2 Prozesse und Threads

Ältere Computer, wie die Homecomputer aus den 80er Jahren, führten immer genau ein Programm aus. Das Betriebssystem (z. B. CP/M[1] oder MS-DOS[2]) kümmerte sich dabei um das Laden des Programms (von Magnetband, Diskette oder Festplatte) und übergab dem geladenen Programm dann die Kontrolle über den Computer. In manchen Fällen wurde dabei ein Großteil des Betriebssystems sogar aus dem Speicher entfernt, denn die Programme funktionierten weitgehend eigenständig ohne Unterstützung durch das System. Nach Beenden des Programms wurde die Kontrolle an ein „Reststück" des Betriebssystems zurückgegeben, das dann eventuell ausgelagerte Teile wieder nachlud und auf weitere Befehle wartete – z. B. den nächsten Programmstart.

Dieses alte Betriebsprinzip heißt **Single Tasking** (engl. *task* = Aufgabe) – zu jedem Zeitpunkt gibt es genau ein aktives Programm.

Mit wachsender Rechnerleistung stiegen die Anforderungen an ein Betriebssystem, und es entstand der Wunsch nach **Multitasking**: der Fähigkeit eines Computers, scheinbar parallel mehrere Programme ablaufen zu lassen, die sich nach Möglichkeit nicht gegenseitig im Weg stehen sollten. Multitasking realisiert ein Betriebssystem, indem es über eine Zeitscheibe allen parallel abzuarbeitenden Programmen ein Stück der Rechenzeit zur Verfügung stellt und den Prozessor gleich wieder entzieht, damit das nächste Programm weiter rechnen kann. Geschieht diese Verteilung der Rechenzeit auf mehrere Programme effizient, entsteht beim Anwender der Eindruck der echten Parallelität.

Durch die Einführung von Multitasking erhöht sich der Verwaltungsaufwand für das Betriebssystem: Da es Programme mitten in ihrer Ausführung unterbrechen muss, gilt es, bei jedem solchen Programmwechsel diverse Informationen zu speichern, z. B. den Inhalt der Prozessorregister, in denen Zwischenergebnisse liegen könnten, die das Programm noch benötigt, wenn es seine Arbeit zu einem späteren Zeitpunkt fortsetzt. Auch ergibt sich eine deutliche Erschwerung der Speicherverwaltung: Wenn mehrere Programme sich den Hauptspeicher teilen, werden die Speicherbereiche sinnvollerweise vor dem Zugriff durch andere Programme geschützt. Welches Programm welchen Speicher nutzen darf, gehört zu den Verwaltungsinformationen, die das Betriebssystem jetzt vorhalten muss.

[1] `http://de.wikipedia.org/wiki/CP/M`
[2] `http://de.wikipedia.org/wiki/MS-DOS`

Ein (Maschinen-) Programm ist ausführbarer Code[3] (meist durch einen Compiler oder einen Assembler erzeugt), der auf einem Datenträger (z. B. der Festplatte) gespeichert ist. Ein Programm, das mitsamt den Verwaltungsinformationen des Betriebssystems ausführbar im Hauptspeicher gehalten wird, heißt **Prozess** – dabei kann es sich auch zweimal um das gleiche Programm handeln, das doppelt ausgeführt wird: in Form von zwei Prozessen, denen aber dasselbe Programm zugrunde liegt.

Das Betriebssystem verwaltet alle Prozesse in einer Liste (der **Prozesstabelle**), die auf unterschiedliche Weise implementierbar ist, z. B. als einfaches Array, als (einfach oder mehrfach) verkettete Liste, als Baumstruktur etc.

Prozesse können auf unterschiedliche Weisen miteinander kommunizieren, was wichtig ist, wenn man eine größere Aufgabe in mehrere Teilaufgaben zerlegt und jede von einem separaten Prozess erledigen lässt – die am gemeinsamen Projekt beteiligten Prozesse benötigen dann einen Mechanismus, über den Sie sich gegenseitig Statusmeldungen und Ergebnisse mitteilen können. Kapitel 6 (Interprozess-Kommunikation) beschäftigt sich mit dieser Thematik.

Wir starten nun mit einer Beschreibung, wie sich Prozesse einem Linux-Anwender präsentieren – danach folgt die Theorie, und mit den so gewonnenen Grundlagen lohnt sich ein Blick darauf, wie der Linux-Kernel die Konzepte umsetzt.

2.1 Praxis: Prozesse unter Linux

Prozesse begegnen dem Benutzer unter Linux in zwei Varianten:

- Aus reiner Anwendersicht geht es darum, Programme zu starten und anschließend unter Kontrolle zu behalten, also z. B. anzuhalten, wieder fortzusetzen oder auch gewaltsam zu beenden.
- Für Programmierer sind die Methoden wichtig, mit denen sich neue Prozesse starten und ihre Rückgabewerte abfragen lassen.

2.1.1 Prozesse aus Anwendersicht

Anwender starten und beenden Programme – mehr ist in der Regel nicht nötig, wenn das System einwandfrei funktioniert. Wird ein Programm aus der Shell heraus aufge-

[3] Shell-Skripte und andere Skripte, die von einem Interpreter ausgeführt werden müssen (z. B. Perl- und Python-Skripte) betrachten wir nicht als Programme: Das Programm ist in dem Fall der Spracheninterpreter, also die Shell, Perl oder Python.

rufen, wird es zu einem „Sohn"-Prozess der Shell; diese ist entsprechend der „Vater"-Prozess des neu gestarteten Prozesses.

So ergibt sich eine Baumstruktur der Prozesse, denn ein Prozess kann auch mehrere neue Prozesse starten. Die Wurzel dieses Baums ist bei Linux der `init`-Prozess, er hat die Prozess-ID 1.[4]

Um ein Programm aus der Shell heraus zu starten, geben Sie einfach seinen Namen ein – die Shell wartet dann, bis Sie das Programm beenden, und zeigt erst danach wieder den Shell-Prompt an. Wollen Sie in der Shell weiter arbeiten, hängen Sie an den Befehl das kaufmännische Und (&) an. Dann teilt die Shell Ihnen mit, welche Prozess-ID für das gestartete Programm vergeben wurde, und lässt Sie sofort in der Shell weiter arbeiten:

```
esser@sony:Skript> emacs test.txt &
[3] 24469
esser@sony:Skript> _
```

Nachdem Sie das Programm beenden, erscheint auf der Shell ein kurzer Hinweis dazu:

```
[3]+  Done                    emacs test.txt
```

2.1.1.1 Prozesse im Blick: jobs, ps, pstree

Mit dem Befehl `jobs` können Sie jederzeit prüfen, welche Programme Sie aus der Shell heraus gestartet haben (und die noch laufen):

```
esser@sony:Skript> jobs
[1]-  Running                 xpdf -remote sk skript-bs.pdf &
[2]+  Running                 nedit kap02/index.tex &
```

In eckigen Klammern steht dabei die so genannte **Job-ID** – sie ist nicht mit der Prozess-ID zu verwechseln, die `jobs` auf Wunsch zusätzlich angibt, wenn Sie den Parameter `-p` verwenden. Das können Sie überprüfen, indem Sie das Kommando `ps` aufrufen und die Ausgabe nach der Prozess-ID durchsuchen:

```
esser@sony:Skript> jobs -l
[1]-  8103 Running                 xpdf -remote sk skript-bs.pdf &
[2]+ 20568 Running                 nedit kap02/index.tex &
esser@sony:Skript> ps w|grep 8103|grep -v grep
 8103 pts/15   S      5:27 xpdf -remote sk skript-bs.pdf
```

[4] Tatsächlich gibt es noch einen Prozess mit der ID 0, der aber nur intern im Kernel verwendet wird – es ist der so genannte Idle-Prozess: ein Prozess, der aktiviert wird, wenn sonst kein Prozess rechenbereit ist.

Mit verschiedenen Befehlen können Anwender unter Linux herausfinden, welche Prozesse laufen, und diese beeinflussen.

Der Befehl `pstree` (Listing 2.1 auf Seite 17) zeigt die Baumstruktur aller Prozesse übersichtlich an – im Beispiel führen die Optionen `-pc` dazu, dass auch die Prozess-IDs ausgegeben werden; häufiger benutzt man aber das Kommando `ps`, dessen Ausgabeformat sich mit zahlreichen Optionen an die individuellen Informationsbedürfnisse anpassen lässt (mehr dazu in der Manpage zu `ps`):

```
> ps auxw
USER    PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME COMMAND
root      1  0.0  0.0    720    92 ?      S    Jun24   0:01 init [5]
root      2  0.0  0.0      0     0 ?      SN   Jun24   1:09 [ksoftirqd ⤸
    .../0]
root      3  0.0  0.0      0     0 ?      S<   Jun24   0:11 [events/0]
root      4  0.0  0.0      0     0 ?      S<   Jun24   0:00 [khelper]
root      5  0.0  0.0      0     0 ?      S<   Jun24   0:00 [kthread]
root      7  0.0  0.0      0     0 ?      S<   Jun24   0:02 [kblockd/0]
root      8  0.0  0.0      0     0 ?      S<   Jun24   0:00 [kacpid]
root    128  0.0  0.0      0     0 ?      S<   Jun24   0:00 [aio/0]
[....]
esser  5733  0.2 12.2  82420 63428 ?      S    Jul24   4:05 /usr/lib/ ⤸
    ...opera/7.50-20040422.1/opera --binarydir /usr/lib/opera ⤸
    .../7.50-20040422.1/
root   2670  0.3  0.0   1368   300 ?      Ss   08:24   2:39 zmd /usr/lib ⤸
    .../zmd/zmd.exe                                  --sleep 85198
esser  8037  0.0  0.6   6452  3384 pts/13 S+   11:23   0:05 ssh -X amd64
```

Die Spalten in der `ps`-Ausgabe haben die folgenden Bedeutungen:

USER Benutzer, mit dessen Rechten der Prozess läuft

PID Prozess-ID

%CPU Prozentsatz der verbrauchten CPU-Zeit

%MEM Prozentsatz des belegten Hauptspeichers

VSZ virtual memory size – Größe des virtuellen Speichers (nicht des tatsächlich vorhandenen Hauptspeichers), die der Prozess belegt

RSS resident set size – Größe des physikalischen Speichers, den der Prozess belegt

TTY Terminal, auf dem der Prozess läuft (bei Daemons und anderen Programmen ohne Terminal erscheint ein Fragezeichen)

```
> pstree -pcA
init(1)-+-acpid(2266)
        |-auditd(2727)---{auditd}(2728)
        |-cron(3234)
        |-cupsd(2706)
        |-dcopserver(4075)
        |-esd(12856)
        |-events/0(3)
        |-gconfd-2(31788)
        |-gpg-agent(4031)
        |-hald(2309)-+-hald-addon-acpi(2616)
        |            |-hald-addon-stor(2911)
        |            `-hald-addon-stor(2914)
        |-iald(3310)
        |-kaccess(4094)
        |-kded(4079)
        |-kdeinit(4072)-+-artsd(7184)
        |               |-kio_file(4402)
        |               |-klauncher(4077)
        |               |-konqueror(22430)
        |               |-konsole(11064)-+-bash(11065)---ssh(31205)
        |               |                |-bash(11083)---ssh(31577)
        |               |                |-bash(11119)---sux(11444)---bash(11447)
        |               |                |-bash(11137)
        |               |                |-bash(25637)-+-ssh(4522)
        |               |                |             `-xmms(7169)-+-{xmms}(7170)
        |               |                |                          `-{xmms}(7171)
        |               |                |-bash(25765)
        |               |                `-bash(15608)
        |               |-konsole(4773)-+-bash(4774)---ssh(8037)
        |               |               |-bash(8040)---ssh(8058)
        |               |               `-bash(8061)-+-less(15188)
        |               |                            |-nedit(9628)
        |               |                            |-pstree(15187)
        |               |                            `-xpdf(8103)
        |               |-kwin(4087)
        |               |-opera(5733)
        |               |-xterm(19857)---bash(19859)
        |               |-xterm(31749)---bash(31751)
        |               `-xterm(23678)---bash(23680)---ssh(23700)
        |-kdesktop(4089)
        |-kdesud(5373)
        |-kdm(3946)-+-X(3970)
        |           `-kdm(3971)---kde(3985)-+-kwrapper(4084)
        |                                   `-ssh-agent(4032)
```

Listing 2.1: `pstree` – Baumstruktur der Linux-Prozesse

STAT Status: Hier können verschiedene Werte stehen:

- R: running / runnable, ausführbare Prozesse (solche in der Runqueue, siehe Abschnitt zum Scheduler ab Seite **??**)
- S: sleeping, unterbrechbarer Schlaf
- T: gestoppt, z. B. durch ein Jobkontrollsignal
- Z: Zombie – Prozess beendet, aber der Vaterprozess hat noch nicht den Rückgabewert abgerufen

START Wann wurde der Prozess aktiviert?

TIME Rechenzeit (Zeit, die der Prozess die CPU genutzt hat)

COMMAND Hier steht der vollständige Befehl (inklusive aller Optionen und Argumente), mit denen das Programm gestartet wurde.

Ruft man `pstree` ohne die Optionen `-pc` auf, fasst das Kommando gleichartige Teilbäume zusammen – im Beispiel auf Seite 17 etwa die Teilbäume, die aus einer Bash-Shell und einem SSH-Prozess bestehen:

```
[...]
     |          |-konsole-+-3*[bash---ssh]
[...]
```

2.1.1.2 Unterbrechung, Vorder- und Hintergrund

Ein Programm, das Sie aus dem Terminal heraus (ohne ein angehängtes &-Zeichen) gestartet haben, können Sie in den meisten Fällen mit [Strg-Z] unterbrechen. Die Shell gibt dann eine Zeile der Form

```
[3]+  Stopped                 vi
```

aus. Mit dem Befehl `fg` (foreground) können Sie ihn fortsetzen, und zwar im **Vordergrund** – dadurch wird die Shell dann wieder blockiert. Alternativ schieben Sie den Prozess mit `bg` (background) in den **Hintergrund**: Das Programm läuft weiter, aber die Shell nimmt weiterhin Befehle entgegen. `fg` können Sie auch verwenden, um einen in den Hintergrund gebrachten Prozess in den Vordergrund zurückzuholen.

Gibt es mehrere Prozesse, die aus der aktuellen Shell heraus gestartet wurden (Sie sehen diese mit `jobs`, können Sie gezielt einen der gestoppten oder im Hintergrund laufenden Prozesse in den Vordergrund holen – dazu geben Sie hinter `fg` als Option das Prozentzeichen gefolgt von der Job-ID ein; entsprechend bringt `bg %NR` den gestoppten Job mit der Nummer NR in den Hintergrund:

```
esser@sony:Skript> jobs
[1]+  Stopped                 xpdf -remote sk skript-bs.pdf
[2]-  Stopped                 nedit kap02/index.tex
[3]   Stopped                 vi
esser@sony:Skript> bg %2
[2]- nedit kap02/index.tex &
esser@sony:Skript> jobs
[1]+  Stopped                 xpdf -remote sk skript-bs.pdf
[2]-  Running                 nedit kap02/index.tex &
[3]   Stopped                 vi
esser@sony:Skript> bg %3
```

(Nach dem letzten Befehl verschwindet die Ausgabe, weil der Editor-Prozess (`vi`) reaktiviert wird und den Inhalt der bearbeiteten Datei anzeigt.)

2.1.1.3 Signale: kill und killall

Mit den Befehlen `kill` und `killall` schicken Sie Prozessen ein Signal aus der Liste, die Sie mit `kill -l` aufrufen können:

```
> kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Der `kill`-Befehl erwartet eine Prozess-ID und optional eine Signalnummer (als Option, mit vorangehendem Bindestrich), z. B. `kill -9 20324`.

Die wichtigsten Signale, die Sie einem Prozess schicken können, sind die folgenden:

SIGTERM (15) Das Standardsignal, das `kill` auch versendet, wenn Sie keine Nummer angeben: Der Prozess wird damit aufgefordert, sich zu beenden, und hat dann noch die Gelegenheit, offene Dateien zu schließen, einen Abschiedsgruß

auf das Terminal zu schreiben und sonstige Aufräumarbeiten vorzunehmen. Das kann auch misslingen, in dem Fall hilft ...

SIGKILL (9) Das KILL-Signal zwingt den Prozess zum sofortigen Abbruch; Inhalte von geöffneten Dateien mit noch nicht auf Platte geschriebenen Änderungen gehen dabei möglicherweise verloren. Wenn dieser Befehl einen Prozess nicht erfolgreich „abschießt", gibt es zwei mögliche Gründe: a) Der Prozess wartet noch auf den Abschluss einer I/O-Operation, woraus ihn nicht mal dieses Signal erwecken kann. b) Der Prozess ist bereits ein **Zombie**: Der Vaterprozess ist eingefroren und kann die Beendigungsnachricht des Prozesses nicht entgegennehmen.

SIGSTOP (19) Das STOP-Signal entspricht der Unterbrechung eines Prozesses über die Tastenkombination [Strg-Z] in der Shell: Das Programm wird unterbrochen. In der Prozessliste erhält es dann im Statusfeld den Wert T.

SIGCONT (18) Einen mit SIGSTOP unterbrochenen Prozess setzen Sie mit dem Signal CONT wieder fort. Das entspricht der Eingabe von `fg` (foreground) oder `bg` (background) in der Shell.

SIGHUP (1) Das Hangup- (HUP-) Signal werten vor allem einige Server-Prozesse (z. B. der Internet-Super-Daemon `inetd` bzw. dessen modernerer Nachfolger `xinetd`) so aus, dass sie ihre Konfigurationsdatei(en) neu einlesen. Damit kann man Änderungen in der Konfiguration sofort wirksam machen, ohne den Server-Prozess abzubrechen und neu zu starten.

2.1.1.4 Shell geschlossen, Programm läuft weiter?

Wird ein Prozess beendet, können im Regelfall auch seine Kindprozesse nicht weiter laufen – denn der Vaterprozess sollte ja z. B. die Rückgabewerte der Kindprozesse auswerten, und generell hängen Prozesse „in der Luft", wenn ihr Vaterprozess verschwindet: Jeder Eintrag eines Kindprozesses in der Linux-Prozesstabelle zeigt auf den Vaterprozess, und solche Verweise werden ungültig.

Typisch ist darum eine Kaskade von Programmabbrüchen: Zunächst werden alle Kinder des beendeten Prozesses beendet, dann deren Kinder usw., bis schließlich der ganze (Unter-) Prozessbaum, dessen Wurzel der beendete Prozess war, verschwunden ist.

Die „vaterlosen" Kindprozesse (engl. *orphans*, Waisen) erhalten das Signal SIGHUP (hang up) und reagieren darauf meist mit dem eigenen Programmende. Es gibt aber verschiedene Wege, dies zu verhindern.

- Um etwa vor dem Schließen eines Terminalfensters dafür zu sorgen, dass aus diesem Fenster gestartete Anwendungen weiter laufen, kann man sie mit

`disown` von der Shell loslösen: Sie taucht danach in der Jobliste nicht mehr auf, und die Shell wird beim Beenden kein Hangup-Signal an Prozesse senden, die mit `disown` abgekoppelt wurden.

```
esser@amd64:~> nedit &
[1] 30664
esser@amd64:~> jobs
[1]+  Running                 nedit &
esser@amd64:~> disown %1
esser@amd64:~> jobs
esser@amd64:~> exit
```

- Einige Prozesse fangen generell das Hangup-Signal ab und unternehmen nichts, wenn sie es empfangen.
- Über das Programm `nohup` lassen sich Programme gezielt mit diesem Verhalten starten (also mit Ignoranz gegenüber SIGHUP-Signalen). Typisch ist der Start von Hintergrundprozessen mit `nohup`, beim Aufruf der Form

  ```
  esser@amd64:~> nohup programm &
  ```

  startet `programm` im Hintergrund, und eventuelle Ausgaben auf die Standard- und Standardfehlerausgabe landen in einer Datei `nohup.out` im aktuellen Verzeichnis.[5]

Das Problem, dass einem Prozess der Vaterprozess entzogen wird, löst man nun in den Fällen, in denen die Kindprozesse weiter laufen sollen, durch Auswahl eines neuen Vaters: Diese Rolle muss der Init-Prozess (mit PID 1) übernehmen. Nach dem Beenden des ursprünglichen Vaterprozesses wird also immer bei allen (noch übrig gebliebenen) Kindprozessen die Parent Process ID (PPID) auf 1 gesetzt:

```
esser@amd64:~> bash   # neue Shell starten
esser@amd64:~> nedit &
[1] 31038
esser@amd64:~> ps -eo pid,ppid,cmd | grep nedit
31038 31029 nedit
esser@amd64:~> disown %1
esser@amd64:~> exit
exit
esser@amd64:~> ps -eo pid,ppid,cmd | grep nedit
31038     1 nedit
```

Dieser Vorgang wird auch *re-parenting* genannt.[6]

[5] Lässt sich diese Datei im aktuellen Verzeichnis nicht anlegen, versucht `nohup` es stattdessen mit `$HOME/nohup.out` – gelingt auch das nicht, scheitert der `nohup`-Aufruf.

[6] vgl. `http://en.wikipedia.org/wiki/Orphan_process`.

2.1.1.5 Prozessgruppen und Sessions

Neben der Prozess-ID (PID) hat jeder Prozess noch zwei weitere IDs, welche Zugehörigkeiten in zwei Gruppierungen ausdrücken:

- Die **Prozessgruppen-ID** (PGID) gibt eine Prozessgruppe an, der der Prozess angehört,
- Die **Session-ID** (SID) gibt eine Session an, zu der dieser Prozess gehört.

Im Regelfall vererben Prozesse ihre PGID und SID beim Erzeugen eines neuen Prozesses an den Sohnprozess. Prozesse können nun eine neue Prozessgruppe gründen (die Gruppen-ID ist dann identisch mit der Prozess-ID des Prozesses, der diese Entscheidung trifft), und sie können auf gleiche Weise eine neue Session starten.

Prozessgruppen und Sessions helfen dabei, zusammengehörende Prozesse leichter zu identifizieren. Sessions starten typischerweise mit der jeweils ersten Shell, die nach einer Anmeldung aufgerufen wird: Eine Session entspricht also meist einer „Session" im traditionellen Sinn, also einer Sitzung am Computer, die mit der Anmeldung beginnt, dann den Start diverser Programme nach sich zieht und schließlich durch das Abmelden beendet wird.

Mit Prozessgruppen ist innerhalb einer Session eine noch feinere Unterscheidung möglich, beispielsweise gehören meist alle Programme, die Teil einer Pipeline sind, zur gleichen Prozessgruppe.

```
> ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
19287  7628  7628 19287 pts/8    19287 S      500   0:00 /bin/sh /usr/bin/mozilla -mail
 7628  7637  7628 19287 pts/8    19287 Sl     500  20:50 /opt/moz/lib/mozilla-bin -mail
 9634 10095 10095 10095 tty1     10114 Ss     500   0:00 -bash
10095 10114 10114 10095 tty1     10114 S+     500   0:00 /bin/sh /usr/X11R6/bin/startx
10095 10115 10114 10095 tty1     10114 S+     500   0:00 tee /home/esser/.X.err
10114 10135 10114 10095 tty1     10114 S+     500   0:00 xinit /home/esser/.xinitrc
10135 10151 10151 10095 tty1     10114 S      500   0:00 /bin/sh /usr/X11R6/bin/kde
10151 10238 10151 10095 tty1     10114 S      500   0:00 kwrapper ksmserver
10258 10270 10270 10270 pts/2    10270 Ss+    500   0:00 bash
10276 10278 10278 10278 pts/4    10278 Ss+    500   0:00 bash
10260 10284 10284 10284 pts/5    10284 Ss+    500   0:00 bash
10275 10292 10292 10292 pts/6    10989 Ss     500   0:00 bash
10259 10263 10263 10263 pts/1    10263 Ss+    500   0:00 bash
10263 28869 28869 10263 pts/1    10263 S      500   0:16 konqueror /media/usbdisk/dcim
10263 28872 28872 10263 pts/1    10263 S      500   0:13 konqueror /home/esser
29201 29203 29203 29203 pts/7    29203 Ss+    500   0:00 bash
 4822  4823  4823  4823 pts/14    4823 Ss+    500   0:00 -bash
 4823 31118 31118  4823 pts/14    4823 S      500   0:00 nedit kernel/sched.c
 4823 31297 31297  4823 pts/14    4823 S      500   0:00 nedit kernel/fork.c
23115 32703 32703 23115 pts/13   32703 R+     500   0:00 ps j
```

Abbildung 2.1: Prozessgruppen und Sessions.

2.1.1.6 Threads in der Shell

Ob ein Prozess „multi-threaded" ist, also aus mehreren Threads besteht, zeigt `ps` mit Standardparametern nicht nur durch ein unscheinbares `l` (kleiner Buchstabe „L", steht für **l**ightweight process) in der Statusspalte an. Detailreicher ist die Ausgabe, wenn Sie die Optionen `-eLf` einsetzen.[7]

```
$ ps auxw
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    17040  0.0  0.0   3580   380 pts/17   Sl+  18:13   0:00 thread1

$ ps -eLf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
esser    16656 25983 16656  0    3 17:55 pts/17   00:00:00 thread1
esser    16656 25983 16657  0    3 17:55 pts/17   00:00:00 thread1
esser    16656 25983 16672  0    3 17:55 pts/17   00:00:00 thread1
```

Die Option `-L` (wieder: **L**ightweight process) führt dabei zur Ausgabe aller Threads und einer zusätzlichen Spalte `LWP`, welche die Thread-ID enthält.

Bei einem Prozess mit mehreren Threads ist der Prozess selbst als erster Thread verzeichnet; seine Thread-ID ist mit der Prozess-ID identisch. Alle weiteren Threads haben separate Thread-IDs, sind aber an der gemeinsamen PID erkennbar.

2.1.2 Prozesse aus Programmierersicht

In diesem Abschnitt betrachten wir die Systemaufrufe, die Sie benötigen, um Prozesse (in der Programmiersprache C) zu erzeugen und damit sinnvoll zu arbeiten.

2.1.2.1 fork()

Der wichtigste Befehl für C-Programmierer ist `fork()`: Er erzeugt einen neuen Prozess, der eine fast identische Kopie des erzeugenden Prozesses ist – der einzige Unterschied ist, dass der neue Prozess eine eigene Prozess-ID hat.

Der engliche Begriff *fork* (Gabel) wird für diesen Vorgang verwendet, weil sich der Programmverlauf eines einzelnen Prozesses beim „Forken" aufgabelt: Im Zeitpunkt der Trennung in die Gabelzacken (Vater- und Sohnprozess) sind die beiden noch identisch, denn beide haben ja bis dahin den gleichen Weg zurückgelegt. Ab der Gabelung geht dann jeder Prozess seinen eigenen Weg.

[7] Beachten Sie, dass andere Unix-Versionen auch andere `ps`-Parameter verwenden, um Thread-Informationen anzuzeigen.

Nach dem `fork()`-Aufruf werden beide Prozesse an der Stelle unmittelbar hinter dem `fork()`-Kommando fortgesetzt. Ob es sich um Vater oder Sohn handelt, erkennen die Prozesse am Rückgabewert von `fork()`:

- Der Rückgabewert 0 bedeutet: Dieser Prozess ist der Sohn. (Seine eigene Prozess-ID kann der Sohnprozess über `getpid()` herausfinden, falls er sie benötigt.
- Ein Rückgabewert $p > 0$ bedeutet: Dieser Prozess ist der Vater, und er hat gerade mit `fork()` einen neuen Prozess mit der PID p erzeugt.
- Der Vollständigkeit halber bedeutet ein negativer Rückgabewert, dass die Prozesserzeugung fehlgeschlagen ist.

Mit einem kleinen Beispiel kann man schnell überprüfen, dass das funktioniert:

```
main()
{
  int pid=fork();   /* Sohnprozess erzeugen */
  if (pid == 0)
     {
       printf("Ich␣bin␣der␣Sohn,␣meine␣PID␣ist␣%d.\n", getpid() );
     }
  else
     {
       printf("Ich␣bin␣der␣Vater,␣mein␣Sohn␣hat␣die␣PID␣%d.\n", pid);
     }
}
```

Speichern Sie dieses Listing als `simplefork.c`, übersetzen Sie es unter Linux mit `gcc -o simplefork simplefork.c` und starten Sie das resultierende Programm mit `./simplefork`.

Im Ergebnis erscheinen zwei Zeilen Text, aus denen man die korrekte „familiäre" Beziehung zwischen den beiden Prozessen ablesen kann:

```
> ./simplefork
Ich bin der Sohn, meine PID ist 23504.
Ich bin der Vater, mein Sohn hat die PID 23504.
```

In diesem Beispiel hat erst der Sohn, also der neu erzeugte Prozess, seine Textzeile ausgegeben und danach der Vater. Was passiert, wenn der Sohn für seine Arbeit länger braucht? Um das auszuprobieren, fügen wir einen `sleep()`-Aufruf[8] in das Programm ein:

[8] Für `sleep()` binden Sie unter Linux die Header-Datei `unistd.h` ein.

```
#include <unistd.h>          /* sleep()                          */

main()
{
  int pid=fork();            /* Sohnprozess erzeugen             */
  if (pid == 0)
     {
       sleep(2);             /* 2 sek. schlafen legen            */
       printf("Ich␣bin␣der␣Sohn,␣meine␣PID␣ist␣␣%d\n", getpid() );
     }
  else
     {
       printf("Ich␣bin␣der␣Vater,␣mein␣Sohn␣hat␣die␣PID␣␣%d\n", pid);
     }
}
```

Wenn Sie diese Variante übersetzen und starten, geschieht etwas Unerwartetes: Das Programm erzeugt den Sohnprozess, der sich erst mal schlafen legt – dadurch kommt gleich wieder der Vaterprozess an die Reihe, gibt seine Textzeile aus und beendet sich. Es erscheint wieder der Shell-Prompt. Erst nach den zwei Sekunden Verzögerung erwacht der Sohnprozess wieder zum Leben, gibt nun seinerseits die Textzeile aus (die sehr unordentlich, gleich hinter dem Eingabeprompt erscheint) und beendet sich dann ebenfalls.

Das sieht auf der Konsole dann so aus:

```
esser@sony:Tests/fork> ./simplefork
Ich bin der Vater, mein Sohn hat die PID  24130
esser@sony:Tests/fork> Ich bin der Sohn, meine PID ist  24130
```

2.1.2.2 wait()

Um dieser Unordnung aus dem Weg zu gehen, verwenden Sie den `wait()`-Befehl: Er lässt den aufrufenden Prozess solange warten, bis ein Sohnprozess beendet wurde.[9]

Im `else`-Zweig fügen Sie dazu den Befehl `wait()` ein:

```
  else
     {
       printf("Ich␣bin␣der␣Vater,␣mein␣Sohn␣hat␣die␣PID␣␣%d\n", pid);
       wait();          /* auf Sohn warten */
     }
```

[9] Falls es gar keine Kinder gibt, wird der `wait()`-Aufruf ignoriert.

Jetzt funktioniert das Programm besser – es erscheint zunächst die Zeile des Vaterprozesses und nach einer zweisekündigen Pause die des Sohns. Erst danach beendet sich der Vaterprozess, und die Shell zeigt wieder den Prompt an.

Mit `pstree` und `ps` können Sie überprüfen, dass Linux wirklich zwei Prozesse erzeugt und diese in einer Vater-Sohn-Beziehung stehen:

```
esser@sony:Skript> pstree|grep simple
     |         |         `-bash---simplefork---simplefork
esser@sony:Skript> ps w|grep simp
25684 pts/16   S+     0:00 ./simplefork
25685 pts/16   S+     0:00 ./simplefork
```

Deutlich zu sehen ist in der `pstree`-Ausgabe, dass die Shell (`bash`) einen ersten `simplefork`-Prozess und dieser einen gleichnamigen zweiten erzeugt hat; in der Prozessliste (`ps`) erscheinen entsprechend zwei `simplefork`-Prozesse mit aufeinander folgenden Prozess-IDs.[10]

2.1.2.3 exec()

Alle Programmstarts laufen so ab, dass zunächst mit `fork()` ein neuer Prozess erzeugt wird – dieser ist aber weitgehend mit dem Vaterprozess identisch, insbesondere läuft das gleiche Programm darin. Um nun ein anderes Programm im neuen Prozess auszuführen, wird ein System Call aus der `exec()`-Familie aufgerufen: Er lädt den Programmcode aus einer ausführbaren Datei von der Festplatte und überschreibt damit den Programmcode-Bereich im Kindprozess. Zudem werden hier die Speicherbereiche des Prozesses auf Standardwerte zurückgesetzt, und schließlich kann das neue Programm starten.

Im folgenden einfachen Beispiel erzeugt das Hauptprogramm mit `fork()` einen neuen Prozess, der dann (`pid==0`) mit `execl()` den Editor `vi` aufruft. Beim Programmstart erhält der Editor als Parameter den Dateinamen `/etc/fstab`.

```
#include <unistd.h>
int main () {
  int pid=fork();
  if (pid==0) {
    /* Code für Sohnprozess */
    execl ("/usr/bin/vi", "vi", "/etc/fstab", (char *) NULL);
    perror ();  /* Diese Zeile sollte nicht erreicht werden! */
  } else {
    /* Code für Vaterprozess */
    wait ();
```

[10] Waren Sie nicht schnell genug, erhöhen Sie einfach im `sleep()`-Aufruf die Wartezeit von 2 auf 10 Sekunden.

```
  }
  exit(0);
}
```

Der Vaterprozess (`pid!=0`) wartet hier wieder nur auf die Terminierung des Sohnprozesses.

`execl()` ist einer von mehreren Aufrufen der „`exec()`-Familie". Insgesamt gibt es die folgenden (deren genaue Beschreibungen Sie in der `exec`-Manpage `exec(3)` finden):

- `int execl(const char *path, const char *arg, ...);`
  erwartet als erstes Argument den vollen Pfad zur auszuführenden Datei, dahinter eine beliebig lange Liste von Argumenten `arg0`, `arg1`, ... Dabei ist `arg0` der Programmname selbst (das Programm erhält den Namen ebenfalls als Argument, damit es erkennen kann, unter welchem Namen es aufgerufen wurde) – erst ab `arg1` werden die Argumente als richtige Programmparameter behandelt.

- `int execlp(const char *file, const char *arg, ...);`
  wie `execl()`, der Programmname kann aber ohne Pfad angegeben werden (z. B. `"emacs"` statt `"/usr/X11R6/bin/emacs"`), wenn das Verzeichnis, welches das Programm enthält, in der Pfadvariable `$PATH` definiert ist, z. B.

  ```
  #include <unistd.h>
  main() {  /* Datei /etc/fstab in emacs oeffnen - ohne Pfadangabe */
    execlp ("emacs", "emacs", "/etc/fstab", (char *)0);
  }
  ```

- `int execle(const char *path, const char *arg, ...,`
  `char * const envp[]);`
  wie `execl()`, erlaubt aber zusätzlich das Setzen der Umgebungsvariablen, die für das nachgeladene Programm gelten sollen; bei `execl()` bleiben die alten Werte erhalten, zum Beispiel:

  ```
  #include <unistd.h>
  main() {  /* Nur Var. HOME und LOGNAME im Environment setzen */
    char *env[] = { "HOME=/home/user", "LOGNAME=user", (char *)0 };
    execle ("/bin/ls", "ls", "-l", (char *)0, env);
  }
  ```

- `int execv(const char *path, char *const argv[]);`
  `int execvp(const char *file, char *const argv[]);`
  `int execve(const char *file, char *const argv[]);`

Varianten von `execl()`, `execlp()` und `execle()`, bei denen die Argumente als Array übergeben werden, also z. B.

```
#include <unistd.h>
main() {
  char *cmd[] = { "ls", "-l", (char *)0 };
  char *env[] = { "HOME=/home/user", "LOGNAME=user", (char *)0 };
  execve ("/bin/ls", cmd, env);
}
```

2.1.2.4 Python-Programmierung

Auch in Python stehen die Funktionen `fork()`, `wait()` und `exec()` zur Verfügung, wenn Sie diese über das `os`-Modul nachladen. Ein einfaches Python-Programm, das einen Sohnprozess erzeugt und auf dessen Terminierung wartet, sieht wie folgt aus:

```
#!/usr/bin/python

import os

print "Hello World"
chid=os.fork()    # chid = child ID
if chid == 0:     # Kind-Prozess
  os.execl ("/bin/ls","ls","/tmp")
else:             # Vater-Prozess
  os.wait()
  print "Das war's"
```

2.1.3 Programmieren mit Threads

Neben Prozessen gibt es noch das Konzept der **Threads**: Auch hier geht es um Parallelität, aber die Nebenläufigkeit findet innerhalb eines einzelnen Prozesses statt. Eine ausführliche Behandlung dieses Konzepts (und verschiedener Methoden, es umzusetzen) folgt im Theorieteil ab Kapitel 2.2.3 (Seite 38) – hier stellen wir Möglichkeiten für Programmierer unter Linux vor.

Ein wichtiger Vorteil von Threads gegenüber Prozessen ist, dass sie einen gemeinsamen Speicherbereich haben: den des Prozesses, zu dem sie gehören. Wenn nämlich ein Prozess mit `fork()` einen Sohnprozess erzeugt, haben zwar anfangs beide Prozesse die gleichen Variablen (weil der Sohn als exakte Kopie des Vaters entsteht) – Änderungen an den Variablen in einem der beiden Prozesse wirken sich aber nicht auf den anderen Prozess aus. Falls ein Datenabgleich notwendig ist, müssen die Prozesse sich daher mit Interprozess-Kommunikation darum kümmern. Bei Threads ist

es anders: Eine Änderung einer globalen (Prozess-) Variable durch einen Thread ist für alle anderen Threads sofort sichtbar.

Auch die Steuerung der Ausführung nach dem Erzeugen des Threads funktioniert anders als beim „Forken" eines Kindprozesses:

- Nach einem `fork()` führen Vater- und Kindprozess den gleichen Code (nämlich den, der auf den `fork()`-Aufruf folgt) aus – über den Rückgabewert von `fork()` erkennen beide Prozesse, ob sie Vater oder Sohn sind und können entsprechend verschiedene Aufgaben übernehmen.
- Ein Thread stellt keine Kopie des erzeugenden Prozesses dar, sondern ist ein Teil des Prozesses. Beim Aufruf gibt der Programmierer daher eine Funktion an, die innerhalb des Threads laufen soll.

Einen Thread zu erzeugen, ist also grundsätzlich vergleichbar mit einem normalen Funktionsaufruf

```
do_something();
```

– mit dem Unterschied, dass die Funktion als eigenständiger Thread arbeitet. Der Programmfluss teilt sich also in zwei Stränge auf: Der aufrufende Codeblock wird hinter dem Thread-erzeugenden Aufruf

```
pthread_create(..., do_something, ...);
```

fortgesetzt, und parallel startet ein neuer Thread mit der ersten Anweisung in der Funktion `do_something()`.

2.1.3.1 Thread-Beispiel in C

Das folgende Beispiel wurde aus einem `pthread`-Tutorial von IBM[11] übernommen:

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_function(void *arg) {
  int i;
  for ( i=0; i<20; i++ ) {
    printf("Thread␣says␣hi!\n");
    sleep(1);
  }
```

[11] `http://www-128.ibm.com/developerworks/library/l-posix1.html`

```
  return NULL;
}

int main(void) {
  pthread_t mythread;

  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error␣creating␣thread.");
    abort();
  }

  if ( pthread_join ( mythread, NULL ) ) {
    printf("error␣joining␣thread.");
    abort();
  }

  exit(0);
}
```

Die Funktionen `pthread_create()` und `pthread_join()` übernehmen für Threads ähnliche Aufgaben wie `fork()` und `wait()` für Prozesse.

2.1.3.2 Thread-Beispiel in Python

Auch in Python lassen sich Threads auf leichte Weise erzeugen, dafür stehen zwei Thread-Bibliotheken zur Verfügung.

Die simplere Variante verwendet die Bibliothek `thread` und benötigt nur den folgenden Befehl, um eine Funktion in einem neuen Thread laufen zu lassen:

```
thread.start_new_thread ( funktion, args )
```

Dabei ist `args` ein Tupel, das die Argumente für `funktion` enthält; ein Beispiel für die Parameterübergabe sieht so aus:

```
def multipliziere (x,y):
  print x, "*", y, "=", x*y

thread.start_new_thread ( multipliziere, (2,3) )
```

Damit entspricht `start_new_thread` der C-Funktion `pthread_create()`. Zur Join-Funktion `pthread_join()` gibt es in der `thread`-Klasse kein Gegenstück. Erschwerend kommt hinzu, dass mit dem Ende des Hauptprogramms auch alle Threads beendet werden.

Mehr Möglichkeiten bietet die `threading`-Bibliothek, die eine neue Klasse `Thread` bereitstellt. Threads werden hier als Klassen implementiert, die von `Thread` erben und als wesentlichsten Aspekt eine Methode `run(self)` implementieren müssen, welche die eigentliche Thread-Funktion enthält. Dazu kommt noch eine Funktion `__init__(self,parameter)`, die beim Erzeugen eines neuen Thread-Objekts ausgeführt wird – hier lassen sich auch Parameter einsetzen, die dann für die Initialisierung des Objekts genutzt werden – im folgenden Code-Beispiel gehört etwa zu jedem `watcher`-Thread ein Dateiname:

1. Über `newwatcher=watcher("/tmp/x.txt")` wird ein neuer Thread erzeugt,
2. Der Parameter `"/tmp/x.txt"` wird als `filename` an `__init__(self,filename)` übergeben – die Init-Methode verwendet dieses Argument, um eine private Variable zu setzen.
3. Im Hauptprogramm erfolgt wird schließlich mit `newwatcher.start()` die `run`-Methode des neuen Threads aktiviert: Der Thread läuft.

Das komplette Listing, das ein krudes Tool zur Logfile-Beobachtung implementiert, folgt.

```
#!/usr/bin/python

# watcher.py
# Version 1.0 (2006-11-07)

import os
from threading import Thread
from time import sleep

class watcher(Thread):
   def __init__ (self,filename):
      Thread.__init__(self)
      # private Variablen initialisieren
      self.filename = filename
      self.lastline = ""
      self.ExitNow = False
   def exitnow(self):
      self.ExitNow = True
   def run(self):
      while not self.ExitNow:
        try:
           # über externen tail-Befehl die letzte Zeile einer Datei ↩
                ...lesen
           f = os.popen("tail -1 "+self.filename, "r")
           self.lastline = f.readline()
```

```
        except:
           self.lastline = "READ␣ERROR"
        sleep(1)

def add_watcher (filename):
   global watchno, watchers
   if not os.path.exists (filename):
      print "Fehler:␣Datei␣existiert␣nicht"
   else:
      try:
         nul = file(filename).readline()
      except:
         print "Fehler:␣Datei␣nicht␣lesbar"
         return()
      newwatcher = watcher(filename)
      watchers.append(newwatcher)
      watchno+=1
      newwatcher.start()

def remove_watcher (filename):
   global watchno, watchers
   found = False
   for w in watchers:
      if w.filename == filename:
         w.exitnow()
         watchers.remove(w)
         watchno-=1
         found = True
   if not found:
      print "Fehler:␣Es␣gab␣keinen␣Watcher␣fuer␣diese␣Datei"

def status ():
   global watchno, watchers
   print "Status:␣Anzahl␣Watchers:", watchno
   for w in watchers:
      print w.filename+":", w.lastline[:50]

def printhelp ():
   print  """\
watch␣<Dateiname>␣␣␣␣␣-␣Datei␣in␣Watch␣List␣aufnehmen
unwatch␣<Dateiname>␣␣␣-␣Datei␣aus␣Watch␣List␣entfernen
status␣/␣s␣␣␣␣␣␣␣␣␣␣␣-␣Statusinformationen␣ausgeben
help␣/␣?␣␣␣␣␣␣␣␣␣␣␣␣-␣diese␣Hilfe␣anzeigen
quit␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣Programm␣beenden"""

watchers = []
watchno = 0
```

```
while 1:
   prompt="["+str(watchno)+"]>␣"
   command = raw_input(prompt)
   if command == "": continue
   try:
      cmd,arg = command.split()
   except ValueError:
      cmd = command
   if   cmd == "watch":         add_watcher (arg)
   elif cmd == "unwatch":       remove_watcher (arg)
   elif cmd in ["status","s"]: status ()
   elif cmd in ["help","?"]:   printhelp ()
   elif cmd == "quit": break
   else: print "Fehler:␣unbekannter␣Befehl"

if watchno > 0:
   for w in watchers:
      w.exitnow()
   print "Warte␣3␣Sekunden..."
   sleep(3)

for w in watchers:
   w.join()
print "Ende"
```

Die `threading`-Bibliothek stellt eine `join`-Funktion bereit, die im obigen Beispiel in den letzten Zeilen verwendet wird.

Zudem gibt es die Funktion `activeCount()`, welche die Anzahl aller aktiven `Thread`-Objekte zurückgibt – damit lässt sich feststellen, ob das Programm gefahrlos beendet werden darf.

2.2 Theorie

In diesem Abschnitt beschäftigen wir uns mit den theoretischen Grundlagen und auch mit der geschichtlichen Entwicklung des **Multitasking**-Konzepts.

Schon früh in der Computer-Geschichte, als die Maschinen noch mit Programmen und Daten auf Lochkarten gefüttert wurden, entstand das Konzept eines „Jobs": Ein Job war damals ein auszuführendes Programm mit seinen Daten, das in den Computer eingelesen und abgearbeitet wurde. War der Job zu Ende, schrieb er noch seine Ergebnisdaten auf eine andere Lochkarte und ging dann zum nächsten Job über. Da man die Lochkarten auf einen Stapel legen konnte, der dann vom Rechner automatisch verarbeitet wurde, entstand für diese Vorgehensweise der Begriff Stapelverarbeitung.[12]

Prägend für diese frühen Maschinen war aber die volle Konzentration auf den aktuellen Job. Nach jedem Jobende konnten alle Informationen im Speicher des Rechners gelöscht werden – für den nächsten Job wurde die Maschine wieder in den Ausgangszustand versetzt.

Seit moderne Betriebssysteme es erlauben (wollen), mehrere Arbeiten quasiparallel auszuführen, ist es nötig, sich etwas mehr Gedanken über die Verwaltung der Jobs (oder Prozesse) zu machen.

2.2.1 Was schon ein einzelner Prozess benötigt

Betrachten wir zunächst den Fall, in dem auf dem Computer ein einzelnes Programm läuft, das auf den gesamten Hauptspeicher zugreifen darf (wie es z. B. bei MS-DOS der Fall ist). Es belegt einen Teil des Speichers mit seinem eigenen Programmcode, und den Rest kann es flexibel für die Daten und die Berechnungen auf diesen Daten nutzen. Die meisten Programme sind strukturiert und gliedern Teilaufgaben in Prozeduren oder Funktionen aus.[13]

Wenn eine Funktion aufgerufen wird, muss der Prozess sich die Rücksprungadresse merken: Das ist die Adresse, an der der nächste Befehl nach dem Funktionsaufruf steht – dort geht es weiter, wenn die Funktion beendet wird.

Neben Rücksprungadressen gibt es weitere Daten, die bei jedem Funktionsaufruf zu sichern sind: So muss zum Beispiel ein Weg gefunden werden, Funktionsargumente an einer geeigneten Stelle abzulegen, aus der die Funktion sie dann auslesen kann. Auch für ihre eigenen lokalen Variablen braucht sie Platz – und das möglicherweise mehrfach, falls die Funktion auch rekursiv arbeitet (sich also selbst aufruft).

[12] siehe `http://de.wikipedia.org/wiki/Stapelverarbeitung`

[13] Auch objektorientierte Programmiersprachen sind letzten Endes prozedural, denn eine Methode ist mit einer Funktion vergleichbar.

Für all diese Daten verwendet man den **Stack**, einen Speicherbereich, der nicht durch andere Daten überschrieben werden darf. Der Stack wächst mit jedem Funktionsaufruf. Ruft also das Hauptprogramm `main()` die Funktion `a()` auf und diese dann `b()`, dann liegen auf dem Stack zwei **Stack Frames** – einer für `a()` und einer für `b()`.

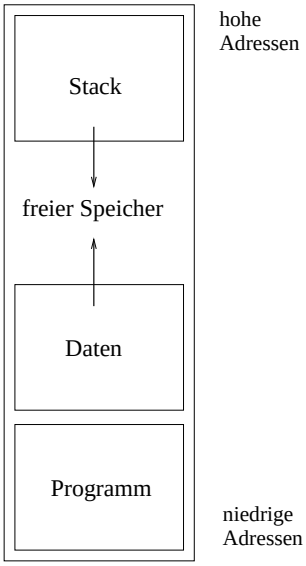

Abbildung 2.2: Speichernutzung eines einzelnen Prozesses.

Abbildung 2.2 zeigt die übliche Art und Weise, wie sich Programm, Daten und Stack den Speicher teilen: Der Programmcode selbst ist unveränderlich und liegt am „Anfang" des zur Verfügung stehenden Speichers – noch darunter finden sich noch das Betriebssystem und dessen Daten.

Die dynamischen Daten und der Stack beginnen dann an den gegenüberliegenden Enden des restlichen Speichers und wachsen zur Mitte hin aufeinander zu. Das bedeutet für den Stack, dass der **Stack Pointer**, der auf die aktuelle Grenze des Stacks zeigt, immer kleiner wird, wenn der Stack wächst.

2.2.2 Multitasking: Prozesse, Hierarchien, Kontext

Einige der Eigenschaften eines Prozesses kann man leicht aus der Anschauung herleiten:

Wenn mehrere Prozesse gleichzeitig im Hauptspeicher liegen und abwechselnd ausgeführt werden sollen, braucht jeder Prozess einen separaten Speicherbereich, in dem

sowohl der Programmcode, die Daten, die es verarbeiten soll, als auch der Stack mit den Parametern und Rücksprungadressen liegen. Auch an welcher Stelle der Programmausführung ein Prozess gerade angekommen ist, muss sich das Betriebssystem merken – dafür gibt es den Befehlszähler. Dazu kommen noch Prozessorregister, die Programme für alle möglichen Aufgaben verwenden, beispielsweise einfache Additionen.

Ein Prozess besteht also aus folgenden Komponenten:

- Separater Adressraum für diesen Prozess
- ausführbares Programm, das in diesen Adressraum geladen wurde
- Programmdaten (Variableninhalte, auch in diesem Adressraum)
- Stack (dito)
- Befehlszähler (Program Counter, PC) und Stack-Pointer
- Inhalt der Hardware-Register (Hardware-Kontext)

Aus Sicht der Betriebssystems kommen noch weitere Verwaltungsinformationen hinzu, beispielsweise die Prozess-ID, Informationen über zugeteilten Speicher, geöffnete Dateien und der Prozess-**Zustand**.

Jeder Prozess ist stets in einem von mehreren Zuständen – welche bzw. wie viele es davon gibt, hängt vom konkreten Betriebssystem ab, aber zumindest die folgenden sind weitgehend standardisiert:

laufend / running Der Prozess wird gerade ausgeführt: Der Prozessor führt seine Befehle aus. Auf einem Ein-Prozessor-System kann immer nur ein einziger Prozess in diesem Zustand sein.

bereit / ready Der Prozess ist bereit, seine Arbeit fortzusetzen, und wartet darauf, dass ihm wieder der Prozessor zugeteilt wird. Sobald das geschieht, wechselt er in den Zustand *laufend*.

blockiert / blocked, waiting Der Prozess ist nicht bereit – er wartet auf ein Ereignis, z. B. den Abschluss einer (vergleichsweise) zeitaufwendigen I/O-Operation. Sobald das geschieht, wechselt er in den Zustand *bereit*.

Beim Übergang von *laufend* nach *bereit* entzieht das Betriebssystem (genauer: dessen so genannter Scheduler, siehe Kapitel 4) dem aktuellen Prozess den Prozessor, um einen anderen Prozess an die Reihe kommen zu lassen – beim umgekehrten Übergang ist der Prozess der glückliche Nächste, der Rechenzeit erhält.

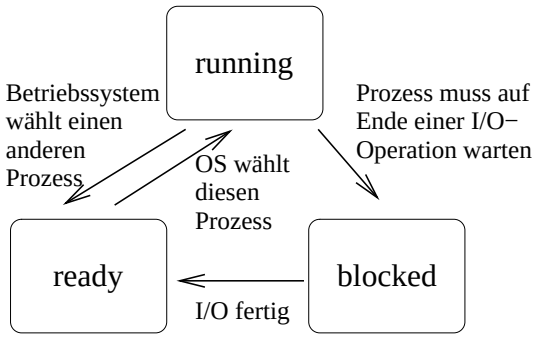

Abbildung 2.3: Speichernutzung eines einzelnen Prozesses.

Abbildung 2.3 zeigt das Übergangsdiagramm für diese drei Zustände. Daneben gibt es weitere Zustände, beispielsweise

suspendiert / suspended Der Prozess wäre zwar prinzipiell bereit, wurde aber durch den Anwender oder Administrator angehalten (Linux: `kill -SIGSTOP` bzw. Tastenkombination [Strg-Z]).

schlafend / sleeping Der Prozess wartet auf ein Signal von einem anderen Prozess.

ausgelagert / swapped-out Der Speicherbereich des Prozesses liegt nicht mehr im Hauptspeicher, sondern wurde auf die Festplatte ausgelagert, weil im RAM nicht mehr genügend Speicherplatz frei war.

2.2.2.1 Prozesshierarchie

Wie bereits am Anfang dieses Kapitels beschrieben, sind Prozesse in einer Hierarchie organisiert, die sich aus den Vater-Sohn-Beziehungen ergibt, die bei der Prozesserzeugung entstehen. Gemeinsame Söhne des gleichen Vaterprozesses heißen auch Geschwister.

Da sich in diesem Prozessbaum jeder Prozess, sobald er seine Aufgaben erledigt hat, beendet und seinen Rückgabewert an den Vaterprozess zurückgibt, kann es keinen Prozess ohne Vater geben – wird also ein Prozess gewaltsam beendet, muss es allen Söhnen, „Enkeln" etc. genauso gehen: Der gesamte Unterbaum (mit dem abgebrochenen Prozess als Wurzel) muss ebenfalls sofort beendet werden.

Ein Prozess kann allerdings seine „Vaterschaft" aufgeben (unter Linux in der Shell mit dem Befehl `disown`), dann geht die Vaterschaft auf einen anderen Prozess (meist

die Wurzel des Prozessbaums) über, und der ehemalige Sohnprozess kann auch beim Abbruch oder beim regulären Beenden des Vaters weiter arbeiten.

2.2.2.2 Prozesslisten

Die Frage, wie man den Prozessbaum in den internen Strukturen speichert, beantwortet jedes Betriebssystem unterschiedlich. Typisch sind z. B. Listen, in denen die Prozesse stehen, welche in einem bestimmten Zustand sind – also etwa eine Ready-Liste für Prozesse, die ausführbereit sind. Solche Listen können dann auch gleich dem Scheduler als Warteschlangen dienen.

2.2.2.3 Process Control Block

Statusinformationen über einen Prozess speichert das Betriebssystem meist in einer Struktur namens **Process Control Block** (PCB, auch Task Control Block), die unter anderem folgende Daten enthält:

- Prozess-ID (PID)
- Registerwerte (darunter den **Program Counter**), die vor dem Aktivieren des Prozesses in die Prozessorregister geladen werden müssen
- Informationen zu vom Prozess genutzten Speicherbereichen
- Priorität und/oder andere für den Scheduler wichtige Parameter, z. B. statistische Daten wie die bisherige Gesamtlaufzeit
- Informationen zu offenen Dateien und Sockets

In den Prozesslisten finden sich dann meist Zeiger auf die PCBs der Prozesse, oder die PCBs sind selbst als solche Listen organisiert (dann enthält die PCB-Struktur etwa einen Zeiger auf den nächsten PCB in der aktuellen Liste).

2.2.3 Threads

Threads stellen eine Möglichkeit dar, Prozesse nochmals in mehrere Aktivitätsstränge zu untergliedern, die separat abgearbeitet werden. Anders als mehrere unabhängige Prozesse können die Threads aber alle zu einem gemeinsamen Prozess gehören und sich damit den Speicherbereich teilen, den dieser Prozess nutzt.

Prozesse haben ja stets getrennte Speicherbereiche, und das Betriebssystem sorgt dafür, dass kein Prozess die Daten eines anderen Prozesses lesen oder gar verändern kann. Gerade das ist aber oft erwünscht.

Beispiel 2.1 Wenn ein Prozess eine interaktive Texteingabe erlaubt (etwa in einem Mail- oder Textverarbeitungsprogramm) und ein zweiter Prozess die soeben eingetippten Worte auf korrekte Rechtschreibung überprüfen soll, dann ist dies im klassischen Prozessmodell nur durch ständigen Versand von Nachrichten zwischen den beiden Prozessen möglich (siehe Kapitel 6 über Inter-Prozess-Kommunikation).

Verwendet man für die beiden Aufgaben (Texteingabe, Rechtschreibkorrektur) nun zwei Threads, die zu einem gemeinsamen Prozess gehören, wird die Aufgabe einfacher: Beide Threads greifen auf den gleichen Speicher zu, der Rechtschreib-Thread kann also einfach den Speicher überwachen, in dem der Texteingabe-Thread die Zeichen ablegt – taucht dort ein neues Wort auf, kann es direkt überprüft werden. ■

Für die Thread-Implementation durch das Betriebssystem gibt es verschiedene Möglichkeiten:

User Level Threads Das Betriebssystem kann ganz auf ein Thread-Konzept verzichten und es den Programmen überlassen, solche Strukturen einzuführen. Bei Programmen, die mit User Level Threads arbeiten, muss eine Thread-Bibliothek eingebunden und verwendet werden, die Funktion zum Thread-Erzeugen und -Manipulieren bereitstellt, wie es unter Linux z. B. das `pthread`-Paket tut. Ein Prozess, der intern in User Level Threads untergliedert ist, muss sich also selbst darum kümmern, die einzelnen Threads der Reihe nach laufen zu lassen.

Der Vorteil dieses Verfahrens ist, dass beim Umschalten zwischen einzelnen Threads nicht der Betriebssystem-Scheduler aktiv werden muss, denn das Betriebssystem hat ja gar keine Kenntnis von der Thread-Struktur.

Genau das führt aber auch zu dem Nachteil, dass beim Blockieren eines einzelnen Threads (der etwa eine I/O-Operation durchführt) der gesamte Prozess blockiert, denn das Betriebssystem sieht nur die blockierende Operation, die „irgendwoher" aus dem Prozess kommt, und muss dessen Status dann auf *wartend* setzen.

Kernel Level Threads Alternativ kann das Betriebssystem Threads selbst verwalten – es legt dann also statt oder neben einer Prozesstabelle eine Thread-Tabelle an. Für das Erzeugen neuer Threads und den Thread-Wechsel ist dann das Betriebssystem zuständig. Ein Umschalten zwischen zwei Threads des gleichen Prozesses ist dabei immer noch schneller erledigt als ein Prozesswechsel, weil sich z. B. nichts am benutzten Speicherbereich ändert – dennoch muss für jeden Thread-Wechsel über einen Systemaufruf in den privilegierten Modus und dann wieder zurückgeschaltet werden.

Ein Vorteil der Kernel Level Threads ist, dass beim Blockieren eines Threads die übrigen noch weiter laufen können, weil das Betriebssystem die Thread-

Struktur des Prozesses kennt und weiß, welche Teile blockiert sind und welche nicht.

Kernel Level Threads heißen auch **Lightweight Process**.

Gemischte Threads Es ist auch möglich, Kernel und User Level Threads miteinander zu vermischen, Sun Solaris macht das beispielsweise. Dabei erzeugen Prozesse über eine Thread-Bibliothek User Level Threads, und diese werden vom Betriebssystem Kernel Level Threads zugeordnet (die nicht in gleicher Anzahl vorhanden sein müssen).

Das ist besonders bei SMP-Maschinen sinnvoll, weil dann auch die Threads eines einzelnen Prozesses auf mehrere CPUs verteilt werden können.

Diese drei Varianten veranschaulicht Abbildung 2.4.

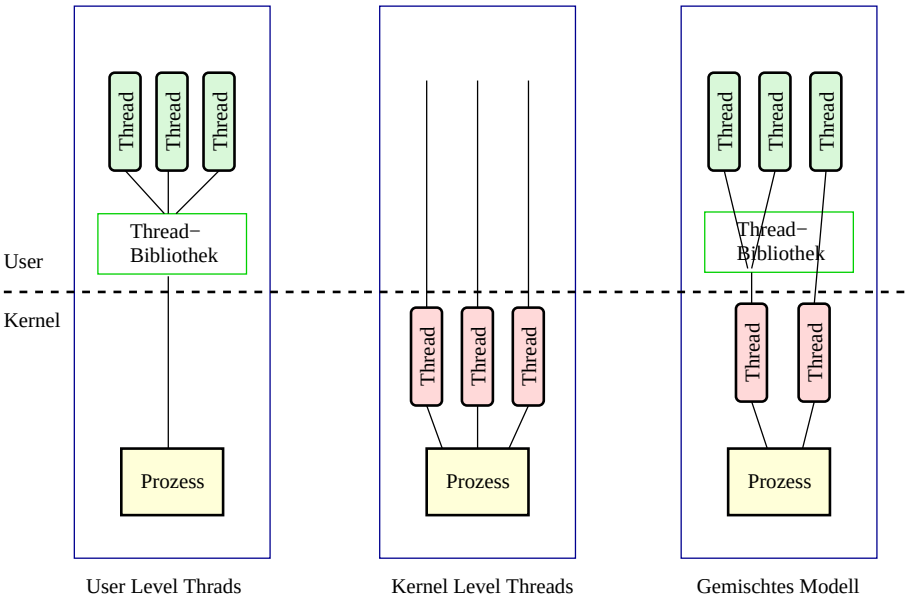

Abbildung 2.4: Drei Thread-Typen im Vergleich.

Auf einem Multi-Prozessor-System können mehrere Threads des gleichen Prozesses echt parallel ausgeführt werden. Es gibt Aufgaben, die von einer solchen Form der Parallelisierung besonders profitieren: Die Programme laufen dann maximal effizient, wenn eine bestimmte Anzahl Threads (aus denen sich das Programm zusammensetzt) stets gleichzeitig CPUs zugeteilt bekommt. Eine detailliertere Beschreibung dieser Thematik bietet das Scheduler-Kapitel 4 ab Seite 31.