlegten Prozess alle 60 Sekunden. Wird hierbei eine Differenz zwischen der in `status` hinterlegten Soll-Information und dem realen Prozessstatus festgestellt, wird der Daemon-Prozess über die Datei `./<dienst>/run` ggf. nachgestartet und `status` neu geschrieben.

5.5.3 svc

svc ist das Kommandointerface von **supervise**; also quasi der Jockey (von Pferderennen verstehe ich nichts). Für Dienste, die unter `/service/` liegen, werden zur erfolgreichen Ausführung *root* Rechte benötigt. **svc** liest die Datei `./<dienst>/supervise/status` aus und kennt hierüber die Pid des Daemonprozesses. Auf diesen Prozess wirkt **svc** mit folgenden Befehlen ein:

svc-Befehl	Bedeutung	Auswirkung
`svc -u`	Up	Startet den Daemon; beendet er sich, wird er neu gestartet.
`svc -d`	Down	Beendet einen laufenden Daemonprozess mittels des TERM Signals.
`svc -o`	Once	Startet den Daemon. Beendet er sich, wird er nicht neu gestartet.

`svc -p`	Pause	Sendet das Signal STOP an den Daemon.
`svc -c`	Continue	Sendet das Signal CONTINUE an den Daemon.
`svc -h`	Hangup	Sendet das HUP-Signal an den Daemone; häufig werden hierdurch die Initialisierungsdateien neu gelesen.
`svc -a`	Alarm	Sendet das ALRM-Signal an den Prozess, was i.d.R. mit der Beendigung laufender Subprozesse quittiert wird.
`svc -i`	Interrupt	Das Signal INT wird an den Prozess übermittelt.
`svc -t`	Terminate	Das Signal TERM wird an den Prozess geschickt.
`svc -k`	Kill	Das KILL-Signal wird an den Prozess geschickt.
`svc -x`	Exit	**supervise** beendet sich, sobald der Prozess stoppt bzw. heruntergefahren wird.

*Tabelle 5-1: **svc**-Kommandos und ihre Bedeutung.*

svc bedient sich somit einer Auwahl von Standard Unix-Signalen (vgl ...). Welche konkrete Auswirkung ein Signal auf einen Prozess hat, wird alleine über den Prozess festgelegt. Häufig ist es so, dass das HUP-Signal zum Neueinlesen der Konfigurationsdaten genutzt wird (ohne den Prozess herunterzufahren und neu starten). Mit einem Blick auf den **xinetd** beispielsweise (der für diese Aktion das Signal USR1 braucht), kann dies jedoch nicht verallgemeinert werden.

Bei der Nutzung der Supervise-Tools kann der Anwender getrost auf die Buchführung der Pids der Daemonprozesse verzichten. Bei korrekter `./run`-Datei nimmt ihm das Gespann **supervise**/**svc** dies ab. Im Gegensatz zum Konstrukt einer `/var/.../dienst.pid` Datei ist hier durch die zusätzlichen Kontrolldateien auch eine wesentlich verbesserte Integrität der Informationen gewährleistet. Es kann eben nicht sein, dass eine verwahrloste `./dienst.pid` Datei die Prozesssteuerung ad absurdum führt.

Der Einstatz von **svc** wird durch zwei weitere Eigenschaften enorm vereinfacht:

- **svc** erlaubt kumulative Kommandos, z.B. `svc -d -u /service/<dienst>` im getopts Stil.
- **svc** honoriert Shell-Ergänzungen. Hierdurch kann z.B. das gesamte DNS-System mittels `svc -d /service/*dns*` heruntergefahren werden.

Bei vernünftiger Wahl der Namen für die `<dienst>`-Verzeichnisse, möchte man auf diese "kummulativen" Möglichkeiten nicht mehr verzichten.

Entsprechendes gilt z.B auch für das Logging mittels **multilog** . Ein Aufruf

```
# svc -d /service/*/log
```

stoppt alle Logging-Prozesse. Einmal richtig aufgesetzt, ermöglichen uns die

Supervise-Tools eine enorm einfache Administration der Daemonprozesse. Die Tatsache, dass hiermit — quasi en passant — eine äusserst effektive Hochverfügbarkeit der Daemonprozesse gewährleistet ist, machen die Supervise-Tools im realen Einsatz nahezu "indispensable".

5.5.4 svok

Einzige Aufgabe von **svok** liegt darin, zu überprüfen, ob **supervise** für <dienst> läuft. Dies geschieht mittels der Datei `./<dienst>/supervise/ok`, die **svok** zum Schreiben öffnen will.

```
svok [/service/]<dienst>
```

Gelingt dies, beendet sich **svok** mit dem Return-Code 0; andernfalls mit 100. Eine zusätzliche verbale Ausgabe auf STDOUT nimmt **svok** nicht vor.

5.5.5 svstat

Das Kommando **svstat** haben wir schon bei der Diskussion der Kontrolldatei `status` kennengelernt. Tatsächlich ist es die vornehmliche Aufgabe von **svstat**, den Inhalt von `.<dienst>/supervise/status` auszulesen und in leserlicher Form zu präsentieren, wobei aber zunächst `ok` konsultiert wird, ob **supervise** überhaupt läuft (vgl. **svok**). Hier ein Beispiel anhand von **dnscache**:

```
# svc -d -u /service/dnscache
# svstat /service/dnscache /service/dnscachex
/service/dnscache: up (pid 52552) 5 seconds
/service/dnscachex: up (pid 91881) 38504 seconds
```

Wie wir im obigen Beispiel sehen, ermöglicht **svstat** die Angabe mehrerer `<dienst>`-Verzeichnisse und gibt die Resultate zeilenweise aus. **svstat** unterstützt auch den Einsatz von Wildcards:

```
# svstat /service/*dns*
/service/dnscache: up (pid 52552) 697 seconds
/service/dnscachex: up (pid 91881) 38877 seconds
/service/tinydns: up (pid 265) 1361786 seconds
```

5.5.6 fghack und pgrhack

fghack und **pgrhack** sind zwei kleine Hilfsprogramme, die dann Einsatz finden, wenn spezielle Daemons unter **supervise** zum Einsatz gebracht werden sollen:

- **fghack** ermöglicht die Überwachung von Daemons (wie **inetd**) unter **svc** auch dann, wenn sie sich selbst in den Hintergrund setzen; allerdings unter

Verzicht auf die Steuerungsmöglichkeit durch Signale über **svc**. Um dies zu realiseren, erzeugt **fghack** eine Unix-Pipe zum Daemon und liest solange hieraus Daten, bis diese geschlossen wird. **svc** macht sich zu nutze, dass in der Regel die untergeordneten Prozesse diese Pipe erben und erst mit dem Beenden des letzten Unterprozesses auch die Pipe erlischt. Das **fghack** Verfahren funktioniert daher nur dann, solange der Daemon nicht von sich heraus offene Datei-Deskriptoren schliesst. Beispiel:

```
#!/bin/sh

echo "Starting inetd"

exec fghack inetd
```

- **pgrhack** generiert für die hierunter initialisierten Daemons einen neue Prozessgruppe; anderfalls erben sie diese von **svscanboot**. Dies ist dann von Belang, wenn ein Daemon glaubt mit seiner Beendigung auch allen in seiner Prozessgruppe befindlichen Prozesse ein TERM-Signal geben zu müssen (hierzu führt Dan Bernstein den **ppd**-Daemon aus):

```
#!/bin/sh

echo "Starting ppd"

exec pgrphack pppd nodetach call myisp
```

Das Verhalten der entsprechenden Daemons muss aber immer im Einzelfall überprüft werden. Häufig wechseln die "Paradigmen" eines Daemons von Release zu Release.

5.6 Logging mit den Daemontools

Viele Daemons nutzen standardmässig das Logging von Ereignissen über die Syslog-Facility (vgl. Abschnitt 2.x). Dieses Call-Interface ist an die Präsenz des **syslogd** gekoppelt der die Ereignisse nach Art und Severity behandelt und in der Regel entweder in entsprechende Log-Dateien (z.B. `/var/log/messages`) schreibt oder gar auf der Konsole ausgibt ("`... su: BAD SU erwin to root on /dev/ttyp3...`"). Beim **syslogd** handelt es sich um einen "Arbiter" zum Puffern und quasi-synchronen Schreiben unabhängiger Ereignisse unterschiedlicher Daemons. Treten die Ereignisse mit hoher Rate bzw. Häufigkeit auf (Ereignisse/Zeit), stellt der **syslogd** mit seinem zentralen Einsatz einen Engpass dar.

Stattdessen hat sich Dan Bernstein beim **multilog** für eine Unix-Pipe entschieden (vgl. Abbildung 5.1). Jeder Daemon ist explizit mit einem **multilog** Prozess gekoppelt. Die zeitliche Abhängigkeiten werden im Gegensatz zum **syslogd** nicht über (innere) zeitliche Verkettung der unabhängigen Ereignisse festgelegt sondern muss (quasi extern) über den aufgezeichneten Zeitstempel ermittelt werden. Es ist sinnvoller, das Logging für asynchrone Ereignisse auch real asynchron vorzunehmen.

Mit Ausnahme des speziellen Qmail-Programms **sploggers,** hat Dan Bernstein auch konsequenterweise keine syslog-Calls in seine Programme aufgenommen. Der generische Ansatz ist immer, die Standard- und Fehlerausgabe eines Programms über eine Pipe einem Logging-Programm verfügbar zu machen: **multilog**. Schwerwiegende Fehler beim Aufruf der Daemons werden zusätzlich per **readproctitle** in einen Puffer geschrieben, der über `ps -auxww | grep readproctitle` einfach ausgelesen werden kann. Im Gegensatz zum aktiven Ausschreiben eines Ereignisses auf der Konsole "verschwinden" die Fehlermitteilungen nicht, falls z.B. ein laufender Prozess eine Ausgabe macht (sehr beliebt: `tail -f /var/log/messages`), sondern verbleiben passiv; verlangen aber vom Adminitrator eine explizit "Einsicht".

Es obliegt dem Systemadministrator zu definieren, was er mit den Loginformationen im weiteren Arbeitsverlauf anstellt. Teilweise gibt es auch zwingende Gründe die Logdateien auszuwerten und zu archivieren. Dies trifft z.B. für Internet Service Provider (ISPs) zu, die per Bundes Datenschutz Gesetz (BDSG) und dem Telekommunikationsgesetz (TKG) dazu aufgefordert sind, entsprechende Informationen vorzuhalten.

Ich möchte hier die Schritte

1. Aufzeichnung und
2. Auswertung

unterscheiden. Hierbei stellen sich im Grunde genommen immer die folgenden Fragen:

- Wer hat eine Transaktion ausgelöst?
- Wann fand sie statt?
- Wie wurde sie beendet?

Aufgabe von **multilog** ist es zunächst eine sichere und zielgerechte Aufzeichnung zu gewährleisten sowie ein Instrument zur Weiterverarbeitung der Loginformationen bereit zu stellen. Der Systembetreiber/administrator kann entschieden, ob er

- die gesamte Loginformation aufzeichnet — oder —
- eine Positiv-Liste führt — oder ob—
- die Negativ-Resultate protokolliert werden.

multilog ermöglicht mittels der Selektionsfilter und dem Überführen der so ausgewählten Informationen in unterschiedliche Logdateien bereits bei der Aufzeichnung eine zielgerichtete Auswahl. **multilog** untescheidet sich hierbei massgeblich von der Syslog-Facility. Dies gilt auch für die zentrale Anforderung, ein gesichertes Schreiben der Loginformationen vorzunehmen.

5.6.1 Multilog

multilog ist der Nachfolger des Programms **cyclog**, das noch Bestandteil der Daemontools Version 0.53 war und beinhaltet aus dieser Entwicklung zustätzlich die Hilfsprogramme **fifo**, **errorsto** und **usually**. Daran erkennt man bereits, dass **multilog** ein relativ komplexes Programm ist.

Die Kooperation zwischen **multilog** und dem eigentlichen Daemonprozess funktioniert auf einem Co-Prozess Mechanismus, den **svscan** bereit stellt:

Der Prozess unter `<dienst>/run` sowie `<dienst>/log/run` werden via **svscan** beim Starten via **supervise** über eine Unix-Pipe gekoppelt. Alles was der Daemonprozess über STDOUT ausgibt, wird vom Co-Prozess (in der Regel **multilog**) über STDIN entgegen genommen. Ist zusätzlich die Fehlerausgabe, also STDERR zu berücksichtigen, muss diese auf den File-Deskriptor 1 (STDOUT) umgelenkt werden.

Innerhalb des Daemonprozess **run**-Skriptes kann diese Umleitung entweder "global" (`exec 2>&1`) ober als Abschluss des über das **exec**-Statements aufgerufenen Programms explizit angegeben werden (`exec` *`daemon`* `2>&1`).

5.6.1.1 Aufgabe von multilog

Eigentlich haben wir bereits die vordringlichste Aufgabe von **multilog** kenngelernt: Die (unkommentierte) Entgegennahme von Daten, die das Programm über FD 1 erhält und diese zeilenweise, geschichert wegzuschreiben. Dies Aufgabe erscheint trivial; doch wer bereits mit zerstückelten, abgebrochenen und nicht mehr rekonstruierbaren **syslogd**-Aufzeichnungen zu tun hatte, wird schnell eines Besseren belehrt.

Doch inwiefern unterstützt uns **multilog**?

1. *Aufzeichnen und Ausgabe von Loginformationen*
   multilog verlangt stringente Disziplin beim Aufzeichnen der Loginformationen:

   - Pro Daemon können *mehrere Logfiles* mit zeilenweise unterschiedlich selektierten Informationen aufgezeichnet werden, die in separate Verzeichnisse, dort aber immer in die Datei `current` geschrieben werden.

   - Ein *Status* der aktuellen Loginformation wird über ein kriteriengesteuertes Schreiben in einen (oder mehreren) Statusfile(s) realisiert.

- Kriteriengesteuerte *Ausgabe* von Loginformationen auf STDERR.

2. *Sicherstellung der Integrität der Log-Informationen*

   - **multilog** liest und schreibt zeilenorientiert, wobei immer die (letzte) Zeile mit NL (Newline) Zeichen ausgeschrieben wird; auch bei Empfang des TERM Signals.
   - Falls **multilog** keinen Platz auf der Platte für die laufende Aufzeichnung findet, wird eine Fehlermittelung auf STDERR geschrieben und die Input-Daten gepuffert. Allerdings führt dies dazu, dass u.U. das Client-Programm angehalten wird.
   - Mehrfache **multilog**-Prozesse auf die gleiche Logdatei (`current`) werden verhindert. In diesem Fall schreibt **multilog** eine Fehlermeldung auf STDERR aus und terminiert mit Exit-Code 111.

3. *Automatische und getriggerte Speicherung der Logfiles*
   Ein häufiges Problem stellen zu grosse Logdateien dar, die mit dem verfügbaren Speicher nicht mehr bearbeitet bzw. analysiert werden können. **multilog** räumt damit auf und bietet und folgende Möglichkeiten:

   - Automatischer Abschluss des aktuellen Logfiles, Sicherung der alten Logs im Rotationsverfahren sowie Anlage einer neuen Datei. Die Grösse des aufzuzeichnenden Logfiles kann per Parameter zwischen 4 Kbyte und 16 Mbyte eingestellt werden, wobei eine Defaultgrösse von knapp 100 Kbyte vorgesehen ist. Die Anzahl der behaltenen Logdateien kann ebenfalls vorgegeben werden; hier sind per Default 10 Logfiles vorgesehen.
   - Getriggerter Abschluss der aktuellen Logdatei beim Empfang des ALRM-Signals.
   - Optional lässt sich die zu sichernde Logdatei über ein zusätzliches Programm (Processor) weiterverarbeiten

4. *Genauer Zeistempel*
   Last but not least:

   - Aufzeichnen der Log-Informationen mit einem TA64N-Zeitstempel, der zu Anfang jeder Logzeile geschrieben wird.
   - Alte und abgeschlossene Logdateien erhalten als Dateinamen einen TAI64N-Datumsstempel.

Der Anwedender spezifiert die einzelnen Aufgaben pro `./log/run`-Datei für jeden einzelnen Daemon. Die Parameter- bzw. Optionenliste beim Aufruf von **multilog** beschreibt Dan Bernstein in seiner Dokumentation daher konsequenterweise etwas undurchsichtig als *script*. Hierauf will ich im folgenden eingehen.

5.6.1.2 Syntax von multilog

Der generelle Aufruf von **multilog** lautet:

`multilog` *`script`*

script setzt sich aus einer Anzahl von Befehlen (mit Parametern) zusammen, deren Komplexität zwischen klein und umfangreich ausfallen kann — je nach Aufgabenstellung; **multilog** hält sich an keine *getopts* Standards.

Die Befehle ordnen sich in Befehlsgruppen mit einer definierten Reihenfolge ein. Prinzipiell ist das Vorgehen

Selektion (Parameter) —> Ausgabe; (Neu)Selektion (Parameter) —> Ausgabe

wobei die Ausgabe der gewünschten Information(en) mit (jeweils unabhängig wirkenden) Selektionsbefehlen in (mehrere) Logdateien, Statusdateien geschrieben bzw. auf dem Bildschirm dargestellt werden kann. Aufgrund der gewählten Syntax, ist die Arbeitsweise von **multilog** leider wenig transparent. Klarheit schafft Tabelle 5-2 in Verbindung mit dem weiter unten aufgeführten Bespiel und insbesonders Abbildung 5-2.

Befehlsgruppe	**Befehl**	**Parameter**	**Beispiel**	**Erläuterung**
Zeitstempel	`t`			fügt eine TAI64N-Zeitstempel zu Anfang jeder Logzeile ein
Selektion	`-`	Muster	`-*status*`	deselektiert eine Logzeile aufgrund eines Musters
Selektion	`+`	Muster	`+*Invalid*`	selektiert eine Logzeile mittels eines Musters
Ausgabe	`=`	Dateiname	`=./log/status`	schreibt die selektierte Logzeile in eine Statusdatei
Ausgabe	`e`			gibt die ersten 200 Byte der selektierten Logzeile auf STDERR aus

Parameter für Logdatei	`s`	Grösse (Byte)	`s16777215`	legt die maximale Grösse der Logdatei `current` fest
Parameter für Logdatei	`n`	Anzahl	`n20`	legt die maximale Anzahl der Backup-Logdateien fest
Parameter für Logdatei	`!`	Post-prozessor	`!postprocessor`	übergibt den Inhalt der Datei `current` nach Abschluss an den hier angegebenen Befehl
Ausgabe	`.` oder `/`	Ver-zeichnis-name		schreibt die selektierte Logzeile in die Datei `current` unter `./dir/`

Tabelle 5-2: Befehle und Parameter von ***multilog;*** *der Parameter* `Muster` *steht für einen "regulären Ausdruck" der aufrufenden Shell*

Das Programm **multilog** entpuppt sich beim genauen Hinsehen als ein mehrstufiges Filter (Abbildung 5-2): Ausgehend von den Eingabedaten (die aufgrund eines "Newline" zeilenweise verarbeitet werden), kann auf jeder Filterstufe eine Logdatei bzw. ein Statusfile geschrieben werden. Dan Bernstein führt in seiner Dokumentation dazu aus "*multilog reads a sequence of lines from stdin and appends selected lines to any number of logs*." Knapper kann man das Verhalten von **multilog** kaum ausdrücken.

Neben den Filtermöglichkeiten, die feinfühlig über (Shell-) Wildcards gesteuert werden können, sind vor allem die Ausgabemöglichkeiten von Interesse. Wie sich Tabelle 5-2 entnehmen lässt, kennt **multilog** drei Arten der Ausgaben:

1. Üblicherweise schreibt **multilog** zeilenweise in eine Logdatei `current`, die in einem — über die Angabe von `/dir` oder `./dir` — definierten Verzeichnis liegt. Der Verzeichnisname kann ohne oder mit abschliessendem Schrägstrich angegeben werden; massgeblich ist, dass der Verzeichnisname mit einem Punkt oder einem Schrägstrich beginnt. **multilog** muss natürlich Schreibrechte in dieses Verzeichnis besitzen. Verfügt **multilog** ebenfalls Schreibrechte im übergeordneten Verzeichnis, wird `./dir` automatisch erzeugt; ansonsten muss es explizit mit den Rechten des **multilog** Users angelegt werden.

Die Datei **current** wird exklusiv für diesen **multilog** Prozess herangezogen, d.h. kann nicht von zwei **multilog** Prozesses gemeinsam genutzt werden (im Gegensatz zum **syslogd**). Andernfalls beendet sich **multilog** mit dem Exit-Code 111.

2. Schreiben einer aktuellen (und selektierten) Statusinformation in eine Datei mittels des Befehls `=datei`. Hierin befindet sich dann immer das letzte aktuelle Ereignis. Es werden maximal 1000 Byte aufgezeichnet.

3. Ausgabe einer selektieren Information auf der Standard-Fehlerausgabe. Vergleichbar der oben vorgestellten Statusinformation, werden hier lediglich die ersten 200 Byte der in Frage kommenden Logzeile dargstellt.

Beispiel zum Einsatz von multilog

Abbildung 5-2 beschreibt einen nicht ganz trivialen Einsatz von **multilog** unter Einsatz aller drei Ausgabemöglichkeiten. Wir beachten, dass (z.B. bei Einsatz von **tcpserver**) zunächst Logzeilen verworfen werden, die das Wort "status" enthalten und der so bereinigte Log nach `./dir1/current` geschrieben wird. Auf der nächsten Filterstufe werden die Logzeilen mit dem Kriterium "alarm" in eine separate Logdatei unter `./dir2/current` eingereiht. Zudem werden diese Logzeilen auf der Standard-Fehlerausgabe zur Anzeige gebracht. In einer letzten Filterstufe werden Logzeilen mit dem "Inval" gefiltert und das jeweils letzte Ereignis in die Statusdatei `./invalid` geschrieben.

Im oberen Teil der Abbildung wurde versucht, die Logik des Programmaufrufs wiederzugeben; während untenstehend der Programmaufruf selbst dokumentiert ist.

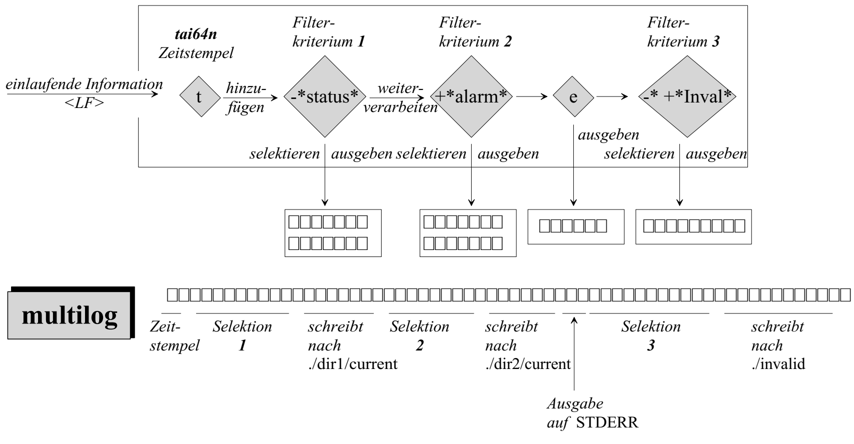

*Abbildung 5-2: Arbeitsweise von **multilog** (oben) und Programmaufruf (unten) in einem nicht ganz trivialen Fall (vgl. Beispiel im Text)*

Die Herkunft aus der Daemontools-Suite verbirgt **multilog** nicht, erzeugt es doch beim Aufruf die Dateien `lock` und `status`, auch wenn es nicht unter **supervise** Regie läuft.

5.6.1.3 Signale für multilog

Für die Verarbeitung der Loginformationen reagiert **multilog** auf drei Signale

- SIGUP — **multilog** beginnt entsprechend seiner Parametrierung damit, Loginformationen aufzuzeichnen.
- SIGTERM — **multilog** liest und verarbeitet die einlaufenden Information bis zum nächsten "Newline" und beendet sich dann. STDIN steht anschliessend auf dem ersten nicht verarbeiteten Byte.
- SIGALRM — **multilog** schliesst die aktuellen `./current` Dateien in den Logverzeichnissen ab und generiert neue.

5.6.1.4 Logfile Rotation

Beim **syslogd** wird die Rotation von Logfiles über die (etwas kryptische) `Datei /etc/syslogd.conf` bestimmt. Wie üblich bei den Programmen von Dan Bernstein, tritt an die Stelle einer *zentraler* Konfigurationsdatei ein *lokaler* Parameter dessen Angabe zudem — ohne das Betriebssystem in Mitleidenschaft zu ziehen — auf vernünftige Defaultwerte gesetzt ist.

Ein neuer Logfile (`current`) wird unter den folgenden Bedingungen geschrieben:

- *Explizit*: Dem **multilog** Prozess wird ein ALRM Signal geschickt.
- *Implizit*: Die Datei `current` wächst über die mittels des Parameters `s` definierte Grösse bzw. die Default-Grösse (automatische Logfile Rotation).

Zunächst wird die alte Logdatei in `previous` (ursprünglich `current`) umbenannt, um sie dann anschliessend unter ihrem TAI64-Datumsstemple zu sichern (typisch `@40000000xxxxxxxxxxxxxxxx`). Diese Datei kann einen der folgenden Suffixe besitzen:

- `.s`: Die Logdatei wurde komplett und fehlerfrei geschrieben.
- `.u`: Die Logdatei wurde während eines (erkennbaren) Fehlerfalles erzeugt; möglicherweise beinhaltet sie nicht alle aufgezeichneten Daten.

Häufig besteht die Aufgabe, die Auswertung der Loginformationen pro Tag vorzunehmen. Mittels **multilog** lässt sich dies mittels zweier unterschiedlicher Vorgehensweisen lösen:

- *Erzwungen (forced):* Es wird eine Auswerteskript erstellt, das — z.B. über eine crontab gesteuert — dem entsprechenden **multilog** Prozess ein ALRM-Signal sendet, die erzeugten Logdateien sichert und sie anschliessend der eigentlichen Auswertung zuführt.

- *Veranlasst (triggered):* Über einen crontab-Eintrag wird dem **multilog** Prozess das ALRM-Signal geschickt; dieses triggert über einen **multilog** Postprozessor (vgl. Tabelle 5.2) die Auswertung.

Wir berücksichtigen, dass ein

```
svc -a /service/<dienst>/log
```

immer alle von **multilog** generierten `current` Dateien in allen Verzeichnissen schliesst und neue generiert. Bei einem sehr geschäftigen Server (z.B. ein Qmail MTA bei einem ISP), liegen häufig mehrere gesicherte `current` Dateien als `@40000000xxxx.s` vor. Diese müssen natürlich gesammelt und gemeinsam verarbeitet werden. Der Dateiname ist identisch dem Generierungszeitpunkt, was sich schnell verifizieren lässt:

```
# ls @4* | tai64nlocal
```

Sollen alle Dateien über ein Skript eingebezogen und vearbeitet werden, reicht in der Regel eine einfache Shell-Schleife mittels

```
for LOGFILE in $(ls @4* 2>/dev/null)
do
       ....
done
```

Aufgrund der Monotonie des Zeitstempels werden hierdurch automatisch die älteren Dateien zuerst verarbeitet.

Abschluss von `@40000000xxxx.s` abwarten

Ein Problem bei der erzwungenen Logfile-Auswertung ergibt sich dadurch, dass nach Empfang des ALRM-Signals **multilog** u.U. bis zum finalen Abspeichern der Logdatei als `@400000xxxx.s` einiges an Zeit braucht; abhängig davon wie gross `current` war und wie geschäftig der Server ist. Es ist ratsam, ins Auswerteskript eine genügend grosse "pause" einzubauen. Wird hingegen das Trigger-Verfahren eingesetzt, entfällt diese Massnahme, da die Auswertung nicht aufgrund der *abgespeicherten* Datei `@4000000xxxx.s` erfolgt, sondern mittels der *abgeschlossenen* Datei `current`.

5.6.1.5 Logfile Postprozessing

mulitlog bietet die Möglichkeit nach Abschluss der Datei `current` automatisch einen Postprozessor zu starten:

```
multilog t !postprocessor /dir
```

Dieser Postprozessor (in der Regel ein Shell-Skript) wirkt ausschliesslich auf die Datei `current` im Verzeichnis `/dir`. Kennzeichnend hierbei ist, dass der Inhalt der Datei `current` von STDIN eingelesen und die Ausgabe von

postprocessor per Default als `@4000000xxxx.s` gespeichert wird. In "Shell-Notation" stellt sich das Verhalten von **postprocessor** folgendermassen dar:

```
postprocessor 0< current 1> @40000000xxxx.s 4< state 5>
newstate
```

Beim Einsatz von **postprocessor** ist folgendes zu berücksichtigen:

- Die alte Loginformation wird <u>nicht</u> zusätzlich gesichert. Will man dies erreichen, muss die ursprüngliche Loginformation in unterschiedliche Verzeichnisse geschrieben, d.h. dupliziert werden.
- Zusätzlich steht **postprocessor** der Inhalt der Datei `state` auf Filedescriptor 4 zur Verfügung. Soll diese Information zusätzlich zu der von `current` (via STDIN) weiterverarbeitet werden, reicht im Shell-Skript **postprocessor** das Statement `exec 4>&0`.
- Ausgaben von **postprocessor** auf dem Filedescriptor 5 werden zunächst in eine Datei `newstate` (im Verzeichnis `/dir`) geschrieben. Nach Abschluss von **postprocessor** "moved" **mulitlog** automatisch die Datei `newstate` zu `state`, die dann beim erneuten Aufruf von **postprocessor** automatisch eingelesen wird.
- Wir können die Grösse der Datei `current` abhängig vom Ausgabeverzeichnis `/dir` wählen. Hierdurch kann der Umschlag der Datei `currrent` und damit die Auswertung per Postprozessor schneller erfolgen als die des Standardlogs.
- Fehler in **postprocessor** führen zum "pausing" von **multilog** und — unter ungünstigen Fällen — zum Stopp der eigentlichen Applikation. Hier ist es unbedingt notwendig, Tests sehr sorgfältig vorzunehmen und die Resultate speziell mittels `ps -auxww | grep readproctitle` zu überprüfen.

```
# ps -auxww | grep readproctitle

root     111  0.0  0.1   860  252 con- S+   24Feb03
0:00.03 readproctitle service errors:
...using\n/usr/local/scripts/test[3]: print: -u: 3: bad file
descriptor\nmultilog: warning: processor failed in
/var/log/qmail-send, pausing\n/usr/local/scripts/test[3]:
print: -u: 3: bad file descriptor\nmultilog: warning:
processor failed in /var/log/qmail-smtpd,
pausing\n/usr/local/scripts/test[3]: print: -u: 3: bad file
descriptor\nmultilog: warning: processor failed in
/var/log/qmail-send, pausing\n
```

5.6.1.6 Erweiterter Einsatz

Beim Einsatz von **multilog** stellen sich schnell vier Fragen:

1. Mit welchem User (und ggf. Gruppe) soll das Schreiben der Loginformation

vorgenommen werden?

2. Wohin, d.h. in welches Verzeichnis (und damit Filesystem) soll geschrieben werden?

3. Wie setze ich die Selektionskriterien am besten auf?

4. Wie nehme ich die Auswertung der Loginformation vor?

Eines der Paradigmen von Dan Bernstein ist es, sowenig wie möglich den User *root* heranzuziehen. Dies gilt insbesonders auch für das Schreiben der Loginformation.

Ebenso existiert keine klare Meinung, wo die Loginfomationen niederzulegen sind. Traditionell hat sich das Verzeichnis `/var/log/` etabliert, dass häufig über den `/var` Mountpunkt auf einem eigenen Filesystem bzw. sogar Platte liegt. Ist dies gegeben, halte ich das Schreiben nach `/var/log/<dienst>/` für die geeignetste Wahl. Betrachten wir z.B. **dnscache**, so erfolgt bei der Standardinstallation das Logging nach `/etc/dnscache/log` und die Logdateien landen damit automatisch auf der Root-Partition `/`. Meiner Meinung nach keine geeignete Wahl. Beim **squid** z.B. werden sowohl die Cache-Informationen als auch das Logging nach `/usr/local/squid/.../` geschrieben (Unterverzeichnisse `./cache/` und `./log/`). Auch nicht unbedingt Performance-förderend.

Selektion der Loginformation

Wir betrachten einmal einen typischen `run`-File zum Mitschreiben der Ausgabe von **tcpserver** im Zusammenspiel mit **qmail-smtpd**. Damit **tcpserver** eine Ausgabe vornimmt, muss das Flage "-v" gesetzt sein. Der Daemon **qmail-smtpd** stellt überhaupt keine Loginformationen bereit; erst mit Ergänzungen, wie z.B. mein SPAMCONTROL wird er dazu gebracht, ein "Negativ"-Logging vorzunehmen:

```
#!/bin/sh

exec setuidgid qmaill multilog t \

       '-*tcpserver: status:*' /var/log/qmail-smtpd \

       '-*' '+*Invalid*' =/var/log/qmail-smtpd/invalid
```

*Listing 5-3: Einsatz von **multilog** bei **qmail-smtpd** (+SPAMCONTROL) und **tcpserver***

Wir wollen uns Listing 5-3 genauer anschauen:

- Voraussetzung ist, dass zunächst **tcpserver** (Flag "-v" und **qmail-smptd** (z.B. mittels SPAMCONTROL) dazu gebracht, per File-Deskriptor-Umleitung und mit Hilfe von **supervise** ihre Loginformationen unter dem User *qmaill* (über das Programm **setuidgid**) an **multilog** weiterzureichen (nicht gezeigt).

- Zunächst wird mittels des **multilog** Flags "t" ein TAI64N-Zeitstempels hinzugefügt (erste Zeile).
- Das erste Selektionskriterium besteht darin, alle von **tcpserver** ausgegebenen Zeilen mit "tcpserver: status" zu *deselektieren* und nach `/var/log/qmail-smtpd/current` zu schreiben (zweite Zeile). Die ausgeschlossenen Zeilen besitzen in der Regel für eine Auswertung (ausser im **tcpserver** Fehlerfall selbst) keine Bedeutung.
- In der weiteren Verarbeitung werden alle Zeilen *verworfen* ('-*'), um anschliessend nur die Informationen zu *selektieren*, wo ein "Invalid" zu finden ist. Ein aktuelles Exzerpt der Loginformation wird dann in der Datei `/var/log/qmail-smtd/invalid` hinterlegt, wo — im Beispiel per SPAMCONTROL — alle die SMTP-Verbindungen protokolliert werden, bei denen entweder die Sender- oder Empfänger-Adresse auf einer `bad*` Liste stehen (dritte Zeile), und somit im Log in den Zeilen "... Invalid ..." auftauchen.
- Bei der Angabe des Log-Verzeichnisses bzw. der Datei für die Exzerpt-Information wurde für **mulitlog** immer der komplette Pfad als Argument angegeben. Erst hierdurch ist es beim Einsatz des Kommandos `ps -auxww | grep mulitlog` möglich, das Zielverzeichnis ausfindig zu machen. Gleichfalls erfolgt nun z.B. bei **qmail-send** per `ps -auxww | grep qmail-send` eine gemeinsame Ausgabe des eigentlichen Prozesses und seines **mulitlog** Co-Prozesses:

```
# ps -auxww | grep multilog

qmaill   118  0.0  0.1   880  372 con- I+   Mon08AM
0:00.02 multilog t /var/log/qmail-send

qmaill   120  0.0  0.1   880  372 con- I+   Mon08AM
0:00.01 multilog t -*tcpserver: status:* /var/log/qmail-
smtpd -* +*Invalid* =/var/log/qmail-smtpd/invalid

# ps -auxww | grep qmail-send

root     112  0.0  0.2   872  400 con- I+   Mon08AM
0:00.02 supervise qmail-send

qmaill   118  0.0  0.1   880  372 con- I+   Mon08AM
0:00.02 multilog t /var/log/qmail-send

qmails   123  0.0  0.2   944  492 con- I+   Mon08AM
0:00.15 qmail-send
```

Bei der Ausgabe in unterschiedliche Logverzeichnisse, Statusdateien oder auf nach STDERR, nimmt **multilog** jeweils eine komplette Neuselektion der Information vor. Es ist verständlich, dass die konkreten Selektions/Deselektions-Statements aufgrund der Analyse des zunächst ungefilteren Logfiles erstellt und erprobt werden müssen. Dies geschieht in der Regel über das Anpassen des entsprechenden `run`-Skriptes für **multilog**. Änderungen werden erst wirksam, wenn das `run`-Skript neu eingelesen, d.h. der **multilog** Prozess neu gestartet werden muss:

```
svc -du /service/.../log
```

In der Regel findet das Logging bei **multilog** nicht über den User *root* statt sondern über einen dedizierten Benutzer; der im System natürlich angelegt werden muss. Im allgemeinen läuft **multilog** in der Regel unter *root*; was zwar nicht per se sicherheitsbedenklich; aber doch unnötig ist. Mittels des Daemontool-Hilfsprogramms **setuidgid** kann der bei Ausführung des `run`-Skripts gewünschter UID bzw. GID gewählt werden. Die effektiven Rechte müssen in jedem Fall mit den Schreibrechten im Log-Verzeichnis übereinstimmen. Als geeignet kann sich z.B die Wahl *daemon* für *Group* herausstellen; falls das Verzeichnis `/var/log/` Schreibrechte (mittels `chgrp +w daemon /var/log`) besitzt.

In der Regel ist die Wahl nicht kritisch. Es sei denn, es ist vonnöten, bestimmten Usern (= Kunden) Lese- (und ggf. Schreib-) Rechte in den Logverzeichnissen zu gönnen.

5.6.2 Das neue Datumsformat: TAI64N

multilog nutzt einen TAI64N-Zeitstempel. Immer wenn vom Datum und Zeit die Rede ist, haben wir es mit den folgenden Fragestellungen zu tun:

- Wie werden Zeit (und Datum) gemessen bzw. definiert ("Uhr") ?
- Wie werden Zeit und Datum dargestellt (Repräsentierung) ?
- Wie können die "Uhren" unterschiedlicher Systeme abgeglichen bzw. verglichen werden?
- Welcher Zusammenhang besteht zwischen "Uhr" und der Zeitzone ?
- Wie hängt das Datum mit dem "Kalender" zusammen ?

TAI64N macht sich daran, diesen Problemkreis radikal zu lösen — unter Verzicht auf bisherige Fehler.

TAI Definition

- TAI steht für *Temps Atomique International*, die per geometrischem Referenzsystem definierte (und somit entsprechend der Allgemeinen Relativitätstheorie koordinierte) streng monotone physikalische "Realzeit".
- Eine TAI-Sekunde ist identisch mit der SI-Sekunde: "Die Sekunde ist das 9192631770fache der Periodendauer der dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustandes von Atomen des Nuklids ^{133}Cs entsprechenden Strahlung" (`http://www.ptb.de/de/org/4/43/432/_index.htm`).

- Die TAI-Zeit wird unabhängig von der "geozentrischen" Zeit (z.B. UTC — Weltzeit definiert). Hierdurch unterscheidet sich die TAI-Zeit von der üblichen Zeit, die an die Rotationszeit der Erde um die eigene Achse und um die Sonne jeweils gekoppelt ist, und somit ab und zu nachjustiert werden muss (Schaltjahr; Schaltsekunde).
- Das *externe* TAI-Format wird üblicherweise sekundengenau als 64-bit Integer-Zahl — mit acht 8-Bit Byte im "Big-Endian" Format — geschrieben. Der (üblicherweise) hexadezimalen Repräsentierung x'b0 b1 b2 b3 b4 b5 b6 b7' entspricht die TAI-Zeit: $b0*2^{56} + b1*2^{48} + b2*2^{40} + b3*2^{32} + b4*2^{24} + b5*2^{16} + b6*2^{8} + b7$.
- Die "TAI-Epoche" beginnt am 1970-01-01 00:00:10 TAI (entsprechend Arthur David Olson's Beginn der TAI-Epoche); dieser Zeitpunkt (Datum) wird (extern) als x'40 00 00 00 00 00 00 00' dargestellt. Eine TAI-Epoche dauert 2^{62} Sekunden (etwa 146 Milliarden Jahre).
- Die verstrichenen Sekunden *s* vor diesem Zeitpunkt werden durch $2^{62} - s$ repräsentiert; also mithin die letze Sekunde in dieser Epoche als 3x'f ff ff ff ff ff ff ff'.
- Die Zeit nach der TAI-Epoche wird als z.B. x'40 00 00 00 2a 2b 2c 2d' = 1992-06-02 08:07:09 TAI bzw. 1992-06-02 08:06:43 UTC geschrieben.
- Die Sekunden *s* der nächsten TAI-Epoche werden zwischen 2^{62} und 2^{63} ausgedrückt. Dan Bernstein bemerkt hierzu, dass dies "adäquat ist, die gesamte erwartete Lebendauer des Universums abzudecken, sodass in diesem Fall keine Notwendigkeit besteht, die Zeitskala zu erweitern" (`http://cr.yp.to/tai64.html`).
- Trotzdem kennt die TAI-Zeit noch Werte zwischen 2^{63} und 2^{64}; diese sind für "zukünftige Erweiterungen" reserviert.
- Die Erweiterung TAI64N ergänzt die TA64-Zeitangabe um einen Nanosekunden-Stempel (10^{-9}), d.h durch die zusätzliche Angabe von 000000001 bis 999999999 (dezimal). Hierdurch wird das TAI64N-Format durch insgesamt zwölf 8-bit Bytes dargestellt.
- Ergänzend soll erwähnt werden, dass auch noch ein Attosekunden-genaues (10^{-18}) TAI Format vorgesehen ist — TAI64NA, das sechzehn 8-bit Bytes umfasst.

Dan Bernstein repräsentiert die TAI64N-Zahlen in Hexadezimal durch ein vorgestelltes "@" Zeichen (hier kurz als DJB-Format bezeichnet). Hierzu dient das Programm **tai64n**, das unter `/usr/local/bin/` verfügbar ist. Wird z.B. die Ausgabe des Befehls **date** nach **tai64n** per Pipe übertragen

```
# date | tai64n
```

```
@400000003e2487de2703f7ec Tue Jan 14 22:57:39 CET 2003
```

erzeugt dies den entsprechenden Datumsstempel und fügt die Ausgabe des Kommandos nach einem Leerzeichen an. Die Datum/Zeitangabe erfolgt nanosekundengenau (sofern es die Präzision des Systems erlaubt):

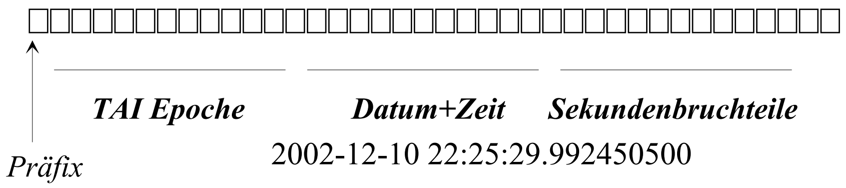

Abbildung 5-3: Darstellung eines Datums im externen TAI64N-Format

Durch den Trick, das TAI64N-Zahlformat durch eine "@" einzuleiten, macht sich dieses Format eindeutig identifizierbar. Software, die das TAI64N-Format zu interpretieren hat, braucht daher nur nach Zeichenketten mit beginnendem "@" zu schauen, um sie sodann in ein leserliches Format zu konvertieren.

Bei der Konvertierung können drei Wege eingschlagen werden:

1. Die im DJB-Format vorliegenden Zeitstempel können in eine TAI-Zeit mit ISO-Darstellung umgewandelt werden.

2. Das Zahlenformat kann in die Weltzeit (UTC) umgesetzt werden.

3. Die per DJB formatierte Zeitangabe kann in die aktuelle Datums- bzw. Zeitangabe der lokalen Zeitzone transformiert werden

Genau letztere Eigenschaft zeichnet das Progarmm **tai64nlocal** aus. **tai64nlocal** liest von der Standardeingabe das DJB-Format und gibt die Zeit der lokalen Zeitzone auf STDOUT aus. Bei der Konvertierung stützt sich **tai64nlocal** auf die Implementierung der Lokalzeit unter Unix (Kommando **localtime**). Bei den unterschiedlichen Implementierungen von **localtime** und den aktuellen Gegebenheit (insbesonders, ob der Rechner zeitsynchronisiert und ob die Zeitzone korrekt eingestellt ist) gibt es natürlich keine Garantie für die Richtigkeit dieser Angabe.

Wir haben bereits gesehen, dass die **multilog** Dateien mit einem TAI64N-Datumsstempel als Dateinamen geschrieben werden. Zwar könnten wir uns mittels **ls -la** auch das Datum anschauen; über eine triviale Unix-Pipe kann dies aber auch anhand des Dateinamens per **tai64nlocal** erfolgen:

```
# ls -la

-rw-r--r--  1 erwin  user    0 Dec 10 22:24
@400000003df65bd33b2797c4.u
```

```
-rw-r--r--  1 erwin  user    0 Dec 10 22:25
@400000003df65bf417c1043c.u
-rwxr--r--  1 erwin  user   65 Dec 10 22:26
@400000003df65c1a2a28984c.s
-rw-r--r--  1 erwin  user    0 Dec 10 22:26
@400000003df65c872a2b72c4.u
-rwxr--r--  1 erwin  user   67 Dec 10 22:28
@400000003df65ca130b8119c.s
-rw-r--r--  1 erwin  user    0 Dec 10 22:28
@400000003df65ce8045e2bf4.u
-rwxr--r--  1 erwin  user   34 Dec 10 22:30
@400000003df65cfd2dfbeb54.s
-rw-r--r--  1 erwin  user    0 Dec 10 22:30
@400000003df65d2029279754.u
-rwxr--r--  1 erwin  user   68 Dec 10 22:31
@400000003df65d3614d09724.s
-rw-r--r--  1 erwin  user    0 Dec 10 22:31 current
-rw-------  1 erwin  user    0 Dec 10 21:58 lock
-rw-r--r--  1 erwin  user  252 Jan 13 21:29 out
-rw-r--r--  1 erwin  user    0 Dec 10 22:31 state
# ls | tai64nlocal
2002-12-10 22:25:29.992450500.u
2002-12-10 22:26:02.398525500.u
2002-12-10 22:26:40.707303500.s
2002-12-10 22:28:29.707490500.u
2002-12-10 22:28:55.817369500.s
2002-12-10 22:30:06.073280500.u
2002-12-10 22:30:27.771484500.s
2002-12-10 22:31:02.690460500.u
2002-12-10 22:31:24.349214500.s
```

Letzteres is vor allem dann praktisch, wenn in einen Verzeichnis mit archivierten **multilog** Dateien solche mit einem spezifischem Datum gesucht und analysiert werden müssen. Mittels des Dateinames als Datumsstempel und den weiter oben dargestellten **multilog**-Suffixen kann sowohl Generierungsdatum der Datei als auch deren Status genau bestimmt werden; was sich vortrefflich in einem Skript verwenden lässt. Der vom Kommando **ls -la** hat dagegen üblicherweise den Nachteil, dass das Datum unterschiedlich ausgegeben wird, ob es sich im aktuellen Jahr befindet oder nicht — ein Grauen für jeden Programmierer.

Manchmal ist es sinnvoll, statt des von **tai64nlocal** ISO formatierten Datums lediglich eine Darstellung der Sekunden und Sekundenbruchteilen seit Beginn der TAI-Epoche zu haben. Dies leistet das Programm **tai64nfrac** von Russ Allberry, das wir schon in Abschnitt 5.3.1 kennengelernt haben. Der Vollständigkeit halber hier der Quellcode:

```
/* Convert external TAI64N timestamps to fractional seconds
   since epoch.
   Written by Russ Allbery <rra@stanford.edu>
   This work is in the public domain.
   Usage:
        tai64nfrac < input > output
   Expects the input stream to be a sequence of lines
   beginning with @, atimestamp in external TAI64N format,
   and a space.  Replaces the @ and the timestamp with
   fractional seconds since epoch (1970-01-01 00:00:00 UTC).
   The input time format is the format written by tai64n and
   multilog.
   Theoutput time format is expected by qmailanalog.  */
#include <stdio.h>
/* Read a TAI64N external format timestamp from stdin and
   write fractionalseconds since epoch (TAI, not UTC) to
   stdout.  Return the character after the timestamp. */
int
decode(void)
{
    int c;
    unsigned long u;
    unsigned long seconds = 0;
    unsigned long nanoseconds = 0;

    while ((c = getchar()) != EOF) {
        u = c - ’0’;
        if (u >= 10) {
            u = c - ’a’;
```

```
            if (u >= 6) break;
            u += 10;
        }
        seconds <<= 4;
        seconds += nanoseconds >> 28;
        nanoseconds &= 0xfffffff;
        nanoseconds <<= 4;
        nanoseconds += u;
    }
    seconds -= 4611686018427387914ULL;
    printf("%lu.%lu ", seconds, nanoseconds);
    return c;
}
int
main(void)
{
    int c;
    unsigned long seconds;
    unsigned long nanoseconds;
    while ((c = getchar()) != EOF) {
        if (c == '@') c = decode();
        while (c != EOF) {
            putchar(c);
            if (c == '\n') break;
            c = getchar();
        }
    }
}
```

Listing 5-4: Das TAI64N-Umwandlungsprogramm `tai64nfrac.c` von Russ Allberry

Wer den Überlegungen der TAI-Zeit bis zu diesem Zeitpunkt gefolgt ist, stellt sich vielleicht die Frage, warum statt eines linearen Zeitschemas (wie bei TAI) ein so komplexer und fehlerträchtiger **localtime** Mechanismus unter Unix implementiert ist. An dieser Stelle möchte ich bemerken, dass auf IBM /370 Rechnern (unter VM/XA) die Umstellung von z.B. Sommer- auf Winterzeit nur durch ein IPL (Initial Program Load), d.h. ein Reboot realisiert werden konnte. Statt einer "Eigenzeit" war die Verarbeitung dort (wie unter Unix) an die Realzeit gekoppelt.

Unter PC-Systemen obliegt es der Realtime Clock (RTC), über das Rechner-BIOS die Zeit auszugeben. Hierdurch muss die RTC auch im ausgeschalteten Zustand des Rechners über eine Batterie versorgt werden. BIOS und Betriebssystem können über das aktuelle Datum/Zeit durchaus unterschiedlicher Meinung sein! Selbst bei teuren Serversystemen, zeichnen sich die RTCs nicht immer durch hohe Konstanz aus. Ein erfahrener System-Administrator wird in der Regel seine Server daher per NTP bzw. XNTP mit einer Referenz abgeleichen, z.B einem öffentlichen NTP-Server (beispielsweise der *Physikalisch Technischen Bundesanstalt* PTB in Braunschweig: 192.53.103.103 bzw. 192.53.103.104) oder einem lokalen System mit DCF77 Empfänger. Bei meinen Servern geschieht ersteres immer um die Mitternachtsstunde per Cron. Teilweise liegen die Server dann aber schon um Sekunden daneben. Es würde also Sinn machen, eine quasi off-line Synchronisation zu besitzen.

Mittels des kleinen Programmpakets Clockspeed (`clockspeed-0.62.tar.gz`) von Dan Bernstein können wir dies realisieren. Das Paket besteht aus den Programmen **clockadd**, **clockspeed**, **clockview**, **sntpclock**, **taiclock** und **taiclockd**.

Mittels einiger Referenzmessungen werden zunächst die Zeitabweichungen des Systems gegenüber einem NTP-Server festgestellt und parametriert. In einem weiteren Schritt erfolgt (per Daemon) im laufenden Betrieb eine geeignete Kompensation der Systemzeit (unter Unix). Dies macht die Systemzeit wesentlich genauer und vor allem robuster gegen evtl. (Nicht-)Verfügbarkeiten der Zeitserver. Darüber hinaus ermöglicht es die Software über das auch, das System als TAI-Zeitgeber (**taiclockd)** bzw. -Zeitnehmer (**taiclock)** einzusetzen.

5.6.3 readproctitle

Das Programm **readproctitle** aus den Supervise-Werkzeugkasten ist so verblüffend einfach, kurz (30 Zeilen c-Code) und dennoch so durchschlagend, das man sich fragen muss, wieso Dan Bernstein erst jetzt darauf kam. Im Grunde genommen, stellt **readproctitle** das Tüpfelchen auf dem "i" in der Abkehr vom **syslogd** dar, wobei (wie wir gesehen haben) **multilog** der eigentliche **syslogd** Ersatz ist.

Aufgabe von **readproctitle** ist es, die Fehlerausgabe (STDERR bzw. FD 2) der aufgerufenen Programme per **ps**-Befehl zugänglich zu machen. Das Programm ist so gestaltet, dass es im fehlerfreien Zustand fünfhundert (500) Punkte (...) per **ps** ausgibt. Wird ein Fehler übermittelt, stellt **readproctitle** ihn sofort an erster

Stelle dar. Um **readproctitle** vollständig zu nutzen, bedarf es zweiter Anstrengungen:

1. Ein sinnvolle Aufgabe von **readproctitle** ist auf den User *root* beschränkt (sofern **svscanboot** — wie weiter oben beschrieben — vom System gestartet wurde.

2. Auf der Kommandozeile zeigt das generische **ps**-Kommando nur 80 Zeichen; es ist daher notwendig mittels des Flags -w (-www) eine umfangreichere Ausgabe von **ps** zu erzwingen.

Hierzu ein typisches Beispiel:

```
% su

# ps -auxww | grep readproctitle

root     220  0.0  0.2   848  264 con- I    12Mar02
0:00.02 readproctitle service errors: ...: fatal: unable to
figure out port number for -1\ntcpserver: fatal: unable to
figure ...
```

Diese Fehlermeldung resultiert aus der Tatsache, dass beim Aufruf von **tcpserver** kein TCP-Port angegeben wurde. Um welchen Dienst, bzw. ***Anwendung*** es sich handelt, verrät uns die Fehlermitteilung leider nicht; hier ist also Suchen angeraten.

Ein anderes Beispiel im Hinblick auf eine etwas komplexere Daemontools-Misskonfiguration:

```
% su

# ps -auxww | grep readproctitle

readproctitle service errors: ...le does not
exist\nsupervise: warning: unable to open
dummy/supervise/status.new: file does not exist\nsupervise:
fatal: unable to start dummy/run: file does not
exist\nsupervise: warning: unable to open
dummy/supervise/status.new: file does not exist\nsupervise:
warning: unable to open dummy/supervise/status.new: file
does not exist\nsupervise: fatal: unable to start dummy/run:
file does not exist\n
```

Der Dienst, über den sich **readproctitle** bzw. **supervise** beschwert, heisst "dummy". Zunächst wurde ein Verzeichnis gleichen namens unter `/service/dummy/` angelegte und somit **supervise** automatisch hierfür gestartet. **supvervise** versucht — wie weiter oben beschrieben — Statusinformationen über den zugeordneten Dienst über die Statusdatei `/servcie/dummy/supervise/status.new` zu erhalten. Pech für **supervise**, dass ich das gesamte Verzeichnis (bzw. den Link) gelöscht hatte ...

Bei der Interpretation der **readproctitle** Information, ist also eine gewisse

Vorsicht zu walten, da diese oft missverstanden werden kann:

- **readproctitle** gibt in erster Linie Auskunft über die "Befindlichkeit" der per exec im `run`-Skript aufgerufenen Programms; dies ist selten die "volle Wahrheit", sondern im allgemeinen nur die punktuelle Interpretation dieses Programms.
- **readproctitle** liefert somit einen Hinweise über einen Fehlerfall; nimmt aber keine eigene Diagnostik vor. Der System-Administrator muss somit mit "Bordmitteln" versuchen, die Fehlerursache zu lokalisieren und den Fehler zu identifizieren und zu beheben.
- **readproctitle** ermöglicht (leider!) keine zeitliche Identifikation der Fehlermitteilung. Der Puffer von **readproctitle** wird bei Bedarf von links nach rechts gefüllt, bzw. die ältesten Ereignisse verworfen. Dies erschwert die Signifikanz einer Mitteilung festzustellen. Auf der Suche nach Fehlern passiert es häufig, dass an einer "falschen Schraube" gedreht wird. Die hierdurch verursachte Fehlermitteilung erscheint dann als neueste in der **readproctitle** Ausgabe; obwohl sie für den eigentlichen Fehler unbedeutend ist.
- **readproctitle** ermöglicht (meines Wissens) kein Zurücksetzen der Ausgabeinformation auf "....". Hier sollte Dan Bernstein überlegen, ob nicht über ein Signal wie ALRM ein Zurücksetzen des Puffers angeraten wäre.

5.7 Einschränken der System-Ressourcen

5.7.1 Welche System-Ressourcen ?

Aktuelle Unix-Implementierung besitzten über Kernelparameter viele festgelegte Ressourcengrenzen. Hierzu zählen neben den "harten" Grenzen wie Hauptspeicher, virtuellem Speicher, Plattenplatz vor allem die Grössen MAXUSER, Anzahl der maximal verfügbaren Dateideskriptoren, Fork-Tiefe ....

Jedes Unix hat auch seine Vorstellungen wie diese Parameter zu verwalten sind; statisch oder über das `/proc` Filesystem. Hinzu kommt, dass diese Parameter in der Regel über Formeln verknüpft sind. Oracle macht z.B. dedizierte Vorgaben, welche Kernelparamater bei HP-UX gesetzt sein müssen. Deren Abhängigkeit leiten sich vor allem vom Hauptparameter MAXUSER ab.

In den aktuellen Versionen von FreeBSD versucht das System selbst über den aktuellen Ressourcenverbrauch die entscheidenden Parameter dynamisch anzupassen (MAXUSER=0 im Kernel).

Berücksichtigen wir, dass in der Regel (und besonders bei DJB-Software) Prozese nicht unter dem User *root* laufen, sondern einem dedizierten User zugeordnet sind, ergeben sich noch Abhängigkeiten durch

1. die Generierung des User-Accounts aufgrund von Vorgaben eventueller Skeleton-Parameter,
2. der Festlegung des Environments für diesen Benutzer, d.h. Shell und Umgebungsvariablen,
3. möglicherweise wirksamer Userlimits (**ulimits**).

Hierbei ist nicht die Präsenz und Wirksamkeit dieser Parameter zu kritisieren, sondern, dass

- diese in der Regel nicht dokumentiert sind (sondern implizit wirken) und
- es (häufig genug) gar keine Grenzen gibt, die es zu überwachen und einzuhalten gilt.

Mit einem Satz von kleinen Hilfsroutinen, insbesonders dem Programm **softlimit,** hilft uns Dan Bernstein hinsichtlich der expliziten Deklaration von Ressourcengrenzen beim Aufsetzen von Daemonprozessen. Aufgrund eines sog. Call-Interfaces können die Hilfsprogramme miteinander verkettet werden.

5.7.2 setuidgid

Die meisten Daemonprozesse werden von *root* initialisert; laufen aber nicht nicht notwendigerweise unter der *root* Kennung. Beim Aufsetzen der `run`-Skripte unter `/service/<dienst>` hilft uns das Programm, den geeigneten User anzugeben. Hierzu dient in der Regel der Aufruf

`setuidgid` *`account`* **`Programm`** *`Argumente`*

setuidgid wechselt zum User *account* und seiner primären Gruppe (ohne die möglichen zusätzlichen Gruppen in Betracht zu ziehen) und startet das Programm **prozess** (mit der Angabe seiner Parameter) unter dieser Kennung. Falls *account* nicht existiert, bzw. **prozess** nicht ausführbar ist, beendet sich **setuidgid** mit dem Exit-Code 111. **setuidgid** muss zum Wechsel des User-Kontextes natürlich unter der *root*-Rechten laufen.

5.7.3 envuidgid

envuidgid und **setuidgid** verhalten sich ähnlich, nur dass **envuidgid** zusätzlich in das Environment des entsprechenden User wechselt; es somit SUID und SGID nutzt.

`envuidgid` *`account`* **`Programm`** *`Argumente`*

Identisch mit **setuidgid** beendet sich **envuidgid** mit dem Exit-Code 111, falls entweder *account* nicht Existent oder **prozess** nicht ausführbar ist.

5.7.4 envdir

In der Einleitung des Kapitels zu den Daemontools habe ich bereits bemerkt, dass es die Aufgabe von **supervise** ist, Daemonprozesse in einer Verzeichnisstruktur abzubilden. Das Programm **envdir** dient hingegen dazu, Environment-Variablen das Dateien in einem Verzeichnis abzulegen. Dem *Dateinamen* entspricht hierbei die Environment-Variablen, der *Inhalt* der Datei wird von **envdir** als Wert der Environment-Variablen gesetzt.

Der generelle Aufruf ist hierbei:

`envdir Verzeichnis` **`Programm`** *`Argumente`*

`Verzeichnis` ist hierbei das Verzeichnis, in dem die Environment-Variablen als Dateien parametriert sind und für **prozess** zur Verfügung gestellt werden.

Für die Dateien in `Verzeichnis` gelten folgende Konventionen:

- Existiert in `Verzeichnis` eine Datei mit Namen `variable`, die in der ersten Zeilen den Wert `<value>` besitzt, entfernt **envdir** zunächt die Environment-Varable `variable`, um sie anschliessend mit dem Wert `<value>` neu zu setzen.
- Der Name von `variable` darf (aus verständlichen Gründen) kein "=" Zeichen beinhalten.
- Der Wert `<value>` wird ohne nachfolgende Leerzeichen oder Tabulatoren gesetzt. Das Zeichen "Null" wird zu einem "Newline" konvertiert.
- Ist die Datei `variable` komplett leer (z.B. über ein `toch variable`, d.h. 0-Byte gross, entfernt **envdir** die korrespondierende Environment-Variable für **prozess**, falls eine solche Umgebungsvariable vorher gesetzt war.

Exemplarisch hat Dan Bernstein diesen Mechanismus bei seinem DJBDNS genutzt. Wir betrachten hierzu ein konkretes Beispiel und wechseln (natürlich nach erfolgter **dnscache** Installation, vgl. Kapitel 8) ins Verzeichnis `/etc/dnscache/env/`. Dort befinden sich (bei meiner Installation) folgende Dateien:

```
% pwd
/etc/dnscache/env
% ls
CACHESIZE  DATALIMIT  FORWARDONLY  IP  IPSEND  ROOT
```

Diese Datein werden beim (Neu-)Start von **dnscache** eingelesen und ins

Environment gestellt. Sie entsprechend damit den Environment-Variable $CACHESIZE, $DATALIMIT, $FORWARDONLY, $IP, $IPSEND sowie $ROOT. Doch was ist der Inhalt der Dateien?

- CACHESIZE — 1000000 (Byte), maximale Grösse des Cache im Speicher
- DATALIMIT — 3000000 (Byte), maximale Grösse des Datensegementes von dnscache
- FORWARDONLY — (leer), Trigger für die Arbeitsweise von **dnscache**
- IP — 127.0.0.1, auf welche IP-Adresse **dnscache** bindet
- IPSEND — 0.0.0.0, welche Clients/Netze bedient werden (vgl. Abschnitt 8.5)
- ROOT — `/etc/dnscache/root`, das Verzeichnis, in dem die Root-Server Konfigurationsdateien hinterlegt sind.

5.7.5 softlimit

Das Programm **softlimit** ist das komplexeste Werkzeug innerhalb der Daemon-Helper. **softlimit** dient zur parametierten Ressourcenkontrolle eines aufgerufenen Programms. Bevor wir uns mit der Syntax und dem Aufruf von **softlimit** befassen, hier zunächst eine Zusammenstellung, was via **softlimit** kontrolliert bzw. beschränkt werden kann.

1. Speichernutzung:
   - Daten-Segment— die (fest alloziierte) Grösse (in Byte) des initialisierten Speicheres des ausführbaren Programms plus seines (noch) nicht initialisierten Speichers für Daten.
   - Stack-Segment — die (variable) Grösse (in Byte) für temporären Speicher.
   - Page Lock — die Grösse (in Byte) der gesperrten Seitenspeicher.
   - Offene Dateien — die Anzahl der Dateideskriptoren für offene Dateien.
   - Prozesse — die Anzahl der Prozesse pro UID.
2. Dateigrösse:
   - Ausgabedateien — ihre Grösse in Byte.
   - Core — seine Grösse in Byte.
3. Effizienz:
   - Permanenter Hauptspeicher — angefordert (in Byte), falls der

Hauptspeicher belegt ist.

- CPU-Zeit — ausführbare Zeit (in Sekunden) nach Empfang des Signals SIGXCPU.

softlimit führt das aufgerufene Programm (d.h. in der Regel der Daemon) mit definierten und kontrollierten Ressourcengrenzen aus. Neben der Angabe des Programms (und seiner Argumente) wird eine spezifische Ressource über einen Parameter voran gestellt. Abbildung 5-4 gibt eine vollständige Übersicht über die Nutzung von **softlimit** und seiner Parameter.

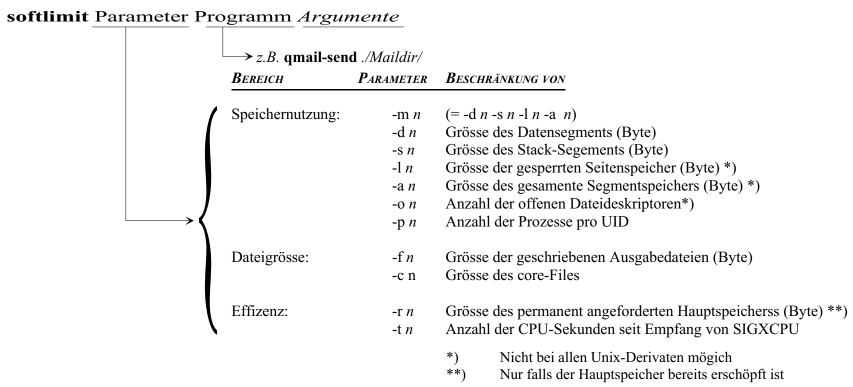

*Abbildung 5-4: Einsatz und Parameter von **softlimit***

Der Vorteil beim Einsatz von **softlimit** besteht in erster Linie darin, dem Daemon-Prozess genaue explizite Grenzen vorzugeben. Hierdurch kann ein Prozess nicht den gesamten Server lahmlegen (vgl. z.B. Felix von Leitner's x42.zip "Virus"), sondern wird auf das Einhalten der maximal definierten Ressourcen verpflichtet.

Problematisch ist der Einsatz von **softlimit** deshalb, weil in der Regel gar nicht genau festgelegt werden kann, wie die Ressourcengrenze zu bestimmen ist. Im allgemeinen gilt folgende Gleichung:

```
Ressourcenverbrauch = "Betriebssystem"+"Anwendung"+"Daten"
```

In der Regel wollen wir ja den nur den "Daten"-Bereich beschränken, d.h. viele Ressourcen einer "Anwendung" zugestanden wird, um eine bestimmte Aufgabe zu erfüllen. Der Anteil des "Betriesbssystem" ist aber durchaus unterschiedlich und beachtlich.

Ich hatte z.B. beim Upgrade meines Server von FreeBSD 4.3 auf 4.7 das Problem, das sich der **tcpserver**-Prozess mit einem `core` File verabschiedete und habe daraufhin meine Bebachtungen in die Qmail Mailingliste gepostet:

```
My Daemontools startup script:
#!/bin/sh
# qmail-smtpd Startup
QMAILDUID=‘id -u qmaild‘
QMAILDGID=‘id -g qmaild‘
export NODNSCHECK=""
MAXCONCURRENCY=‘cat /var/qmail/control/concurrencyincoming‘
exec softlimit -m 200 000 \
        tcpserver -vR -l qmailer.fehnet.de -c
$MAXCONCURRENCY \
        -u $QMAILDUID -g $QMAILDGID 0 smtp
/var/qmail/bin/qmail-smtpd 2>&1


With this configuration, tcpserver yields a core
(tcpserver.core) in the /service/qmail-smtpd directory.
Running the ./run script manually returns an exit code 139.
After adjusting SOFTLIMIT to -m 300 000 the following error
messages is shown:
# /usr/libexec/ld-elf.so.1: /usr/lib/libc.so.4: mmap of
entire address space failed: Cannot allocate memory
Raising the SOFTLIMIT to -m 1 000 000 (!) solved the
problem.
```

Dies habe ich mit der Bemerkung versehen, dass **tcpserver** doch bitteschön beim Start zu überprüfen hätte, ob überhaupt noch genügend Ressoucen zur Ausführung bereit stehen. Darauf bemerkte Dan Berstein:

```
"You misunderstand. Bugs of this type are in the operating
system’s C startup functions. Those functions dump core
before they even give the tcpserver code a chance to run."
```

5.7.6 setlock

Das Programm **setlock** ist quasi Dan Bernstein's eigener und gern eingesetzer Trick (`http://cr.yp.to/docs/selfpipe.html`), hier als explizites Programm realisiert, um Anwendungen exklusiv — d.h. nicht konkurrierend — zu starten. Mittels **setlock** wird eine Anwendung aufgerufen und zugleich eine Datei geschrieben und solange "gelocked" bis sich die Anwendung wieder beendet.

`setlock Parameter` `Dateiname` **`Programm`**

Hierbei steht `Dateiname` selbstverständlich für die Datei die als *lock file* zum

Einsatz gebracht wird und **Programm**, ist der Prozess, für den ein exklusiver Aufruf vorgesehen ist.

Die möglichen (im getopts-Stil) vorhandenen Parameter lauten wie folgt:

- -n: No delay. Ist `Dateiname` "gelocked", startet **setlock** keine neuen Prozess.
- -N: (Default.) Delay. Falls `Dateiname` "gelocked" ist, wartet **setlock** zum Start des neuen Prozesses bis `Dateiname` wieder freigegeben ist (und es dann sofort einen neuen *lock* generiert).
- -x: (Exit 0.) Falls `Dateiname` nicht geöffnet oder "gelocked" werden kann, beendet sich **setlock** mit dem Exit-Code 0.
- -X: (Default.) Falls `Dateiname` nicht geöffnet oder "gelocked" werden kann, beendet sich **setlock** mit einem Exit-Code ungleich 0 und gibt eine Fehlermeldung aus.

5.8 Effektiver Einsatz der Daemontools

Das Aufsetzen der Daemontools und ihre Pflege beschreibt Dan Bernstein in seinen Dokumenten lediglich rudimentär. Daher möcht ich aus meiner Erfahrung einige Tipps und Stolpersteine beschreiben, dir mir im Zusammenhang mit den Daemontools besonders aufgefallen sind. Dies ist daher eine umfassende Darstellung, sondern soll lediglich einige Fälle beschreiben, dir mir in der Praxis aufgefallen sind.

5.8.1 Daemon-Qualifizierung

Die grundliegende Idee der Daemontools von Dan Bernstein ist so überzeugend, und der Nutzwert in der Praxis so beachtlich, dass z.Z. bei vielen Softwareprojekten die Kompatibilität mit den Daemontools zumindest für das nächste Release angekündigt bzw. beabsichtigt ist. Dies trifft z.B zu auf den HTTP-Server Apache, Samba in der Version 3.0 (ein Patch für 2.2.8 steht auch zur Verfügung) und selbst Bind.

Andererseits gibt es viele Standard-Daemons, die sich nicht sinnvoll über **supervise** administrieren lassen, geschweige dann, ihren Log-Output an **multilog** übertragen. Den **pppd**-Dienst unter FreeBSD habe ich z.B. versucht auf **supervise**-Kontrolle "umzuerziehen" — ohne Erfolg.

Bei allen Unix-Systemen liegt noch ein grosser Bestand an "Legacy"-Software vor, der sich nicht sinnvoll auf die Daemontools umstellen lässt. Zunächst gilt es daher, die Kompatibilität des entsprechenden Daemons zu überprüfen:

- Lässt sich der Prozess per Unix-Befehle administrieren (vgl. Tabelle 5-1)?

- "Spawned" der Prozess, d.h. erzeugt er mehrere Unterprozesse?
- Ist es möglich, den Prozess im Vordergrund zu betreiben; braucht also das **fghack**-Programm nicht angewandt werden?
- Sendet der Prozess bei einem TERM-Signal auch ein **kill** an die Prozesse seiner Prozessgruppe?
- Kann der Output des Daemons umgelenkt werden, d.h. besteht der Prozess nicht darau das Logging per **syslogd** vorzunehmen bzw. auf die Konsole auszugeben?

Letzteres lässt sich häufig auch über ein Flag im `./config` Skript vor dem Kompilieren festsetzten; alle anderen Aspekte sind in der Regel eine Frage der Architektur des Daemons.

5.8.2 Aufsetzen und Pflege der run-Skripte

Vor dem erstmaligen Aufsetzen der Daemontools **run**-Skripte, sollten einige Vorüberlegungen angestellt werden.

Die Verzeichnisse, die später für einen spezifischen Daemon <**dienst**> die `run`-Skripte beinhalten (und — wie wir gesehen haben — unter **supervise** quasi als Platzhalter für den Dienst fungieren), werden in der Regel als Symlinks unter `/service/<dienst>/` erzeugt. Dieses Vorgehen ist vergleichbar dem Generieren der Run-Level Skripte bei System-V Systemen unter `/etc/rc.d/` und das anschliessende Erzeugen eines namentlich korrekten Symlinks im korrespondierenden Run-Level Verzeichnis.

Unter Rückgriff auf die von Dan Bernstein für DJBDNS vorgesehene Struktur, werden die `run`-Skripte im Stammverzeichnis des Dienstes untergebracht. Dies möchte ich als dezentrale, Dienst-zugeordnete Variante bezeichnen. Hierbei sollte die strenge Zuordung

```
Name des Dienstes ./<dienst>  ==
Name des Heimatverzeichnises des zugehörigen run-Skriptes
```

eingehalten werden.

Wie später erläutert wird, halte ich z.B. für Qmail die Verzeichnisse `/var/qmail/qmail-send/`, `/var/qmail/qmail-smtpd/` und `/var/qmail/qmail-pop3d/` hierzu geeignet. Da Qmail mehrere Daemons besitzt, wird jeder Daemon (= Dienst) in einem eigenen Verzeichnis eingerichtet.

Alternativ können natürlich auch alle für Steuerung unter **supervise** vorgesehenen Dienste — wie bei den System-V Runlevel-Skripten — unter einem zentralen "Oberverzeichnis" eingestellt werden.

Ohne, dass dies explizit genannt ist, glaube ich, Dan Bernstein würde die dezentrale Variante bevorzugen. Sinnvollerweise kann hierbei jeder Dienst in einem einem Stammverzeichnis eingerichtet werden. Zur Administration lässt sich dann folgende Struktur konsistent aufbauen:

```
Stammverzeichnis von Dienst: ./<dienst>

    ./       –     Heimat der run-Skripte;
                   wird nach /service/<dienst> gelinkt
                   (/service/<dienst> -> ./dienst)
    ./bin    –     die auführbaren Dateien
    ./man    –     die man-pages
    ./env    –     per envdir zugeordneten
                   Environment-Variablen
    ./etc    –     sonstige Konfigurationsdateien
    ./log    –     Link auf das Logverzeichnis,
                   das unter /var/log/ existiert
                   (./log -> /var/log/<dienst>)
```

Listing 5-5: Empfohlene Verzeichnisstruktur für einen Dienst unter Daemontools

Sind die `run`-Skripte zu modifizieren, sollte zunächst das ursprüngliche `run`-Skripte nach z.B. `run.new` kopiert und editiert editiert werden. Anschliessend wird der Daemon-Prozess per `svc -d /service/<dienst>` gestoppt. Vor dem "Scharfschalten", sollte zunächst dass neue Skript `run.new` auf seine Ausführbarkeit und seine Funktionstüchtigkeit überprüft werden. Per Konstrukt bleibt der Prozess dann im Vordergrund und kann sodann durch ein *ctl-c* auf er Konsole beendet werden. Sollte diese erfolgreich erfolgt sein, wird `run.new` nach `run` umkopiert und in einem letzten Schritt das Skript bzw. der Daemon-Prozess via `svc -u /service/<dienst>` gestartet. Abschliessend sollte verifiziert werden, dass in der **readproctitle** (**ps -auxww | grep readproctitle**) keine Fehlermeldungen auftauchen.

Sollte dieses Verfahren nicht von Erfolg begleitet sein (insbsonders bei der Ausgabe von **multilog** in nicht ordentlich "ge-**chown**-te" und "ge-**chmod**-ete" Verzeichnisse unter `/var/log/`, so sollte per **touch** Befehl unter `/service/<dienst>/` und ggf. unter `/service/<dienst>/log/` die Datei **down** erzeugt werden. Diese verhindert im Fehlerfall den Neustart des Daemons per **superivse**.

Das Löschen eines Dienstes vollzieht sich entsprechend der Empfehlung (FAQ) von Dan Bernstein über folgende Befehle:

```
# cd /serice/<dienst>

# rm /service/<dienst>

# svc -dx  . log
```

Dieses Vorgehen erzeugt bein ersten Hinschauen zunächst kribbeln in der Magengegend; es ist aber absolut folgerichtig. Im ersten Schritt wird ins Verzeichnis `/service/<dienst>` gewechselt. Wir landen aber anschliessend im Stammverzeichnis des Dienstes, also z.B. in `/usr/local/daemons/<dienst>/`. Von dort aus lässt sich prächtig den Mount-Punkt `/service/<dienst>` löschen und anschliessend den (noch) laufenden Daemon **dienst** samt seinem **multilog** Prozess und den jeweils zugeordneten **supervise** Prozessen löschen. Über die Shell-Auswertung des Names des lokalen Verzeichnisses gestaltet sich das ganz einfach; vorausgesetzt das Stammverzeichnis des **run**-Skriptes für **dienst** heisst `./dienst`. Voila.

5.8.3 Pitfalls

Im folgenden habe ich versucht, einige Standardfehler beim Einsatz der Daemontools als solche erkennbar zu machen

Kein *run*-Skript vorhanden

Im Erfolgsfall, d.h. wenn ein konkreter Daemon gestartet wurde und dieser läuft, ist die Aufgabe von **svscan** beendet — sofern nicht neue Dienste zu starten sind, bzw. neue Unterverzeichnisse eingerichtet worden. Verfehlt allerdings ein `run`-Skript sein "Klassenziel", so versucht **svscan** dennoch, das `run`-Skript auszuführen. Dies äussert sich dann in folgender Ausgabe:

```
/service/dummy/: up (pid 99244) 1 seconds
 # svstat /service/dummy/
/service/dummy/: down 1 seconds, normally up, want up
# svstat /service/dummy/
/service/dummy/: up (pid 99250) 1 seconds
```

In diesem Falle habe ich einfach unter `/service/` das Verzeichnis `./dummy` erzeugt. Anschliessend versucht **svscan** permanent — aber natürlich ohne Erfolg — das ensprechende (aber nicht vorhandene) `run`-Skript im Sekundenrythmus zu startet. Steht der Rechner neban, macht sich dies in einem unangenehmen "kratzenden" Zugriff auf die Festplatte bemerkbar (zumindest bei meinen SCSI-Disks); ein Geräusch; das einem die Nachenhaare hochstellen lässt.

Test des Logings mittels *mulitlog*

Es empfiehlt sich also dringend, den weiter oben gemachten Vorschlag zu beherzigen und zunächst die für **supervise** notwendige Verzeichnisstruktur ausserhalb von `/service/` zu erstellen und die Lauffähigkeit der Skripte dort zu verifizieren. Beim Einsatz von **multilog** kann z.B. die Wirksamkeit der Filter und das korrekte Schreiben in die Logdateien einfach dadurch geklärt werden, indem eine geeignete Sequenz per **echo** Befehl an das **multilog** `run`-Skript über eine Unix-Pipe übertragen, also z.B.

```
# echo "Testpattern Invalid" | ./log/run
```

und anschliessend die Ausgabe in den relevanten Logfiles überprüft wird.

Fehlendes exec Statement in run-Skripten

Häufig passierte es, dass das **exec** Statement am Anfang des run-Skriptes (nach Angabe des Befehlsinterpreters #!/bin/sh) vergessen wird. Dies hat für die Ausführbarkeit des Skripes keine Bedeutung; der Prozess wird anstandslos gestartet — er kann nun nicht mehr per **svc** beendet werden, da die hinterlegte Pid nicht dem Prozess selbst, sondern dem run-Skript zugeordnet ist!

Prozess in den Hintergrund gestellt

Ein besonders gern gemachter Fehler ist, in der run-Datei den Dienst bzw. Prozess per "&" in den Hintergrund zu stellen. Dies ist unter **supervise** tabu; alle Prozesse müssen zur Administration unter **supervise** im Vordergrund laufen.

Bash Pfad-Expansion

Beim Entfernen von **supervise** Diensten wartet die **bash** mit einem besonders gemeinen Verhalten auf. Zwar genügt es, für eine temporäre Stilllegung von Diensten unter **supervise** einfach per

```
# svc -d /service/<dienst>
# svc -d /service/<dienst>/log
```

zu arbeiten. Ebenso kann der Neustart einfach durch das Erzeugen der down Dateien verhindert werden

```
# touch /service/<dienst>/down
# touch /service/<dienst>/log/down
```

doch das hindert **svscan** natürlich nicht daran, auch permanent zü überprüfen ob sowohl Verzeichnis als auch die Datei down vorhanden sind; was unnötige Plattenzugriffe zur Folge hat. Nach dem Herunterfahren von <dienst> und dem Beenden von **supervise** mittels

```
# svc -x <service/<dienst>
# svc -x <service/<dienst>/log
```

bleiben nur knapp fünf Sekunden, um das Verzeichnis /service/<dienst>/ zu löschen; andernfalls startet **svscan** erneut den **supervise** Prozess.

Als Systemadministator verwende ich gerne die **bash**, da diese mir über das Wiederholen von Kommandos einigen Komfort ermöglicht. Das Kommando

```
bash # rm -r /service/<dienst>
```

kompletiert die **bash** korrekterweise mit

```
bash # rm -r /service/<dienst>/
bash # ls -la /service
<dienst>
```

Hups. Das Verzeichnis gibts ja noch! Was ist da passiert? Verd... , dafür ist das Verzeichnis /usr/local/daemon/<dienst>/svc/ weg! Und jetzt startet

svscan schon wieder den Daemon und beschwert sich:

```
# ps -auxww | grep readproctitle
```

```
readproctitle service errors: ...le does not
exist\nsupervise: warning: unable to open
<dienst>/supervise/status.new: file does not
exist\nsupervise: fatal: unable to start <dienst>/run: file
...
```

Die **bash** hat statt den Link zu löschen, mittels ihrer Erweiterungsfunktion das "Mutterverzeichnis" ins Nirwana geschickt. Damit ist genau das Gegenteil eingetreten, was wir eigentlich gewollt haben: Statt die Linkdatei zu entfernen und die Daemontools ruhig zu stellen, haben wir alle Konfigurationsdateien gelöscht und zudem noch **supervise** auf die Palme gebracht!

Also: Der Befehl "**rm** `-r`" ist beim Entfernen von Diensten unter `/service/` striktest untersagt. Jeder Einsatz wird mit dem sofortigen Löschen wichtiger Konfigurationsdateien bestraft!

Lock-Dateien

Im laufenden Betrieb stützt sich **supervise** auf die Erzeugen von Lock-Dateien, um den exklusiven Aufruf eine Ressource zu gewährleisten. Diese Lock-Dateien liegen nicht unter `/service/<dienst>/`, sondern natürlich im jeweiligen Stammverzeichnis des Dienstes. Will man eine Datensicherung vornehmen, ist darauf zu achten, dass nicht der Versuch unternommen wird, diese Lock-Dateien mitzukopieren. Beispielsweise bliebt der Prozess

```
cp -pR /var/qmail/ /home/backup/qmail/
```

beim Lesen der ersten Lock-Datei einfach stehen, sofern Qmail noch läuft.

5.8.4 Perspektiven

Dan Bernstein hat sich flexibel gezeigt, die Daemontools auf die Bedürfnisse der Anwender hin zu modifizieren. Tatsächlich bedeutet aber die notwendige Konvergenz aber auch ein Anpassen der Daemons an **supervise** & co. Viele in der Weiterentwicklung befindliche Daemons (speziell im Netzwerk-Bereich) werden bereits an die Anforderungen der Daemontools angepasst. Schlechter gestellt sind hierbei die "Legacy"-Daemons unter Unix.

Es stellt sich die Frage, welchen Umfang die "DJB"-isierung von Daemons annehmen soll. In der Qmail Mailing-Liste wurde bemerkt, das auch der INIT-Prozess für einen Neustart der eingebundenen Daemons sorgt.

Bleibt vor allem die Anforderung, dass die Daemons optional statt über den **syslogd** ihre Ausgabe auch (d.h. entweder/oder) per STDOUT bzw. STDERR verfügbar zu machen. Allerdings gibt es bislang keine Übereinstimmung, WELCHE Ausgabe WOHIN (d.h. auf welchen Dateideskriptor) umzuleiten ist. In der Regel werden die Dateideskriptoren STDOUT und STDERR beim Einsatz von **multilog** auf die Datei `current` umgeleitet. Daher sind Unterschiede also im Logfile von `current` nicht erkennbar. An die Stelle von *facility* und *level* beim **syslogd** muss daher beim **multilog** ein TAG-Wert stehen, damit ein entsprechendes Ereignis gefiltert werden kann.

Häufig kommt daher die Forderung auf, dass — wie beim **syslogd** — mehrere Daemons in eine Logdatei schreiben. Beim derzeitigen Konzept der Daemontools, das wie erläutert auf dem File-Locking-Mechanismus aufbaut, ist dies kaum zu realiserieren. Möglich wäre eine Erweiterung im Hinblick auf Named-Pipes.