*Ein Spaziergang durch zwanzig Jahre Multitasking*

## Am Runden Tisch

*Die Anekdote der dinierenden Philosophen wurde von Edsger Wybe Dijkstra 1971 erfunden. 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 dringender denn je, wie ein hungriger Philosoph zu seinen Spaghetti kommt.*

### Das Problem der fünf Philosophen

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

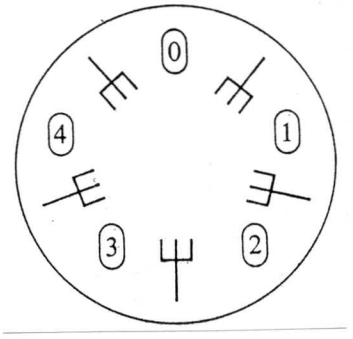



**Figure 1**: Dijkstras Tisch

Ihr einziges Problem – abgesehen von solchen philosophischer Natur – besteht darin, dass die aufgetragene Mahlzeit eine sehr schwierige Art von Spaghetti ist, zu deren Verzehr zwei Gabeln benötigt werden. Neben jedem Teller liegen zwei Gabeln, sodass dies zwar grundsätzlich keine Schwierigkeiten bereitet: jedoch hat das zur Folge, dass zwei Nachbarn nicht zur selben Zeit essen können."[1]

### Deadlock und individueller Hungertod

Seit Dijkstra diese kleine Anekdote vor zwanzig Jahren vorgestellt hat, sind viele Lösungen für dieses Problem vorgeschlagen worden. Die meisten dieser Lösungen 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,

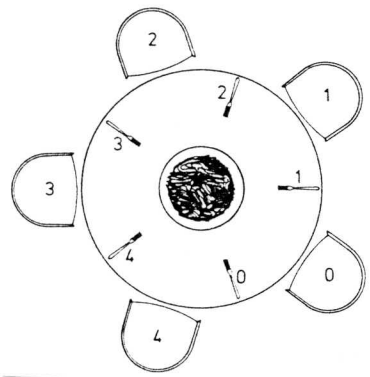
1. übersetzt aus: E. W. Dijkstra, Hierarchical Ordering of Sequential Processes, Acta Informatica, 1(2), 1971, S. 131

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.

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

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

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




```
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||phil(i:0..4)::PHIL].
```

**Figure 2**: Hoares Spaghetti

Diese schöne und klare Lösung hat nur den Nachteil, dass sie zum „Deadlock" führen kann: es ist ein zeitlicher Ablauf vorstellbar, wo alle Philosophen mehr oder weniger zur selben Zeit hungrig werden, gemäss der Anweisung die linke Gabel ergreifen und dann ewig auf die rechte warten. Der Gesamtdurchsatz des Systems ist für immer auf Null gesunken.

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

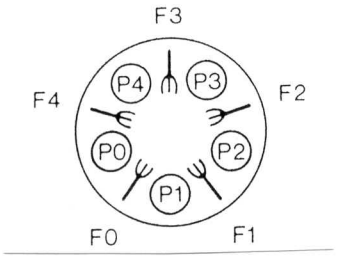
Auf dem Titelbild des Kompendiums über die Implementation von Betriebssystemen haben Holt, Graham, Lazowska und Scott zwar E. W. Dijkstra den Ehrenplatz zugewiesen.[3]

**Figure 3**: Autoren und Grossmeister?

Die Lösung, die sie in diesem Buch 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.

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




```
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)FIXED;
      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;

    ... (other philosophers) ...

    END /* OF SPAGHETTI */;
```

Figure 4: Der Tisch der Betriebssysteme

Man kann leicht zeigen, dass mit diesem „Programm" zwar der Deadlock, wo niemand mehr zum essen kommt, ausgeschlossen ist, dass aber der individuelle Hungertod dem Einzelnen immer noch droht: Wenn sich die beiden Nachbarn rechts und links leicht überlappend beim Essen abwechseln, verhungert man glatt vor der vollen Spaghettischüssel!

Eisenbahnsignale gegen die Komplexitätsexplosion

Dijkstra hatte schon drei Jahre vor der Erfindung der Philosophenparabel[4] die Grundelemente für ihre Lösung vorgeschlagen. 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 Aufgaben änderten oder sich die Zusammensetzung der Peripheriegeräte änderte, musste das Programm nachgeführt werden. Je mehr Ressourcen den immer mächtiger werdenden 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, wurde mit einem flexiblen Mechanismus versehen, solche Anwenderteile zu laden und laufen zu lassen. 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.

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 Kommunikationsmechanismus zwischen Prozessen und Geräten schlug er vor, die von der holländischen Eisenbahn inspirierte „Semaphore" zu verwenden. Für Betriebssysteme, welche diese klare Trennung mehrerer Prozesse und Geräte sauber handhaben, hat sich der Name Multitasking-Systeme eingebürgert.

Eine „Semaphore" ist ein Signal, das von den verschiedenen Prozessen, und nicht von einer Zentralinstanz, betätigt wird. Die Prozesse kooperieren miteinander, und warten, wenn „Warten" signalisiert ist. 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 das selbe Signal stellen kann. In einer Welt (einem Betriebssystem), die mit diesen beiden Grundoperationen ausgerüstet ist, können unerwünschte zeitliche Abläufe verhindert werden und damit das Verhalten unabhängiger Instanzen transparent gestaltet werden.

Das klassische Grunderfordernis für sauberes Programmieren gleichzeitiger Prozesse, ist die Möglichkeit des gegenseitigen Ausschlusses, während gewisser kritischer Operationen. Dieser gegenseitige Ausschluss („mutual exclusion") wird mit den Semaphorenoperationen leicht bewältigt. Eine Semaphore MUTEX, die auf

---

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

FREI steht, wird vom ersten Prozess, der eine kritische Region durchlaufen will, vorher auf WARTEN gestellt. Weitere Prozesse, die ihrerseits in die kritische Region einzutreten wünschen, stehen vor dem Signal Schlange und warten bis der erste Prozess das Signal 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 gehalten werden 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 bekanntmachten, 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 weit verbreiteten 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 MS-DOS.

## Unabhängigkeit und Kooperation

Die Grundeigenschaften der Semaphoren führen dazu, dass die verschiedenen Prozesse weitgehend unabhängig von ihrer zeitlichen Durchmischung betrachtet werden dürfen und dass die Korrektheit der einzelnen Prozesse wieder einfach zu beweisen ist, ohne dass das Verhalten aller anderer Prozesse mitberücksichtigt zu werden braucht.

Eine korrekte, aber ungünstige Lösung der Ressourcenzuteilung beim Philosophenproblem 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 andern Seite rot. (An Stelle der wenig bekannten Eisenbahnsemaphore, verwenden wir das Bild der Verkehrsampel). Zu Beginn des Gelages, wird üblicherweise angenommen, dass alle Philosophen sich 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.

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 wir zu stark sequentialisiert haben. Grundsätzlich könnten immer zwei auf

einmal essen, ohne sich gegenseitig zu stören. Mit unserer Radikallösung haben wir den Durchsatz des Systems stark verschlechtert. Dies wird besonders deutlich, wenn man etwa vom Fünf-Philosophen-Problem übergeht zum Fünfhundert-Philosophen-Problem.

Dijkstra stellt fest, dass ein Philosoph genau zum Essen übergehen muss, wenn er hungrig ist, und keiner seiner Nachbarn am Essen ist. Ein solcher Zustand kann auf zwei Arten erreicht werden: durch das Hungrigwerden eines Philosophen oder durch das Freiwerden von Gabeln nach dem Essen eines Philosophen. Beim Übergang zum Hunger sollte man also testen, ob beide Gabeln zur Verfügung stehen und daher gleich mit dem Essen begonnen werden sollte. Bei Beendigung des Essens sollte man 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 eine private Semaphore statt, die 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 dann 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.

In the universe we assume declared

1) the *semaphore* mutex, initially $=1$
2) the *integer array* $C[0:4]$, with initially all element $=0$
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 now 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.
```

Figure 5: Dijkstras Lösung

Der gegenseitige Ausschluss, den wir oben kritisiert hatten, findet auch in dieser Lösung statt. Wir nehmen aber an, dass die Semaphoren-Operationen „kurze" Operationen sind im Vergleich mit der Aktivität Essen. 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. Genau genommen handelt es sich um eine sorgfältig formulierte Lösung des oben angeführten Typs 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 der Deadlock. Das individuelle Verhungern ist für ihn sekundär. Dijkstra behauptet, dass eine vollständige Lösung die Einführung eines weiteren Zustands „Sehr Hungrig" nötig macht, ansonsten aber nach demselben Schema vorgegangen werden kann.

Dijkstras Semaphoren sind Grundelemente der meisten modernen Multitasking-Betriebssysteme geworden. Wie Dijkstras Lösung des Philosophen-Problems zeichnen sich diese Systeme aus durch hohe Unabhängigkeit der Teilnehmer, durch Diskretheit des Betriebssystems — das nur in Erscheinung tritt, bei der Behandlung der globalen Semaphoren und dem Entscheid, welcher der auf eine Semaphore wartenden Prozesse aufgeweckt werden soll — , durch die Gleichberechtigung der Teilnehmer, durch die Symmetrie ihrer Programme, durch ihre Kooperationsbereitschft und durch die gute Ausnützung der Ressourcen.

## Mücken und Elefanten

Wie Dijkstra an anderer Stelle schreibt,[5] war es sein Ehrgeiz, aus einer Menge Mücken 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 Ganzen 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 vorwärts getrieben worden.

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 Multitasking-AmigaDOS einzuholen, hat Apple angekündigt, das neue Betriebssystem des Mac werde dann ein echtes Multitaskingsystem sein — unter Preisgabe von viel Kompatibilität mit alten Anwendungen. Die Minis mit dem Pionier-Multitaskingsystem UNIX waren besonders bei Wissenschaftlern beliebt, die mit vielen Geräten

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

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 Betreibern grösserer lokaler Netze, wegen seiner kommunikationsfreudigeren Multitasking-Eigenschaften.

Es wird behauptet, dass wir am Ende der PC-Dekade stehen und dass die letzte Dekade dieses Jahrtausends sich als die Netzwerk-Dekade angekündigt hat. Neben den Multitaskingsystemen mit ihren eng gekoppelten 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 in diesem verteilten Szenario noch viel weiter 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 wenigstens 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 auf rot für alle, für immer.

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-Zustand 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. 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 Ein-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 Aussender Deadlock melden, der hat aber inzwischen schon gegessen.

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

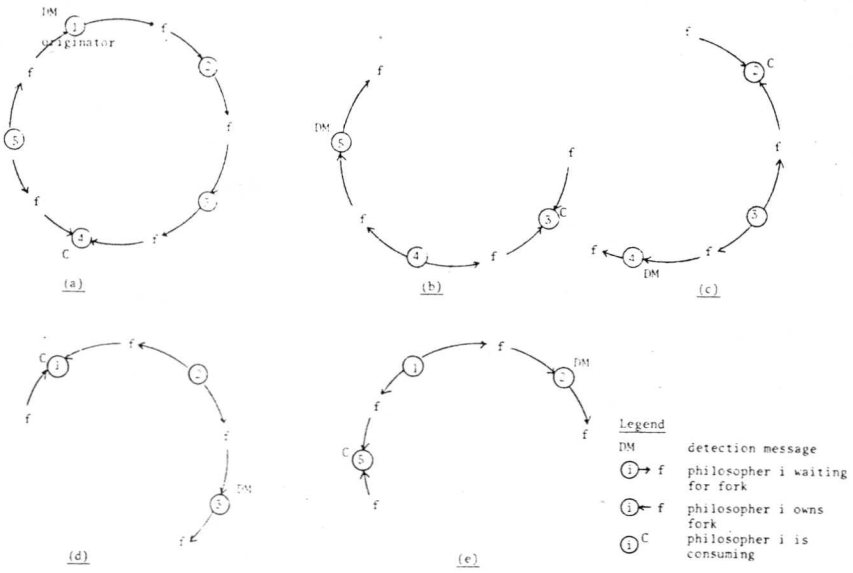



Fig. 1. Illustrating a Wave of Consumption with a trailing Detection Message.

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 passes it on; in a non-critical state, stamps it OK.
3. A philosopher getting back its Detection Message unchanged detects deadlock if its Consumptiom Sequence Number is still the same as the one on its Detection Message.
4. In the best solution, a philosopher detecting deadlock issues a Relinquish 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 relinquished fork.
5. An alternate solution is for a mutual exclusion mechanism to be invoked, which allows only one philosopher to emit the Relinquished Message. An R-philosopher receiving a Relinquish Message gives up his held fork and request, going back to the thinking state. He passes on his R-token to his immediate neighbour, in a consistent direction.
6. A philosopher who consumes increments his Consumption 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 relinquishing for any particular philosopher. In both cases, ($n$ div 2), spaced alternately, will give maximal parallel activity following deadlock detection.

**Figure 6: Changs Fresswelle**

Falls nun Deadlock festgestellt wird, kommt in Changs Lösung ein kompliziertes Verfahren in Gang, wo 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 Deadlockmeldungen verloren gehen, wenn irgendetwas nicht rund läuft, verhungern alle! Eine überraschend primitive robuste distributive Lösung von Cargill[7] hat keinen dieser Defekte 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).

7. T. A. Cargill: A Robust Distributed Solution to the Dining Philosophers Problem, Software—Practice and Experience, **12**, 1982, S. 965-969

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.

```
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;
```

Figure 7: "Maître d'" in Ada

## Kommunikation und Zufall

Die Unsicherheiten und blinden Zufällige 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, da solche Programme mit ihrer Fehlerbehandlungskomplexität oft die Transparenz verlieren und ihre Korrektheit nicht mehr garantiert werden kann. Immer häufigere

Deadlocksituationen in grösseren Banknetzwerken, die durch immer phantastischere Flickmassnahmen behoben werden, 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 stecken bleiben, dringen Computerviren ein, obwohl ihnen im allgemeinen der Weg nicht gerade geebnet wird. Die Kommunikationsleistung und die Transparenz und Korrektheit stochastischer Programme lässt uns hier Lösungen für heutige Kommunikationsprobleme suchen.

Eine stochastische Lösung des distributiven Philosophenproblems wird von Francez und Rodeh vorgeschlagen.[8] Diese Autoren implementieren das Problem in CSP und führen neben den Philosophen noch die Tellerprozesse ein.

```
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 describe 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
          ]
```

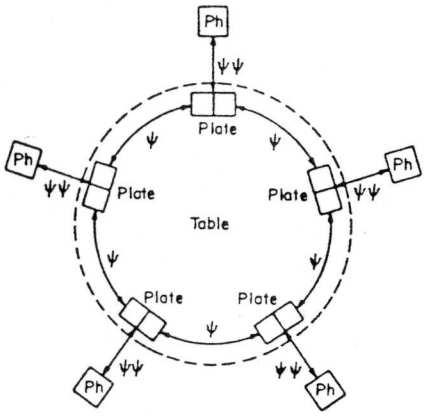



Figure 8: Tisch aus Israel

Wenn du hungrig bist und keine Gabel besitzest, warte, bis einer deiner Nachbarn denkt und nimm dann seine Gabel an dich. Wenn du hungrig bist, und schon eine Gabel --- zum Beispiel die linke --- besitzest, so warte, bis enweder 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

8. Nissim Francez und 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

die Autoren nun den göttlichen Zufall zur Hilfe und wählen zufällige Zeitspannen für die Beschäftigung mit der linken und rechten Gabel. Die Wahrscheinlichkeit, dass diese systematisch ungünstig sind, gehe gegen Null. Inwiefern diese Strategie gegen die oben angeführten überlappend essenden Nachbarn hilft, wird aus dem Artikel nicht ganz klar. Die Lösung scheint sich von derjenigen von Holt, Graham, Lazowska und Scott nur im Formalismus zu unterscheiden.

Desto 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 Philosophenproblems die Anforderungen nicht erfüllt, scheint der eingeschlagene Weg erfolgversprechend. Es genügt für reale Situationen, wenn die Wartezeiten umso unwahrscheinlicher werden, je länger sie dauern. Ein individuelles Verhungern mit asymptotischer Wahrscheinlichkeit Null kann in Kauf genommen werden, 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. Gute 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.

H. Thomas Juli 1990