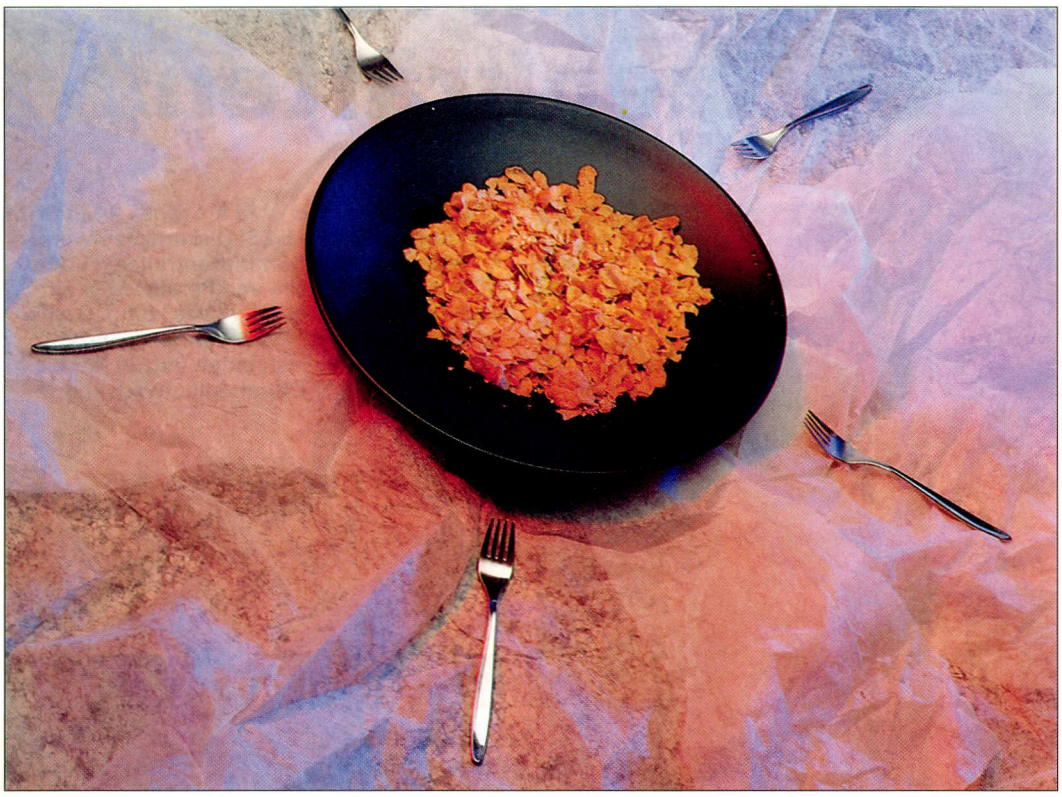


Dinner for five

Ein Spaziergang durch zwanzig Jahre Multitasking

Hartwig Thomas

1971 lancierte Edsger Wybe Dijkstra die Anekdote der 'dinierenden Philosophen'. Sie charakterisiert wichtige Grundeigenschaften der Programmierung von Betriebssystemen, Echtzeitproblemen und der Verwaltung der Kommunikation gekoppelter Prozesse. Mit der zunehmenden Bedeutung von Netzwerken, Mehrprozessorsystemen und Multimedia-Anwendungen wird die Frage, wie ein hungriger Philosoph zu seinen Spaghetti kommt, dringender denn je.

Ja, es geht hier wirklich um Multitasking- und Netzwerkprobleme. Nicht, daß ein geplagter Netzwerkuser ob der Unbill eines LANs philosophisch veranlagt sein müßte. Aber Ressourcen (sprich Gabeln) wollen einigermaßen gerecht auf verschiedene Benutzer (Philosophen) verteilt sein. Und die Frage, ob eine übergeordnete Kontrollinstanz nötig ist oder ob die konkurrierenden Prozesse sich selbst einigen sollten, liest sich in dieser Form einfach viel netter:

'Das Leben eines Philosophen besteht aus den sich abwechselnden Tätigkeiten Denken und Essen. ... Fünf Philosophen, von 0 bis 4 durchnumeriert, leben in einem Haus, wo der Tisch für sie gedeckt ist. Jeder Philosoph hat seinen Platz am Tisch.

Ihr einziges Problem – abgesehen von solchen philosophischer Natur – besteht darin, daß die aufgetragene Mahlzeit eine sehr schwierige Art von Spaghetti ist, zu deren Verzehr man zwei Gabeln benötigt. Neben jedem Teller liegen zwei Gabeln, so daß grundsätzlich keine Schwierigkeiten auftauchen sollten. Es hat jedoch zur Folge, daß zwei Nachbarn nicht zur selben Zeit essen können.' (übersetzt aus [1])

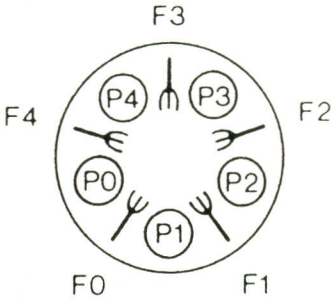
Seit Dijkstra diese kleine Anekdote vor fast zwanzig Jahren vorgestellt hat, sind viele Lösungen für das Problem vorgeschlagen worden. Die meisten davon sind in der einen oder anderen Pseudo-Programmiersprache gegeben. Solche formalen Lösungen des Problems dienen bei genauer Definition der Semantik des Formalismus sicher der höheren Präzision, verstellen aber oft den Blick aufs Wesentliche. Im folgenden sind alle Lösungen und Diskussionen des Problems in der Sprache der Anekdote wiedergegeben. Die formale Programmform figuriert jeweils neben den entsprechenden Tisch als Illustration.

Alles geregelt

Dijkstra hat an anderer Stelle ausgeführt, daß die Philosophen unabhängig voneinander vom Zustand des Denkens in den des Essenwollens übergehen. Genaugenommen besteht ihr Leben also aus drei Zuständen: Denken, Hunger, Essen, die immer in dieser Reihenfolge durchlaufen werden. Die Lösung des Problems sollte in einer allgemeinen Verhaltensanweisung, einem 'Programm', für jeden Philosophen vorliegen. Die Einführung eines 'Oberkellners', der Ordnung schafft, oder die unterschiedliche Behandlung der Philosophen kommt aus zum Teil unten näher erläuterten Gründen nicht als 'echte' Lösung in Betracht.

Die Unabhängigkeit der Zustandswechsel der Philosophen hat zur Folge, daß äußerst vielfältige, komplexe zeitliche Abfolgen denkbar sind. Eine erste Versuchslösung, die C. A. R. Hoare formal durchgeführt hat [2]:

Wenn du hungrig bist, setz dich an deinen Platz. Warte,



Auf den ersten Blick sieht das Problem der dinierenden Philosophen nach reiner Gedankenspielerei aus. Dabei stehen die insgesamt nur fünf Gabeln für verknappte Ressourcen in einem großen System. Dann wird auch deutlich, warum die Philosophen sich nicht gütlich einigen und warum diese Anekdote in der Informatik weithin bekannt ist.

```
PHIL = *[...during ith lifetime... →
        THINK;
        room!enter( );
        fork(i)!pickup( ); fork((i+1) mod 5)!pickup( );
        EAT;
        fork(i)!putdown( ); fork((i+1) mod 5)!putdown( );
        room!exit( )
        ]

FORK = *[phil(i)?pickup( ) → phil(i)?putdown( )
       □(phil((i-1) mod 5)?pickup( ) → phil((i-1) mod 5)?putdown( )
       ]

ROOM = occupancy:integer; occupancy:=0;
       *[(i:0..4)phil(i)?enter( ) → occupancy := occupancy + 1
       □[(i:0..4)phil(i)?exit( ) → occupancy := occupancy - 1
       ]


All these components operate in parallel:

[room::ROOM||fork(i:0..4)::FORK||(i:0..4)::PHIL].
```

Hoare formulierte seine Lösung zum Philosophen-Problem in CSP, einem Algol-60-Abkömmling. Sie ist einfach, aber unvollständig.

bis links eine Gabel frei wird, und ergreife sie. Warte, bis rechts eine Gabel frei wird, und fang an zu essen. Nach dem Essen gib beide Gabeln frei.

Diese schöne, weil klare Lösung hat dummerweise den Nachteil, daß sie zum 'Deadlock' führen kann: es ist ein zeitlicher Ablauf vorstellbar, wo alle Philosophen mehr oder weniger zur selben Zeit hungrig sind, gemäß der Anweisung die linke Gabel ergreifen und ewig auf die rechte warten. Der Gesamtdurchsatz des Systems sinkt dann für immer auf null.

Viele Köche ...

Auf dem Titelbild ihres Kompendiums über die Implementation von Betriebssystemen haben Holt, Graham, Lazowska und Scott [3] zwar E. W. Dijkstra den Ehrenplatz zugewiesen – in einigermaßen volkstümlicher Darstellung –, die Lösung, die sie vorschlagen, läßt aber zu wünschen übrig:

Wenn du hungrig bist, setz dich an deinen Platz. Warte, bis links und rechts die Gabel frei ist. Ergreife beide und fang an zu essen. Nach dem Essen gib beide Gabeln frei.

Man kann leicht zeigen, daß mit diesem 'Programm' zwar der Deadlock, wo niemand mehr zum Essen kommt, ausgeschlossen ist. Der individuelle Hungertod des einzelnen droht aber immer noch: wenn sich die beiden Nachbarn rechts und links leicht überlappend beim Essen abwechseln, verhungert man glatt vor der vollen Spaghettischüssel!

Eisenbahnsignale

Dijkstra hatte schon drei Jahre vor der Erfindung der Philosophen-Parabel die Grundelemente für ihre Lösung vorgeschlagen [4]. Er diagnostizierte eine Komplexitätsexplosion beim Programmieren von Betriebssystemen, wo auf äußere Signale reagiert werden muß, deren zeitlicher Ablauf nicht vorhersagbar ist.

In den Jugendjahren der programmierbaren Computer gab es auf einem Rechner genau ein Programm. Dieses erfüllte die Aufgaben der Maschine. Wenn sich diese Aufgabe oder die Zusammensetzung der Peripheriegeräte änderte, mußte ein neues Programm her. Je mehr Ressourcen den immer leistungsfähigeren Computern zur Verfügung standen, desto größer und komplexer wurde das Programm. Da sich die Aufgaben der Computer häufiger änderten als die Hardware-Konfiguration, gliederte man gewisse Programmteile aus, die man Anwenderprogramme nannte.

Das Hauptprogramm, nun Betriebssystem oder Operating System (OS) genannt, versah man deshalb mit einem flexiblen Mechanismus, der solche Anwenderteile laden und laufen lassen konnte. Die Aufgabe des Betriebssystems bestand darin, zwischen den Peripheriegeräten und den Anwenderprogrammen zu vermitteln. Die Konkurrenz verschiedener äußerer Signale um die 'Aufmerksamkeit' des Rechners wie auch die Konkurrenz verschiedener Anwenderprogramme um beschränkt verfügbare Geräte führte zu immer komplexer werdenden Techniken der Ressourcenzuteilung durch das Betriebssystem.

```
SPAGHETTI:PROCEDURE OPTIONS(CONCURRENT);
   DECLARE(TRUE,FALSE)BIT;
   FORKS:MONITOR;
         DECLARE(#_FORKS(0:4))FIXED,
            (READY(0:4))CONDITION,
            (J,#_PHILOS)FIXED;
         ... (define LEFT and RIGHT) ...
         DO;
            #_PHILOS=5;
            DO J=0 TO #_PHILOS-1;
               #_FORKS(J)=2;
               END;
            TRUE='1'B;
            FALSE='0'B;
            END;

      PICKUP:ENTRY(ME);
         DECLARE(ME)FIXED;
         IF #_FORKS(ME)¬=2 THEN
            WAIT(READY(ME));
         #_FORKS(RIGHT(ME))=#_FORKS(RIGHT(ME))-1;
         #_FORKS(LEFT(ME))=#_FORKS(LEFT(ME))-1;
         END;

      PUTDOWN:ENTRY(ME);
         DECLARE(ME)FIXED;
         #_FORKS(RIGHT(ME))=#_FORKS(RIGHT(ME))+1;
         #_FORKS(LEFT(ME))=#_FORKS(LEFT(ME))+1;
         IF #_FORKS(RIGHT(ME))=2 THEN
            SIGNAL(READY(RIGHT(ME)));
         IF #_FORKS(LEFT(ME))=2 THEN
            SIGNAL(READY(LEFT(ME)));
         END;
      END /* OF FORKS MONITOR */;

   COMMON_PHILOSOPHER:PROCEDURE(I);
      DECLARE(I)FIXEF;
      DO WHILE(TRUE);
         CALL PICKUP(I);
         Busy eating;
         CALL PUTDOWN(I);
         Busy thinking;
         END;
      END;

   PHILOSOPHER_0:PROCESS;
      CALL COMMON_PHILOSOPHER(0);
      END;
   PHILOSOPHER_1:PROCESS;
      CALL COMMON_PHILOSOPHER(1);
      END;
   PHILOSOPHER_2:PROCESS;
      CALL COMMON_PHILOSOPHER(2);
      END;
   PHILOSOPHER_3:PROCESS;
      CALL COMMON_PHILOSOPHER(3);
      END;
   PHILOSOPHER_4:PROCESS;
      CALL COMMON_PHILOSOPHER(4);
      END;
END /* OF SPAGHETTI */;
```

Dieses Programm in CSP/k ist zwar deadlock-frei, aber der einzelne Philosoph kann verhungern.

Um dieser Komplexitätsexplosion mit den ihr zugehörigen Programmfehlern und Abstürzen entgegenzuwirken, seien die Prozesse und die I/O-Behandlung möglichst sauber voneinander zu isolieren, postulierte Dijkstra. Als einzigen kontrollierten/kontrollierenden Kommunikationsmechanismus zwischen Prozessen und Geräten schlug er eine Anleihe bei der holländischen Eisenbahn vor und führte die sogenannten 'Semaphore' ein (griechisch 'Zeichenträger'). Für Betriebssysteme, welche auf diese Weise eine klare Trennung mehrerer Prozesse und Geräte sauber handhaben, hat sich der Name Multitasking-Systeme eingebürgert.

Ein Semaphor ist ein Signal, das von den verschiedenen Prozessen und nicht von einer Zentralinstanz betätigt wird. Die Prozesse kooperieren miteinander und warten, wenn das Si-

```
In the universe we assume declared

1) the semaphore mutex, initially = 1
2) the integer array C[0:4], with initially all elements = 0
3) the semaphore array prisem [0:4] with initially all elements = 0
4) procedure test(integer value K);
   if C[(K-1) mod 5]&2 and C[K]=1 and C[(K+1) mod 5]&2 do
              begin C[K]:=2; V(prisem[K]) end;

In this universe the life of philosopher w can be coded

cycle begin think;
            P(mutex);
              C[w]:=1; test(w);
            V(mutex);
            P(prisem([w]); eat;
            P(mutex);
              C[w]:=0; test [(w+1) mod 5]; test [(w-1) mod 5];
            V(mutex)
      end.
```

Auch Dijkstra, der Erfinder des Problems, sichert den Denkern nicht das Überleben. Dafür arbeitet sein Algorithmus recht effizient.

gnal 'Warten' anzeigt. Dijkstra nannte die zwei Grundoperationen für Semaphoren P (Warten auf Freigabe der Ressource, WAIT) und V (Freigeben der Ressource, SIGNAL). Diese Grundoperationen werden als unteilbar angenommen. Das heißt, daß nur ein Prozeß auf einmal dasselbe Signal stellen kann. Ein Betriebssystem, das mit diesen beiden Grundoperationen ausgerüstet ist, kann unerwünschte zeitliche Abläufe verhindern und damit das Verhalten unabhängiger Instanzen transparent gestalten.

Ausschlußverfahren

Die klassische Grundvoraussetzung für sauberes Programmieren gleichzeitiger Prozesse ist die Möglichkeit des gegenseitigen Ausschlusses während gewisser kritischer Operationen. Dieser gegenseitige Ausschluß ('mutual exclusion') wird mit den Semaphoren-Operationen leicht bewältigt. Will ein Prozeß eine kritische Region durchlaufen, stellt er zunächst ein Semaphor *Mutex*, sofern es auf *Frei* steht, auf *Warten* um. Weitere Prozesse, die ihrerseits in die kritische Region einzutreten wünschen, stehen vor dem Signal Schlange und warten, bis es der erste Prozeß am Ende seiner kritischen Operation wieder auf *Frei* stellt. Der schnellste der wartenden Prozesse beginnt dann seine kritische Region, indem er das Signal auf *Warten* stellt ...

Es ist klar, daß kritische Regionen möglichst kurz sein sollten, um parallele Verarbeitung mit möglichst hohem Durchsatz zu ermöglichen. Ebenfalls klar ist, daß Kooperation der beteiligten Prozesse nötig ist, um eine hohe Gesamtleistung des Systems zu ermöglichen.

Die saubere Ableitung der klaren Organisation von Betriebssystemen, wie sie Dijkstra und seine Kollegen bekannt machten, führte zu einer allmählichen Modernisierung der Betriebssystemarchitektur der Großcomputer in den siebziger Jahren. Kleine neue Entwicklungen mit Echtzeitanforderungen berücksichtigten die neuen Erkenntnisse etwas früher als die weitverbreiteten Maschinen mit Verpflichtung zur Rückwärtskompatibilität. Fälschlicherweise kam damit die Technik des Multitasking in den Ruf, nur für Echtzeitsysteme (Real Time Operating System, RTOS) nützlich zu sein. Das Vorurteil, daß Multitasking zu kompliziert und für Einplatzsysteme überflüssig sei, führte bei der Entwicklung des Betriebssystems für die neuen PCs zu den vorsintflutlichen Systemarchitekturen CP/M und MSDOS.

Die Grundeigenschaften der Semaphoren führen dazu, daß sich die verschiedenen Prozesse weitgehend unabhängig von ihrer zeitlichen Durchmischung betrachten lassen und daß man die Korrektheit der einzelnen Prozesse wieder einfach beweisen kann, ohne das Verhalten aller anderen Prozesse mit berücksichtigen zu müssen.

Eine korrekte, aber ungünstige Lösung der Ressourcenzuteilung beim Philosophen-Problem mit Semaphoren baut auf dem Prinzip des gegenseitigen Ausschlusses auf. Zu diesem Zweck legen wir einen zweifarbigen Stein in die Mitte des Tisches. Dieser ist auf der einen Seite grün und auf der anderen rot. (An Stelle des weniger häufigen Eisenbahnsignals verwenden wir das Bild der Verkehrsampel.) Zu Beginn des Gelages wird üblicherweise angenommen, daß sich alle Philosophen im Zustand des Denkens befinden. In diesem Moment zeige der Stein seine grüne Seite. Das Programm für den einzelnen Philosophen könnte jetzt so lauten:

Wenn du hungrig bist, setz dich an deinen Platz. Warte, bis der Stein in der Mitte grün anzeigt, drehe ihn dann auf rot. Nimm beide Gabeln zur Rechten und zur Linken und fang an zu essen. Nach dem Essen dreh den Stein auf grün.

Volltreffer

Mit diesem Programm ißt höchstens einer auf einmal. Sofern die Auswahl des nächsten zu aktivierenden Prozesses beim Freiwerden des Signals fair gehandhabt wird, sind sowohl Deadlock als auch individuelles Verhungern ausgeschlossen. Der einzige Ärger besteht darin, daß der gesamte Vorgang zu stark sequentialisiert abläuft. Bei insgesamt fünf Gabeln könnten grundsätzlich immer zwei auf einmal essen, ohne sich gegenseitig zu stören. Mit der angegebenen Radikallösung halbiert sich aber der maximale Durchsatz des Systems. Drastischer wird das noch, wenn man etwa vom 5-Philosophen-Problem zum 500-Philosophen-Problem übergeht.

Dijkstra stellt fest, daß ein Philosoph genau dann zum Essen übergehen muß, wenn er hungrig und keiner seiner Nachbarn am Essen ist. Ein solcher Zustand tritt durch zwei mögliche Ereignisse ein: ein Philosoph wird hungrig oder andere geben Gabeln frei, nachdem sie gegessen haben. Beim Übergang zum Hunger muß ein Denker also testen, ob beide Gabeln zur Verfügung stehen und er daher gleich mit dem Essen beginnen sollte. Wenn er gegessen hat, muß er für die Nachbarn prüfen, ob diese jetzt essen sollten. Damit das Testen zweier Gabeln und des Zustands des Philosophen nicht durch fremde Zustandsübergänge verwirrt wird, hat dieses unter dem Schutz des gegenseitigen Ausschlusses stattzufinden. Die Veranlassung zum Essen findet über ein privates Semaphor statt, das vor jedem Teller liegt und anfänglich auf rot steht.

Wenn du hungrig bist, setz dich an deinen Platz. Zum Testen warte, bis der Stein in der Mitte grün anzeigt und drehe ihn auf rot. Wenn nun beide Gabeln erhältlich sind, nimm sie an dich und drehe den Stein bei deinem Teller auf grün. Drehe den Stein in der Tischmitte wieder auf grün. Warte, bis der Stein bei deinem Teller auf grün steht. Drehe den Stein bei deinem Teller auf rot und iß.

Zum Testen warte wieder, bis der Stein in der Mitte auf grün steht und drehe ihn auf rot. Wenn dein Nachbar zur Linken hungrig ist und beide Gabeln zur Verfügung hat, drehe den Stein bei seinem Teller auf grün. Wenn dein Nachbar zur Rechten hungrig ist und nun beide Gabeln zur Verfügung hat, drehe den Stein bei seinem Teller auf grün. Drehe den Stein in der Tischmitte wieder auf grün und fang an zu denken.

Der gegenseitige Ausschluß, den wir oben kritisiert hatten, findet auch in Dijkstras Lösung statt. Wir nehmen dabei aber an, daß die Semaphoren-Operationen im Vergleich mit der Aktivität Essen 'kurz' sind. Insofern schadet die totale Sequentialisierung des Zugangs zu den privaten Semaphoren dem Durchsatz des Gesamtsystems nicht.

Ärgerlicherweise hat auch Dijkstras Lösung den kleinen Nachteil, daß das individuelle Verhungern nicht ausgeschlossen ist. Genaugenommen handelt es sich nur um eine sorgfältig formulierte Version der Lösung von Holt, Graham, Lazowska und Scott. Es fällt auf, daß alle Autoren die Ausarbeitung einer Lösung, die den individuellen Hungertod vermeidet, dem Leser überlassen – die Sorge des Programmierers des klassischen Betriebssystems ist bestenfalls der Deadlock. Dijkstra

behauptet, daß eine vollständige Lösung die Einführung eines weiteren Zustands 'Sehr Hungrig' nötig machen würde, ansonsten aber nach demselben Schema vorgehen könnte.

Dijkstras Semaphoren sind Grundelemente der meisten modernen Multitasking-Betriebssysteme geworden. Wie Dijkstras Lösung des Philosophen-Problems zeichnen sich diese Systeme durch folgende Eigenschaften aus:

- hohe Unabhängigkeit der Teilnehmer,
- Diskretheit des Betriebssystems (es kümmert sich nur um die Behandlung der globalen Semaphoren und den Entscheid, welcher der auf ein Semaphor wartenden Prozesse aufgeweckt werden soll),
- Gleichberechtigung der Teilnehmer,
- Symmetrie ihrer Programme,
- Kooperationsbereitschaft und
- gute Ausnutzung der Ressourcen.

Zu diesen modernen Multitasking-Betriebssystemen gehört sicherlich Andrew S. Tanenbaums Minix. Er scheint aber in seinem Lehrbuch zum Betriebssystembau [9] die Problemstellung nicht besonders ernst zu nehmen. Er handelt sie auf knapp drei Seiten ab und präsentiert eine ebenso kurze wie unvollständige Lösung.

Dies mag unter anderem an einer Begriffsverwechslung liegen. Den Zustand, wo alle Philosophen im Gleichtakt Gabeln aufnehmen und wieder hinlegen, nennt er *Starvation* (Verhungern). Genaugenommen bezeichnet man das aber als *Threshing* (Dreschen): alle Prozesse arbeiten wie wild, ohne produktiv zu sein. Um 'echte' Starvation hat er sich auch nicht gekümmert.

Tanenbaum arbeitet wie Dijkstra mit einem globalen Semaphor *mutex* und fünf lokalen, die verhindern, daß ein Philosoph eine Gabel nimmt, während einer seiner Nachbarn ißt. So bleibt sein Programm letztendlich eine Übertragung der Dijkstraschen Lösung in eine andere Sprache.

Wie Dijkstra an anderer Stelle [5] schreibt, war es sein Ehrgeiz, aus einer Menge Mücken 'nur' einen Elefanten zu machen. Jeder Programmierer weiß, daß Mücken leicht und elegant herstellbar sind, daß sich aber bei der Zusammensetzung vieler kleiner Teile zu einem großen scheinbar unverhinderbar der Dinosaurier-Effekt einstellt: das Ganze wird mehr und mehr durch sein Eigengewicht am Fortkommen gehindert. Mit der Einführung der Semaphoren und dem Design moderner Betriebssysteme ist das Elefanten-Projekt einen guten Schritt vorwärtsgetrieben worden.

Mücken und Elefanten

Als besonders stark hat sich das Multitasking-Konzept in der Behandlung externer, um die Aufmerksamkeit des Prozessors konkurrierender Signale verschiedenster Quellen erwiesen. Die klassische Multimedia-Applikation 'CD-I' von Philips hat darum das schlanke OS-9 als Echtzeitbetriebssystem adoptiert. Um den Multimedia-Vorsprung des AmigaDOS einzuholen, hat Apple angekündigt, das neue Betriebssystem des Mac werde ein echtes Multitasking-System sein – unter Preisgabe von viel Kompatibilität mit alten Anwendungen. Die Minis mit dem Pionier-Multitasking-System Unix waren besonders bei Wissenschaftlern beliebt, die mit vielen Geräten kommunizieren und auf komplexe Konstellationen reagieren wollten. Die Unhandlichkeit von Netzwerkanwendungen auf dem IBM PC ist der unglücklichen Wahl eines Betriebssystems zuzuschreiben, das schon 1980 konzeptionell vorsintflutlich war. Der OS/2-Dinosaurier erfreut sich denn auch regen Zuspruchs bei (gutbetuchten) Betreibern größerer lokaler Netze, vor allem wegen seiner kommunikationsfreudigeren Multitasking-Eigenschaften.

Manch einer behauptet, wir ständen am Ende der PC-Dekade, und die letzte Dekade dieses Jahrtausends werde die der Netzwerk-Dekade sein. Neben den Multitasking-Systemen mit ihren enggekoppelten Prozessen spielen immer mehr distributive Systeme mit lose gekoppelten Prozessen eine Rolle. Man kann sich da etwa ein Netzwerk von PCs vorstellen, die ab und zu über Wählleitung und Modem kommunizieren.

Von der Vision der vielen Mücken, die alle harmonisch im Einklang summen, sind wir aber in diesem verteilten Szenario noch weit entfernt. Wenn eine Arbeit auf so lose gekoppelte Stationen verteilt werden soll, so stehen keine globalen Semaphoren mehr zur Verfügung. (Wenn der Tisch sehr groß ist, kann man keinen Stein in die Mitte legen, der allen jederzeit erreichbar ist.) Ein distributives System soll trotz Ausfällen einzelner Knoten annähernd das leisten, was ohne diese Knoten noch möglich ist. Wenn in Dijkstras Lösung ein Philosoph in seiner kritischen Region einen Herzinfarkt erleidet, bleibt der zentrale Stein aber auf rot für alle, für immer ...

Stille Post

Diese kritischen Einwände habe zu moderneren Lösungen des Problems geführt, die globale Semaphoren vermeiden und nur Nachbarkommunikation zulassen. Chang [6] identifiziert das Problem der Distributivität: Der Deadlock ist ein globaler Zustand, zu dem alle beitragen, und über den man nur etwas erfährt, wenn man mit allen kommuniziert hat. Die Distributivitätsbedingung verunmöglicht die Verwendung von globalen Variablen.

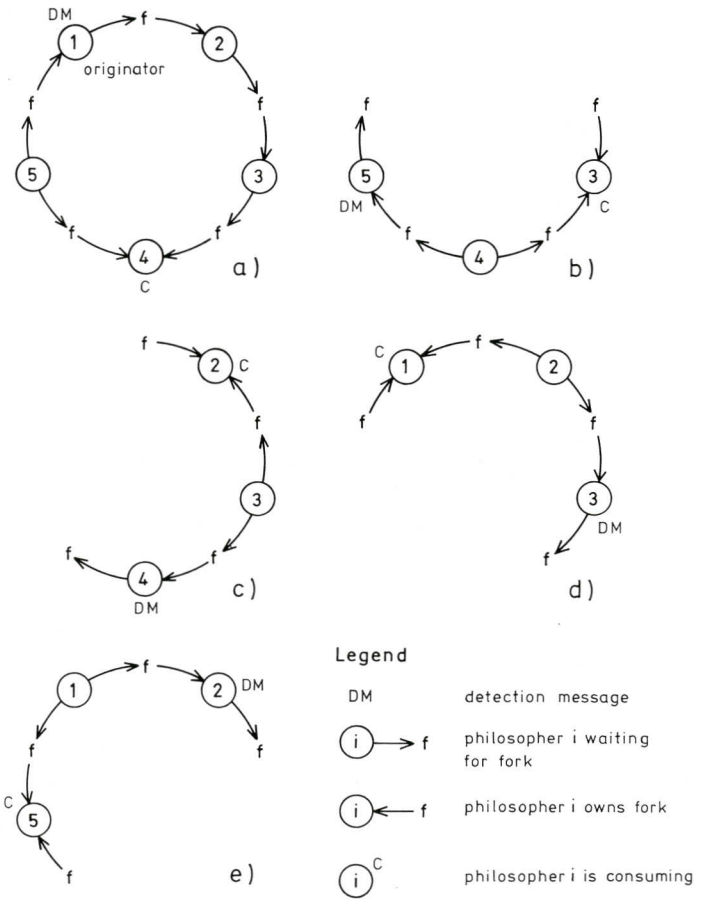
```
 1 #define N        5           /* Anzahl der Philosophen */
 2 #define LEFT     (i-1)%N     /* Nummer seines linken Nachbarn */
 3 #define RIGHT    (i-1)%N     /* Nummer seines rechten Nachbarn */
 4 #define THINKING 0           /* der Philosoph denkt */
 5 #define HUNGRY   1           /* der Ph versucht, Gabeln zu erhalten */
 6 #define EATUNG   2           /* der Philosoph ißt */
 7 typedef int semaphore; /* Semaphoren sind besondere "int" */
 8 int state[N];          /* Vektor für Zustände */
 9 semaphore mutex = 1;   /* zum gegenseitigen Ausschluß */
10 semaphore s[N];        /* ein Semaphor pro Philosoph */
11
12 ph(i)
13 int i;                 /* Philosoph Nummer, 0 bis N-1 */
14 {
15     while (TRUE) {     /* wiederhole */
16         think();       /* Ph denkt */
17         take_forks(i); /* zwei Gabeln erwerben oder blockieren */
18         eat();         /* schmatz-schmatz, Spaghetti */
19         put_forks(i);  /* legt beide Gabeln auf den Tisch zurück */
20     }
21 }
22
23 take_forks(i)
24 int i;                 /* Philosoph Nummer, 0 bis N-1 */
25 {
26     down(mutex);       /* Eintritt in kritischen Abschnitt */
27     state[i] = HUNGRY; /* notiert: Ph "i" ist hungrig */
28     test(i);           /* versucht, zwei Gabeln zu erwerben */
29     up(mutex);
30     down(s[i]);        /* blockiert, wenn Gabeln nicht erhalten */
31 }
32
33 put_forks(i)
34 int i;                   /* Philosoph Nummer, 0 bis N-1 */
35 {
36     down(mutex);         /* Eintritt in kritischen Abschnitt */
37     state[i] = THINKING; /* Ph "i" beendet das Essen */
38     test(LEFT);          /* prüfen: kann linker Nachbar nun essen? */
39     test(RIGHT);         /* prüfen: kann rechter Nachbar nun essen? */
40     up(mutex);           /* verläßt kritischen Abschnitt */
41 }
42
43 test(i)
44 int i;                   /* Philosoph Nummer, 0 bis N-1 */
45 {
46     if (state[i] == HUNGRY && state[LEFT]  != EATING
47                            && state[RIGHT] != EATING)
48     {
49         state[i] = EATING;
50         up(s[i]);
51     }
52 }
```

Auch Andrew S. Tanenbaum, der Macher von Minix, hat sich ein Programm einfallen lassen, natürlich in C. Es ist zwar kurz, verständlich und deadlock-frei, verhindert aber auch nicht die 'Verschwörung' zweier Prozesse, die einen Dritten verhungern lassen.

```
Treating the n-philosophers problem in the context of decentralized
control in a distributed system, we now summarize the behaviour of our
proposed solutions:

1.Any philosopher entering a critical state issues a Detection Message
  carrying his Consumption Sequence Number.
2.Any philosopher in a critical state getting a Detection Message pas-
  ses it on; in a non-critical state, stamps it OK.
3.A philosopher getting back its Detection Message unchanged detects
  Deadlock if its Consumption Sequence Number is still the same as on
  its Detection Message.
4.In the best solution, a philosopher detecting deadlock issues a Re-
  linquish Message, which circulates in a consistent direction. An R-
  philosopher receiving such a message relinquishes its held fork, but
  maintains its current request, and also places a request for its re-
  linquished fork.
5.An alternate solution is for a mutual exclusion mechanism to be in-
  voked, which allows only one philosopher to emit the Relinquish Mes-
  sage. An R-philosopher receiving a Relinquish Message gives up its
  held fork and request, going back to the thinking state. He passes
  on his R-token to his immediate neighbour, in a consistent direc-
  tion.
6.A philosopher who consumes increments his Consumtion Sequence Number
  by one.
The placement of R-tokens is such that no two neighbours should be R-
philosophers. In the best solution, R-philosophers are fixed. In the
alternate solution, they vary from deadlock to deadlock, and should
not exceed (n div 2) in number, to minimize the relinquishing for any
particular philosopher. In both cases, (n div 2), spaced alternately,
will give maximal parallel activity following deadlock detection.
```

Changs textuelle Lösung bedient sich eines distributiven Kommunikationsmodells. Sie ist zwar korrekt, aber sehr aufwendig und fehleranfällig. Er organisiert den Tisch der fünf Philosophen als Ring. Ein Knoten, der eine Gabel besitzt und einen Deadlock feststellt, schickt eine Meldung los, die seine Kollegen auffordert, Gabeln freizugeben. Diese Meldung kann einer Welle des Essens hinterhereilen. Nur wenn sie unverrichteter Dinge zurückkommt, gibt der Philosoph seine eigene Gabel ab, so daß das Spiel von vorne beginnen kann.



Daher führt Chang einen distributiven Mechanismus der globalen Kommunikation ein. Er nimmt an, daß alle Philosophen in einem Ringrufsystem miteinander verbunden sind, in welchem man Meldungen austauschen kann. Er läßt jeden Philosophen bei Hunger die erste Gabel ergreifen, die er kriegen kann. Während er auf die andere Gabel wartet, schickt er eine Deadlock-Meldung durch die Runde, die jeder unverändert weiterreicht, der sich auch im 1-Gabel-Zustand befindet. Jemand, der ißt oder denkt, ändert die Meldung zu 'Kein Deadlock'. Da die Meldung einer Freßwelle hinterherwandern könnte, ohne je verändert worden zu sein, könnte sie zwar fälschlicherweise dem Urheber einen Deadlock melden, aber der hat inzwischen schon gegessen.

Falls nun Deadlock festgestellt wird, kommt in Changs Lösung ein kompliziertes Verfahren in Gang, in dem ein Teil der Philosophen ihre Gabel aufgeben und noch mal von vorne anfangen.

Changs Lösung ist zwar grundsätzlich korrekt, hat aber die Nachteile der hohen Komplexität und der mangelnden Robustheit. Wenn die Meldungskette unterbrochen ist, wenn Deadlock-Meldungen verlorengehen, wenn irgend etwas nicht rund läuft, dann verhungern alle! Eine überraschend primitive *und* robuste, distributive Lösung von Cargill [7] hat keinen dieser Defekte, doch um den Preis einer leichten Asymmetrie: Man nehme an, die Gabeln seien der Reihe nach in der Runde numeriert (von 0 an, wie Dijkstra dies als typischer Programmierer vorschlägt). Neben jedem Philosophen befindet sich nun mindestens eine 'gerade' Gabel. Die robuste, distributive Anweisung an den Philosophen lautet nun:

Wenn du hungrig bist, setz dich an deinen Platz. Warte, bis eine Gabel mit einer geraden Nummer neben dir frei wird und ergreife sie. Warte, bis die andere Gabel neben dir frei ist. Ergreife sie und fang an zu essen. Nach dem Essen laß beide Gabeln los.

Die Lösung muß natürlich ergänzt werden mit einer neuen, für distributive Ansätze typischen Anmeldestruktur. Ein neuer Teilnehmer im (nun als unübersichtlich angenommenen) Netz muß sich beim 'Maître d' registrieren. Dieser teilt Gabeln aus und weist Plätze an und kann dafür sorgen, daß immer abwechselnd eine gerade und eine ungerade Gabel auf dem Tisch liegt.

Kommunikation und Zufall

Die Unsicherheiten und blinden Zufälle eines distributiven Systems kontert man am wirkungsvollsten durch Einführung von Redundanz. Deterministische Redundanz in Form von achtstöckig geschichteten, standardisierten Kommunikationsprotokollen schadet aber oft mehr, als sie nützt. Solche Programme mit ihrer Fehlerbehandlungskomplexität verlieren oft die Transparenz, und ihre Korrektheit kann man nicht mehr garantieren. Immer häufigere Deadlock-Situationen in größeren Banknetzwerken, die man durch immer phantastischere Flickmaßnahmen zu beheben versucht, sind die Folge dieser Maßnahme.

Am Beispiel der Computerviren sieht man, daß stochastische (zufallsgesteuerte), 'blinde' Redundanz – in diesem Fall gegeben durch 'blindwütige' Vermehrung – die Tücken des distributiven Umfelds weit besser bewältigt. Wo ausgeklügelte Kommunikationsprogramme steckenbleiben, dringen Computerviren ein, obwohl ihnen im allgemeinen der Weg nicht gerade geebnet wird. Die Kommunikationsleistung und die Transparenz und Korrektheit stochastischer Programme weisen hier einen Weg zu Lösungen für heutige Kommunikationsprobleme.

Eine stochastische Lösung des distributiven Philosophen-Problems schlagen Francez und Rodeh vor [8]. Diese Autoren implementieren das Problem in CSP und führen neben den Philosophen noch die Tellerprozesse ein.

Wenn du hungrig bist und keine Gabel besitzest, warte, bis einer deiner Nachbarn denkt, und nimm seine Gabel

```
Ph[i]:: var state: (thinking, hungry, waiting);
        state:=hungry;
        *[state=hungry; Table! hunger( )→ state:=waiting
         □state=waiting; Table? twoforks( )→ EAT; state:=thinking;
         □state=thinking; Table! twoforks( )→ THINK; state:=hungry
         ]

EAT and THINK are left as unspecified program sections.

Now we decribe the structure of the table. As said before,
Table::[Plate[0]||...||Plate[n-1]];

Plate[i]::var ishungry, right, left: Boolean;
          ishungry:=false; right:=true; left:=false;
          *[ Ph[i]?hunger( )→ ishungry:=true
            □Ph[i]?twoforks( )→ ishungry:=false
            □ishungry; left; right; Ph[i]!twoforks( )→ skip
            □ishungry; left;~right; Plate[i-1]!fork1( )→ left:=false
            □ishungry;~left; right; Plate[i+1]!fork1( )→ right:=false
            □~ishungry; left; Plate[i-1]!fork0( )→ left:=false
            □~ishungry; right; Plate[i+1]!fork0( )→ right:=false
            □ishungry; left; Plate[i+1]!fork1( )→ right:=true
            □ishungry; right; Plate[i-1]!fork1( )→ left:=true
            □ishungry; Plate[i-1]!fork0( )→ left:=true
            □ishungry; Plate[i+1]!fork0( )→ right:=true
           ]
```

Francez und Rodeh arbeiten auch mit CSP. Ihre Lösung arbeitet mit Philosophen- und Tellerprozessen. Einziger Schönheitsfehler: nur der (als normal angenommene) Zufall verhindert das individuelle Verhungern.

an dich. Wenn du hungrig bist und schon eine Gabel – zum Beispiel die linke – besitzest, so warte, bis entweder dein rechter Nachbar nicht ißt oder dein linker Nachbar hungrig und im Besitze seiner linken Gabel ist. Im ersten Fall nimm auch die rechte Gabel, im zweiten Fall gib ihm deine linke Gabel. Wenn du hungrig bist und zwei Gabeln hast, iß. Wenn du gegessen hast, lege deine Gabeln hin und denke.

Diese Lösung vermeidet globale Semaphoren, verwendet nur nachbarschaftliche Kommunikation, maximiert den Durchsatz und ist deadlock-frei. Wenn ein hungriger Philosoph eine einzige Gabel hält, kommt entweder er selber oder sein Nachbar zum Essen. Leider ist das individuelle Verhungern nach wie vor möglich. Hier rufen die Autoren nun den göttlichen Zufall zu Hilfe und wählen zufällige Zeitspannen für die Beschäftigung mit der linken und rechten Gabel. Die Wahrscheinlichkeit, daß diese systematisch ungünstig sind, soll dabei gegen null gehen. Inwiefern diese Strategie gegen die oben angeführten überlappend essenden Nachbarn hilft, wird aus dem Artikel nicht ganz klar. Die Lösung scheint sich gegenüber der von Holt, Graham, Lazowska und Scott nur im Formalismus zu unterscheiden.

Schöne Aussichten ...

Um so besser sind die Argumente der Autoren für stochastische Lösungen im allgemeinen: Eine stochastische Lösung erlaubt, recht naive Ansätze zu verwenden, da es oft genügt, mit Wahrscheinlichkeit Eins nicht zu verhungern oder den Deadlock auszuschließen. Anhand anderer Beispiele zeigen Francez und Rodeh, wie durch zufälliges Wechseln der Kommunikationspartner unendliche fruchtlose Kreiskommunikation vermieden werden kann.

Auch wenn die vorgeschlagene Lösung des Philosophen-Problems die Anforderungen nicht erfüllt, scheint der eingeschlagene Weg erfolgversprechend. Es genügt für reale Situationen, wenn die Wartezeiten um so unwahrscheinlicher werden, je länger sie dauern. Ein individuelles Verhungern mit asymptotischer Wahrscheinlichkeit null kann man in Kauf nehmen, wenn dafür die Strategie natürlich ist und im Normalfall zu guter Auslastung führt.

In der Dekade der Netzwerke wird das Gebiet der Distributivität immer heißer umkämpft. Gut verteilte Lösungen müssen am runden Tisch erarbeitet werden. Sie sind kaum im Bereich der heute üblichen, hyperkomplexen Kommunikationsprotokolle zu suchen, sondern wohl eher unter den stochastischen Programmiertechniken der Computerviren zu finden.

(it)

```
N: constant integer; -- no value is specified here
subtype Fork is integer range 1 .. N;

task Maitre_d is
  entry Place_assignment(left,right : out Fork);
end Maitre_d;
task body Maitre_d is
begin
  for f in 1 .. N loop
    accept Place_assignment(left,right : out Fork) do
      left:=f;
      right:=f mod N + 1;
    end Place_assignment;
  end loop;
end Maitre_d;

task type Fork_proprietor is
  entry Allocate;
  entry Release;
end Fork_proprietor;
task body Fork_proprietor is
begin
  loop
    accept Allocate;
    accept Release;
  end loop;
end Fork_proprietor;

task type Philosopher;
task body Philosopher is
  left,right : Fork;
begin
  Place_Assignment(left,right);
  loop
    -- think
    if odd(left)
      then
        Allocate(left);
        Allocate(right);
      else
        Allocate(right);
        Allocate(left);
    end if;
    -- eat
    Release(left);
    Release(right);
  end loop
end Philosopher;

Philosophers : array (1 .. N) of Philosopher;
Fork_proprietors : array(1 .. N) of Fork_proprietor;
```

Cargill bemüht einen Platzanweiser an den Eßtisch der fünf Grübler. Sein Ada-Programm ist simpel, und es funktioniert ganz ordentlich. Dafür gibt es hier keine Gleichberechtigung mehr.

Literatur

[1] E. W. Dijkstra, Hierarchical Ordering of Sequential Processes, Acta Informatica, 1(2), 1971, S. 131

[2] C. A. R. Hoare, Communicating Sequential Processes, Communications ACM, 21(8), 1978, S. 666–677.

[3] R. C. Holt, G. S. Graham, E. D. Lazowska, M. A. Scott, Structured Concurrent Programming with Operating System Applications, Addison-Wesley, Reading, Mass., 1978.

[4] E. W. Dijkstra, Co-operating Sequential Processes, in: Programming Languages, Hg.: F. Genuys, Academic Press, New York 1968, S. 43–112.

[5] E. W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer, New York, 1982.

[6] Ernest Chang, n-Philosophers: an Exercise in Distributed Control, Computer Networks, 4, 1980, S. 71–76.

[7] T. A. Cargill, A Robust Distributed Solution to the Dining Philosophers Problem, Software – Practice and Experience, 12, 1982, S. 965–969.

[8] Nissim Francez, Michael Rodeh, A Distributed Abstract Data Type Implemented by a Probabilistic Communication Scheme, Proc. 21st Annual Symp. on the Foundations of Computer Science, 1980, S. 373–379.

[9] Andrew S. Tanenbaum, Betriebssysteme, Entwurf und Realisierung, Teil 1: Lehrbuch. Hanser, Prentice Hall International 1990