öS 70,– sfr 8,50 **DM 8,50** H 10554 Oktober 1998

10

c't

MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK

Anschluß für die Firma:
Internet-Provider

IBMs S/390 und Siemens' BS2000:

Mainframes

Großrechner im Detail

LAN-Sicherheit:

Remote-Zugänge schützen

Update:

Debian GNU/Linux 2.0

Softwareentwicklung:

Sicherheit mit CORBA
Methoden für die UML

Web-Programmierung:

DB-Anbindung mit ASP

Virtual Reality:

Twosuns' audiovisuelle Projektionen

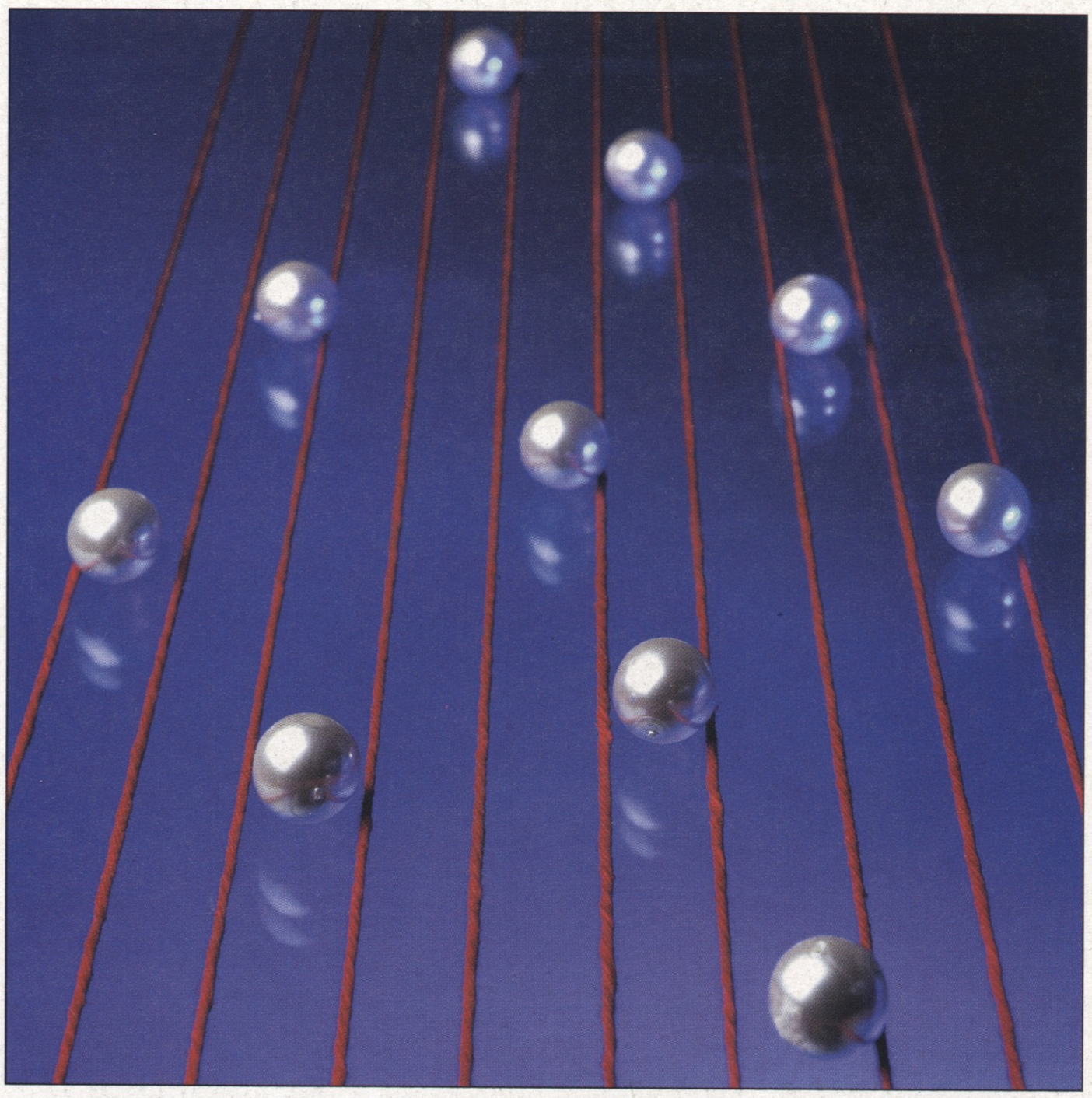
Getestet:

BioMouse nimmt Fingerabdruck
Transtecs 4-Prozessor-Xeon-Server
Smalltalk-Umgebung VisualWorks 3.0

4 399148 508506 10

Multithreading mit Perl 5.005

Aufgefädelt



Michael Schilli

Perl 5.005 ist da. Hinter dem Tausendstelsprung in der Versionsnummer verbergen sich verbesserte Windows-Kompatibiliät, ein Compiler-Kit und erstmals Threads. Was man mit ihnen anstellen kann, zeigt der folgende Überblick.

Sinnvoll eingesetzt, steigern Threads wie leichtgewichtige Prozesse die Produktivität eines Programms: Nicht nur auf Multiprozessormaschinen, die tatsächlich parallele Threads ausführen und damit im Idealfall die Performance ver-n-fachen, sondern auch auf ordinären 1-Prozessor-Systemen tragen die emsigen Wusler zur besseren Resourcennutzung bei.

In folgenden Situationen bringen Threads klare Vorteile:

- Es liegt ein Mehrprozessorsystem vor, und die gestellte Aufgabe läßt sich in unabhängig voneinander durchführbare Teilaufgaben zerlegen.
- Der Prozessor wartet länger auf Daten von langsamen Schnittstellen (zum Beispiel bei Netzzugriffen), bevor er diese aufwendigen Berechnungen unterzieht. Lassen sich die langsamen Vorgänge in parallele Threads abspalten, entfallen Wartephasen und selbst Ein-Prozessor-Systeme zeigen bessere Performance.
- Das System muß trotz starker Prozessorauslastung bedienbar bleiben, etwa bei GUIs.

Multithreading bringt allerdings eine ganze Reihe von Problemen mit sich: Leicht schleichen sich Programmierfehler ein, die aufgrund der Quasi-Parallelität kaum zu reproduzieren sind. Gibt es also durch ihren Einsatz nicht viel zu gewinnen, weil sich zum Beispiel die gestellte Aufgabe nicht entsprechend zerlegen läßt, bringen Threads mehr Verdruß als Vorteil.

Anders als per *fork()* erzeugte Prozesse, die sich jeweils in ihrem eigenen Adreßraum bewegen, sind Threads lediglich parallele Kontrollflüsse mit eigenem Stack. Sie schleppen nicht die ganze Prozeßumgebung mit sich herum, brauchen daher weniger Platz als ihre großen Brüder und lassen sich so haufenweise billig erzeugen. Daß Threads sich Variablen außerhalb des Stacks teilen (globale Variablen und Instanzvariablen von gemeinsam benutzten Objekten), vereinfacht zwar den Datenaustausch zwischen ihnen. Der Preis dafür ist jedoch zusätzliche Komplexität, denn der Zugriff auf sie muß synchronisiert werden.

Objekte abstrahieren Threads

Listing *chaos.pl* zeigt die Entstehung von Threads: Der Konstruktor *new* erwartet eine Referenz auf eine Subroutine und optional eine Reihe von Parametern, startet einen neuen Thread und ruft die angegebene Routine mit den spezifizierten Parameterwerten auf. Gelangt ein Thread ans Ende der Routine, terminiert er automatisch.

Schon seit dem Programmstart läuft der Main-Thread, der dreimal *new()* aufruft, so daß schließlich insgesamt vier Threads parallel im System laufen, von denen drei allerdings wegen *sleep(3)* für die nächsten drei Sekunden in *function()* träumen.

Der Konstruktor liefert eine Objektreferenz zurück, die der Main-Thread dazu verwendet, mittels *join()* auf das Ende des Abkömmlings zu warten. Ohne diese Zeilen wäre seine Arbeit erledigt, das Programm würde abbrechen und die noch laufenden Threads gnadenlos in Luft auflösen.

Welcher der drei Threads als erster erwacht, hängt von vielen Faktoren ab und läßt sich nicht vorhersagen. Lautet die Ausgabe von *chaos.pl* beispielsweise

```
12:00:00> Thread 1 started
12:00:00> Thread 2 started
12:00:00> Thread 3 started
12:00:03> Thread 2 ends
12:00:03> Thread 1 ends
12:00:03> Thread 3 ends
```

entstanden die drei Threads in weniger als einer Sekunde (Zeitpunkt 12:00:00). Während der Main-Thread auf den ersten wartete, erwachte etwa drei Sekunden später Thread 2 – zufällig noch bevor Thread 1 und danach Thread 3 wieder aktiv wurden.

Dies kann zu chaotischen Ergebnissen führen: Das in *dprint.pl* ausgelagerte *DPRINT()* führt zwei *print*-Anweisungen aus und zeigt zunächst die aktuelle Uhrzeit sowie anschließend eine Meldung an. Ruft der aktuelle Thread diese Funktion auf, muß aber kurz nach dem ersten *print* die Kontrolle einem anderen überlassen, weil seine Zeitscheibe abgelaufen ist, verwendet dieser unter Umständen ebenfalls *DPRINT()* und kleistert seine Uhrzeit in die Standardausgabe, bevor der erste Thread seine Nachricht abschließen konnte.

Zur Regulierung bietet das Thread-Paket eine Reihe von Mechanismen an, die die laufenden Threads dazu zwingen, bestimmte 'kritische' Sektionen nur im Alleingang zu durchwandern und an deren Eingang eine geordnete Warteschlange zu bilden.

Trägt eine Subroutine das Attribut *locked*, wofür

```
use attrs qw(locked);
```

sorgt (s. *dprint.pl*), schleust das Betriebssystem die Threads einzeln durch die Funktion und läßt Drängler vor ihr warten, bis sie der aktuelle Benutzer verlassen hat. Diese Serialisierung bremst parallele Threads naturgemäß drastisch ab, und so eignet sich das Locking von Subroutinen nur für kurze Codestücke, die absolut keine parallele Ausführung vertragen.

Methoden von Objekten blockieren

Wenn parallel laufende Threads auf Methoden von Objekten zugreifen, besteht keine Gefahr, solange sie nicht dasselbe Objekt benutzen wollen. Diesen Sonderfall deckt die Konstruktion

```
use attrs qw(locked method);
```

ab: In Objektmethoden blockiert sie Threads, die dasselbe Objekt bearbeiten, läßt aber parallele Bearbeitung unterschiedlicher Objekte zu.

Zur Synchronisierung paralleler Zugriffe auf globale Variablen (oder auch Instanzvariablen von Objekten) bietet Perl 5.005 den *lock*-Befehl, der eine angegebene Variable (alle Typen, einschließlich Referenzen auf Subroutinen) mit Beschlag belegt. Er läßt nur einen Thread gewähren; andere, die ebenfalls einen Lock setzen wollen, blockieren, bis der erste Thread seinen freigibt. Mangels unlock-Befehls erfolgt die Freigabe automatisch am Ende des innersten Code-Blocks, in dem der Lock steht.

Listing *lock.pl* zeigt fünf Threads, die quasi gleichzeitig jeweils *function()* anspringen und dort zwei Elemente an eine Liste anhängen, für Thread 1 haben diese die Werte 1a und 1b, für Thread 2 die Werte 2a und 2b und so weiter. Da sich im Normalbetrieb Thread-Wechsel nur zufällig einschleichen, provoziert *lock.pl* einen solchen genau zwischen den beiden *push*-Anweisungen mit *yield()*. Er gewährt 'Vorfahrt', veranlaßt also den aktuellen Thread, kurzfristig die Kontrolle abzugeben, um andere Threads dranzulassen. Ohne den Lock in Zeile 23 lautete die Ausgabe

```
LIST = 1a 2a 3a 4a 5a 1b 2b 3b 4b 5b
```

während bei gesetztem Lock folgendes passiert:

```
LIST = 1a 1b 2a 2b 3a 3b 4a 4b 5a 5b
```

Trotz provoziertem Thread-Wechsel in Zeile 25 bleiben die anderen Threads am *lock*-Befehl hängen, solange ein Thread den Lock hält und den Block noch nicht verlassen hat.

lock.pl wartet, bis alle erzeugten Threads zurückgekehrt sind, indem es mittels *Thread->list()* eine Liste von Objektreferenzen aller laufenden Threads holt und jeweils deren *join*-Methode aufruft – nur den Main-Thread spart sie aus. Diesen identifiziert sie mit der Methode *tid()*, die die ID eines Thread zutage fördert – und der Main-Thread führt immer die Nummer 0.

Andererseits entstehen bestimmte Gefahren erst durch Locks: Wartet Thread 1 auf die Variable A, die Thread 2 in Beschlag hat, und wartet andererseits Thread 2 auf die Variable B, die Thread 1 gesperrt hat, ist eine klassische Deadlock-Situation entstanden, und das Programm hängt für immer. Um diesen Fall zu umgehen, ist einfach darauf zu achten, daß beide Threads die Variablen in der gleichen Reihenfolge sperren.

Außerdem muß man bedenken, daß Perl Container-Variablen nicht vollständig sperrt: So schließt ein *lock(@array)* ein nachfolgendes *lock($array[3])* nicht

Installation

Threads kommen mit perl 5.005 – keine frühere Version bietet sie an. Die unter http://www.perl.com/CPAN/src/5.0/perl5.005_01.tar.gz erhältliche aktuelle Version ist für Thread-Betrieb mit

```
./Configure -Dusethreads
```

zu konfigurieren, die Frage 'Build a threading Perl?' muß man mit 'y' beantworten.

Getestet wurden die Beispiele auf Linux 2.0.30 (mit linuxthreads-0.6-4.i386.rpm und linuxthreads-devel-0.6-4.i386.rpm aus der Red-Hat-Distribution), Sun Solaris 2.5 und Irix 6.2. Laut Dokumentation unterstützt Perl unter anderem auch Digital Unix 4.x, OS/2, VMS, Next4.1, Windows 95 und NT. Für letztere ist leider (noch) keine Binär-Distribution verfügbar, so daß man erst mittels eines C-Compilers (Visual C++ 4.0 oder Borland C++ 5.02) die Perl-Source-Distribution übersetzen muß.

iX-TRACT

- Perl 5.005 bietet neben einigen anderen Erweiterungen erstmals Threads an.
- Zur Erzeugung von Threads dient wie bei anderen Perl-Objekten *new*, andere Methoden verwalten und synchronisieren Variablenzugriffe.
- Neben Locks auf Variablen und Subroutinen können Threads auch Semaphore zum kontrollierten Zugriff auf gemeinsame Daten nutzen.
- Zur Zeit gilt die Implementierung als Beta-Software, denn die Schnittstellen können sich ändern. Der größte Teil der Perl-Module in den CPAN-Archiven ist noch auf Thread-Sicherheit zu prüfen.

CHAOS.PL

```
#!/usr/local/bin/perl -w

use Thread;
require 'dprint.pl';

$thread1 = Thread->new(\&function, 1);
$thread2 = Thread->new(\&function, 2);
$thread3 = Thread->new(sub { function(3); });

$thread1->join();
$thread2->join();
$thread3->join();

sub function {
    my $param = shift;

    DPRINT("Thread $param started");
    sleep(3);
    DPRINT("Thread $param ends");
}
```

DPRINT.PL

```
#!/usr/local/bin/perl -w

sub DPRINT {
  # Nachricht mit Zeitangabe ausgeben
    use attrs qw(locked);

    printf "%02d:%02d> ",      # Uhrzeit
          (localtime(time))[2,1,0];

    print "@_\n";              # Nachricht
}

1;
```

LOCK.PL

```
 1 #!/usr/local/bin/perl -w
 2
 3 use Thread qw/yield/;
 4
 5 @LIST = ();
 6
 7 foreach $i (1..5) {
 8     my $thread = Thread->new(\&function, $i);
 9 }
10
11 foreach $thread (Thread->list()) {
12     next if $thread->tid() == 0;
13     $thread->join();
14 }
15
16 print "LIST = @LIST\n";
17
18 ######################
19 sub function {
20 ######################
21     my $param = shift;
22
23     { lock @LIST;
24       push(@LIST, "${param}a");
25       yield();
26       sleep(1);
27       push(@LIST, "${param}b");
28     }
29 }
```

ERROR.PL

```
#!/usr/local/bin/perl -w

use Thread;

$thread = Thread->new(\&function);

sleep(5);    # Zeit vergeht ...
 # Auf Thread warten und Fehler abfangen
eval {$thread->join()};

if($@) {     # Fehler aufgetreten?
    print "Fehler: $@\n";
}

sub function {
    print "In function ...\n";
    die "Böser Fehler!";
}
```

aus – der Lock bezieht sich immer nur auf die angegebene Container-Variable, nicht auf ihre Elemente.

Semaphore helfen beim Synchronisieren

Auch die guten alten Dijkstra-Semaphoren stehen zur Thread-Synchronisierung zur Verfügung. Mit

```
use Thread::Semaphore;
$sem = Thread::Semaphore->new();
```

entsteht ein neuer Semaphor, ein Zähler, der normalerweise den Wert 1 führt. Möchte nun ein Thread exklusiven Zugriff auf ein Stück Code, darf er den Semaphor um 1 herunterzählen – aber nur, falls sein Wert noch größer Null ist, andernfalls blockiert der Thread. Schiebt also ein Thread den Riegel mit

```
$sem->down();
```

vor, ist der Semaphor 0, was den Code für den nächsten Thread blockiert, bis der erste den Semaphor mit

```
$sem->up();
```

freigegeben hat. Das Semaphor-Objekt garantiert, daß die Operationen atomar ablaufen, also zwischen Test und Setzen nichts Unvorhergesehenes passiert.

Teilen sich Threads Aufgaben, schieben sie sich gegenseitig Teilergebnisse zu. Dazu können sie entweder mit Locks gesicherte globale Variablen benutzen oder Warteschlangen vom Typ *Thread::Queue*. Eine neue Warteschlange entsteht mit

PC.PL

```
#!/usr/local/bin/perl

use Thread;
use Thread::Queue;

$queue = Thread::Queue->new();

$producer = Thread->new(\&produce, $queue);
$consumer = Thread->new(\&consume, $queue);

$producer->join();
$consumer->join();

sub produce {
    my ($queue) = @_;
    foreach $i (1..10) {
        $queue->enqueue($i);
        print "Produced $i\n";
        sleep(rand(3));
    }
}

sub consume {
    my ($queue) = @_;
    my $val;
    do {
        $val = $queue->dequeue();
        print "Consumed $val\n";
    } while($val != 10);
}
```

```
use Thread::Queue;
$queue = Thread::Queue->new();
```

und Daten gelangen mit

```
$queue->enqueue($item);
```

hinein. Nach der First-in-last-out-Regel holt

```
$item = $queue->dequeue();
```

die abgelegten Daten wieder hervor. Ist die Warteschlange leer, blockiert die *dequeue*-Methode den aktuellen Thread so lange, bis wieder Daten vorliegen.

BOSS.PL

```
#!/usr/local/bin/perl -w

use Thread;
use Thread::Queue;
use Thread::Semaphore;

$| = 1;                          # Ausgabe entpuffern
$max_parallel_jobs = 3;          # Parallele Jobs
$total_jobs        = 5;
                                 # Neue Ergebnis-Queue
$results = Thread::Queue->new();

$sem = Thread::Semaphore->new($max_parallel_jobs);

foreach $job (1..$total_jobs) {   # Jobs erzeugen
    my $sleep_time = int(rand(10)+1);
                                 # Task in TODO stecken
    push(@todo, [\&task, $job, $sleep_time]);
}

# Weitermachen, solange noch Jobs in TODO stehen

while($#todo >= 0) {
  my($funcref, $job, $sleep_time) = @{shift(@todo)};

  $sem->down();          # Nur N Tasks gleichzeitig

                         # Neuen Thread erzeugen
  my $thread = Thread->new($funcref, $sleep_time,
                           $results, $sem);
  $thread->detach();     # Abkoppeln

  # Formatierter Zeitstempel für Ausgabe
  $time = sprintf "%02d:%02d",
                  (localtime())[2,1,0];

  # Anzahl laufender Threads ermitteln - Main
  # zaehlt mit, also $#threads, nicht $#threads+1
  @threads = Thread->list();
  print "$time> task($job) started (RUNNING[",
        $#threads, "] TODO[", $#todo+1, "])\n";
}

# Ergebnis ausgeben:
for $job (1..$total_jobs) {
                         # Ergebnis aus Queue holen
                         # (notfalls warten)
    $result = $results->dequeue();
    my ($thread_id, $sleep_secs ) = @$result;
                    # ID und Sekunden ausgeben
    print "Thread $thread_id slept for " .
          "$sleep_secs seconds\n";
}

sub task {
    my ($sleep_secs,     # Parameter
        $result,         # Ergebnis-Queue
        $sem             # Semaphor
       ) = @_;

    # Kostspiele Aufgabe ausführen
    sleep($sleep_secs);

    # Ergebnis ablegen
    $result->enqueue([Thread->self->tid(),
                      $sleep_secs]);

    # Semaphor um Eins erhöhen, eine Task mehr
    # zulassen, da der Thread gleich beendet wird.
    $sem->up();
}
```

Den praktischen Einsatz zeigt Listing *pc.pl*: Ein Ergebnisse produzierender Thread *$producer* legt Daten in einer Queue ab, aus der der parallel laufende Thread *$consumer* sie, falls sie vorliegen, wieder hervorholt und ausgibt. Damit letzterer nicht blockiert, falls der Erzeuger keine Daten mehr produziert, müssen sich beide auf ein Ende-Kriterium einigen: Entweder liegt dann in der Warteschlange ein spezieller Wert (beispielsweise *undef*), oder *$consumer* wartet auf eine bestimmte Datenmenge, wie in diesem Fall.

Threads sterben mit Verzögerung

Ein Thread, der auf eine *die*-Anweisung läuft, gibt nicht sofort eine Fehlermeldung aus. Wie *error.pl* zeigt, kommt der Fehler erst zum Vorschein, falls ein anderer Thread mit

```
$thread->join();
```

auf den Thread wartet. Mit einem *eval* umschlossen, läßt sich der Programmabbruch vermeiden und die Fehlermeldung diskret behandeln.

Listing *boss.pl* enthält einen Work-Queue-Manager, der eine Reihe von Aufgaben an Threads delegiert, wobei er darauf achtet, nicht mehr als eine einstellbare Anzahl (*$max_parallel_jobs*) von Threads gleichzeitig zu starten.

In der Liste *@todo* stecken die Parameter der einzelnen Jobs, eine laufende Nummer, eine Referenz auf die auszuführende Funktion *task* und die aus einem Zufallswert ermittelte Anzahl der Sekunden, die diese als Parameter erwartet, um die entsprechende Zeit zu verschlafen und ein Ergebnis in der Schlange *$results* abzulegen.

Den Semaphor *$sem* initialisiert *boss.pl* anfangs nicht wie üblich mit 1, sondern mit der maximal zulässigen Anzahl von Threads. Vor dem Start eines neuen Thread führt das Script die Methode *$sem->down()* aus und stellt dadurch sicher, daß sich maximal *$max_parallel_jobs* Threads im Prozeß bewegen – fällt der Wert des Semaphors auf Null, blockiert *down()*, bis wieder Platz verfügbar ist.

Wenn die vorgeschriebene Schlafphase verstrichen ist, packt *task()* in die Ergebnis-Queue eine Referenz auf ein Array, das die Thread-ID und die eingestellte Schlafzeit enthält. Am Ende von *task()*, also kurz vor dem Ableben des Thread, sorgt der Aufruf von *$sem->up* dafür, daß der Manager-Thread wieder neue Threads ins Rennen schickt. Lautet die Ausgabe von *boss.pl* etwa

```
12:00:00> task(1) started (RUNNING[1] TODO[4])
12:00:00> task(2) started (RUNNING[2] TODO[3])
12:00:00> task(3) started (RUNNING[3] TODO[2])
12:00:04> task(4) started (RUNNING[2] TODO[1])
12:00:04> task(5) started (RUNNING[3] TODO[0])

Thread 2 slept for 4 seconds
Thread 3 slept for 4 seconds
Thread 1 slept for 7 seconds
Thread 5 slept for 5 seconds
Thread 4 slept for 10 seconds
```

startete der Manager zunächst drei Threads (Zeitpunkt 12:00:00) und blockierte dann, denn das war die maximal zulässige Zahl. Nach vier Sekunden waren Thread 2 und 3 beendet, und der Manager legte Thread 4 und 5 nach. Damit ist die *while*-Schleife erledigt. Das folgende *for* gibt die Ergebnisse in der Reihenfolge aus, in der sie in der *$results*-Queue eingetroffen sind und wartet wegen der *dequeue*-Methode, bis alle Threads beendet sind.

Noch gilt die Thread-Implementierung in Perl 5.005 als Beta-Version. Nicht alles funktioniert reibungslos, und besonders die vielen CPAN-Module müssen noch Thread-sicher gemacht werden. Auch wenn sich die Schnittstellen zum Thread-Interface noch weiterentwickeln, sollte man sich langsam an die neue Syntax gewöhnen. (ck)

MICHAEL SCHILLI

arbeitet als Web-Engineer für America Online Inc., San Mateo. Er ist Autor des 1998 bei Addison-Wesley erschienenen Buches 'GoTo Perl 5'.

Literatur

[1] Dan Sugalski; Threads; The Perl Journal, ISSUE #10, S. 53

[2] Andrew D. Birrell; An Introduction to Programming with Threads; Digital Equipment Coorporation, 1989; http://www.research.digital.com/SRC/staff/birrell/bib.html

[3] David R. Butenhof; Programming with POSIX Threads; Addison-Wesley 1997; ISBN 0-201-63392-2

[3] Bil Lewis, Daniel J. Berg; Multithreaded Programming with Pthreads; Sun Microsystems 1998; ISBN 0-13-680729-1

[3] Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrell; Pthreads Programming; O'Reilly & Associates, Inc. 1996; ISBN 1-565-92115-1