

```

$$ FETCH C1;
/* Will cause one tuple to be assigned to
the structure C1 */
DO WHILE (STATUS = 0);

    PUT SKIP(2) LIST( C1.PNAME, C1.SALARY);
    SAL = SAL + SALARY;
    COUNT = COUNT +1;

$$ FETCH C1;

END;

PUT SKIP(2) LIST('Average Salary is ',
SAL/COUNT);

$$ CLOSE C1;

END REPORT;

```

CONCLUSION

The WVU DBMS has been very well received by students in the undergraduate database course at West Virginia University. It is easy to use, simple in design yet contains most of the features that a database instructor would want to illustrate in the discussion of the relational algebra.

DINING PHILOSOPHERS-- continued from page 24

ing line after thinking and before eating even if neither of his neighbors is eating. Starvation would mean that some hungry philosopher stays in the waiting line forever. The solution guarantees that whatever hungry philosopher P is first in line is waiting for at least one of his two neighbors to complete his dinner. This must happen within T time units, at which time P begins to eat and is removed from the waiting line. All other waiting philosophers move one step closer to the beginning of the line. Thus P will have to wait at most $3T$ time units before beginning to eat and starvation is precluded.

Maximal Efficiency: The solution achieves maximal efficiency since only those philosophers who are neighbors of the philosopher at the head of the waiting line are constrained from beginning to eat by the solution's control protocol. Maximal efficiency does not allow the situation where one philosopher dines while the rest wait to eat. This cannot happen in this solution since at most one neighbor of the head philosopher can be the solitary dining philosopher. If the solitary dining philosopher is not a neighbor of the head philosopher, the head philosopher may proceed to eat since his neighbors are not eating. If the solitary dining philosopher is a neighbor of the head philosopher, then the one philosopher who is a non-neighbor of both the head philosopher and the solitary dining philosopher may begin to eat.

Finally this solution is symmetric with respect to the philosopher processes since they are identical in form. The reader may show that the solution will continue to work when the number of philosophers N is increased.

The WVU DBMS executes on the VAX line of computers running VMS. The WVU DBMS is available at no cost to any academic institution that wishes to use it. The User's Guide and complete syntax as well as instructions for acquiring the software can be obtained by writing:

John Atkins
Department of Statistics
and Computer Science
305 Knapp Hall
West Virginia University
Morgantown, W.Va. 26506

REFERENCES

1. Atkins, J.M., and D.M. Henry, "A Database Management System Project for an Undergraduate Database Design Course," Proceedings of the 1985 ACM Annual Conference, 1985, pp. 266-270
2. Stout, Quentin F. and Patricia A. Woodworth, "Relational Databases," The American Mathematical Monthly, February, 1983, Vol.90, No. 2, pp. 101-118
3. Ullman, Jeffrey D., Principles of Database and Knowledge-Base Systems, Volume 1, Computer Science Press, 1988

Bibliography

- [1] Dijkstra, E.W. *Cooperating Sequential Processes*, Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.
- [2] Dijkstra, E.W. Hierarchical ordering of sequential processes. *Acta Inf.*, 1(1971), 115-138.
- [3] Hoare, C.A.R. *Communicating Sequential Processes*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1985.
- [4] Holt, R.C. *Concurrent Euclid, The UNIX* System, and TUNIS*, Addison-Wesley, 1983.
- [5] Ringwood, G.A. Parlog86 and the Dining Logicians, *Comm. ACM*, Jan. 1988, 10-25.
- [6] Tanenbaum, A.S. *Operating Systems: Design and Implementation*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1987.

Dining Philosophers Revisited

Armando R. Gingras
Metropolitan State College
Denver, Colorado

Abstract

In 1965 Dijkstra posed and solved the Dining Philosophers problem. Since then the problem has become a classic test case for concurrency mechanisms and an example often discussed in operating systems courses. Two theorems prove the correctness of seatings where all philosophers always prefer to pick up a fork with a particular hand first. This note shows the subtlety required to solve the problem by showing that a recently published solution is incorrect. A correct solution is provided.

The Problem

This note discusses the well-known Dining Philosophers Problem. It was originally stated and solved by E.W. Dijkstra in 1965 [1 and 2]. The problem has become a standard synchronization test case [5].

The problem is stated as follows. There are five philosophers, P_i , $i = 0, \dots, 4$, who devote their time in an endless cycle of thinking, hungering, and eating. They sit at their own chair around a circular table with P_{i+1} sitting to the right of P_i , where addition is modulo 5. Before each philosopher is a plate of spaghetti and between each pair of adjacent plates is a fork. See figure 1 for a diagram of the table arrangement of the five plates and forks.

Requiring two forks to eat spaghetti, each philosopher can only use those two forks on either side of his plate, i.e., a philosopher cannot reach across the table for a fork. Whenever a philosopher becomes hungry, he must wait until these two forks are free to use. When the two forks become available, he holds them until he is finished eating, at which time he releases them and sinks into a thinking stupor. A neighbor may then pick up the dirty fork and proceed to use it if he is hungry enough. The problem is to devise an algorithm that guarantees minimally that no two neighboring philosophers eat simultaneously.

There are several other assumptions concerning the problem. The philosophers are so myopic that they cannot see what any other philosopher is doing and so act independently and asynchronously of one another. The time given over to any activity by a philosopher is unknown. Thus the solution cannot make use of special knowledge of individ-

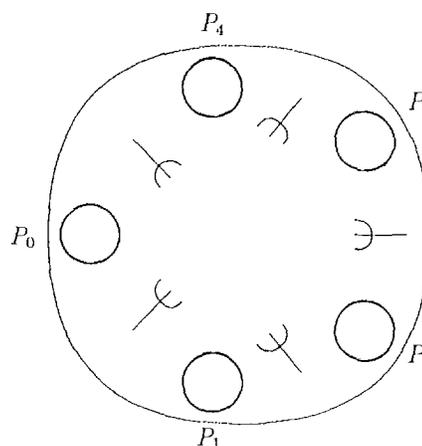


Figure 1: Seating setting.

ual eating and thinking patterns. However, there may be a maximum T placed on the time that a philosopher can eat so that an estimate can be made of the maximum time a hungry philosopher must wait before eating. Also, a philosopher cannot be an overly greedy eater, i.e., once a fork is released by a philosopher, it cannot be picked up again by that philosopher until his neighbor, if he is hungry, has had a chance to pick up that fork. And no philosopher is altruistic enough to stage a hunger strike for more forks. Moreover, there are basic requirements for an acceptable solution.

Solution Requirements

Any acceptable solution to the Dining Philosophers problem must meet several minimum criteria. First of all, the philosophers must observe basic table manners, i.e., no reaching across the table, no hoarding or hiding of forks, etc. One particular aspect is that no fork can be shared simultaneously by two philosophers. This is the *mutual exclusion* property.

Another requirement of any solution is that it must avoid *deadlock*. Deadlock occurs when some subset of philosophers find themselves obstinately holding one fork and waiting for the second fork, and no one in the subset can eat.

The third requirement is the prevention of *starvation*. Starvation occurs when a continuing pattern of dining can occur that systematically locks out at least one philosopher from eating.

Since there are five forks, two philosophers can eat simultaneously. The fourth requirement is that a solution have maximal concurrency or *efficiency*. An efficient solution never allows four hungry philosophers to wait on one dining philosopher.

Synchronization Primitives

To understand the approaches discussed below, two synchronization primitives will be reviewed briefly.

In 1965 Dijkstra introduced the **semaphore**, to count the number of sleeping processes waiting for a wakeup call. He introduced two operations on semaphores to manipulate their values. A process uses the **down** operation on a semaphore as follows: if the semaphore's value is positive, the semaphore is decremented and the process continues; if the value is zero, the process is put to sleep. When a process uses an **up** operation on a semaphore, the semaphore is incremented; if its value was zero, one of the sleeping processes is awakened and completes its down operation. See [6] for a fuller discussion of semaphores.

The second mechanism used in this article is the **monitor**. A monitor, as implemented in the programming language Concurrent Euclid, is a module that enforces mutually exclusive use of its procedures and data by processes requiring coordination. It manages condition queues wherein delayed processes may wait. Interprocess synchronization is effected by two statements: **wait(cond)**, places a process in a sleeping state in queue *cond*, and **signal(cond)**, which awakens a waiting process queued in *cond*. See [4] for more details on monitors.

While the concurrency mechanisms provided by a specific programming language must certainly be adequate to solve a given problem, a correct algorithm is still necessary.

Pseudo-Solutions

One trivial solution is to have all philosophers enter a queue when they become hungry. While this linearization avoids deadlock and starvation, it fails to provide any concurrency. A second possible solution is to have each philosopher, when struck by hunger, to grab the first available fork (or wait if until one is free), and wait for the second fork while holding the first. Unfortunately, it is possible for all five philosophers to simultaneously grab their left forks, and enter a circular deadlock waiting for their obstinate right neighbors to release the desired forks.

To demonstrate the subtlety of the problem, consider a proposed solution reproduced in figure 2 as given by Tanenbaum [6, p.78] in an undergraduate textbook on operating systems. When a philosopher finishes eating, he allows his left neighbor to proceed if possible, then permits his right neighbor to proceed. Notice that the order in which the neighbors are checked and permitted to proceed is irrelevant.

```

#define N          5 {number of phil's}
#define LEFT      (i-1)%N {i's left neighbor}
#define RIGHT     (i+1)%N {i's right neighbor}
#define THINKING  0 {Phil is thinking}
#define HUNGRY    1 {P waiting for forks}
#define EATING    2 {P is eating}
typedef int semaphore; {define a semaphore}
int state[N];          {track phil's' state}
semaphore mutex = 1;  {shared data guard}
semaphore s[N];       {i semaphore per phil}

philosopher(i)
int i;                {phil 0 to N-1}
{ while (TRUE) {      {repeat forever}
    think();          {phil is thinking}
    take_forks(i);    {get 2 forks or block}
    eat();            {yum-yum, spaghetti}
    put_forks(i);    {put both forks back}

take_forks(i)
int i;                {phil 0 to N-1}
{ down(mutex);       {enter critical area}
  state[i] = HUNGRY; {phil i is hungry}
  test(i);           {try to get 2 forks}
  up(mutex);        {exit critical area}
  down(s[i]);        {block if not 2 forks}

put_forks(i)
int i;                {phil 0 to N-1}
{ down(mutex);       {enter critical area}
  state[i] = THINKING; {phil done eating}
  test(LEFT);        {let left neigh eat}
  test(RIGHT);       {let right neigh eat}
  up(mutex);        {exit critical area}

test(i)
int i;                {phil 0 to N-1}
{ if state[i] == HUNGRY
    && state[LEFT] != EATING
    && state[RIGHT] != EATING
  { state[i] = EATING; {phil i can eat}
    up(s[i]); }}

```

Figure 2: Tanenbaum's solution.

Let us make the following simplifying assumptions for our counterexample. Our philosophers spend practically no time thinking and find themselves in the configuration where philosophers P_0 , P_1 , and P_3 wait with hunger while philosophers P_2 and P_4 dine in leisure. Now consider the following admittedly unlikely sequence of philosophers' completions of their suppers.

| EATING | HUNGRY |
|--------|--------|
| 4 2 | 0 1 3 |
| 2 0 | 1 3 4 |
| 3 0 | 1 2 4 |
| 0 2 | 1 3 4 |
| 4 2 | 0 1 3 |

Each line of this table is intended to indicate the philosophers that are presently eating and those that are in a state of hunger. The dining philosopher listed first on each line is the one who finishes his meal next. For example, from the initial configuration, philosopher P_4 finishes eating first, which permits P_0 to commence eating. Notice that the pattern folds in on itself and can repeat forever with the consequent starvation of philosopher P_1 .

It is not the low probability of the occurrence of a troublesome pattern that renders a trial solution acceptable, but rather the mere existence of such a pattern that is not logically prevented from occurring that renders the solution unacceptable.

Asymmetric Solutions

An exercise credited to Toscani [4, p.140] asks the student to explore the situation where every philosopher but one always picks up his left fork first (a "lefty") and the last one always picks up his right fork first (a "righty"), with each philosopher always waiting when he cannot pick up his next desired fork and not releasing any forks until after finishing his supper.

Before generalizing this exercise, we make explicit the actions of a lefty using the following process form:

```
loop think
  TABLE.PickUp(left(i))
  TABLE.PickUp(right(i))
  eat
  TABLE.PutDown(left(i))
  TABLE.PutDown(right(i))
end loop
```

Using the monitor TABLE, lefty P_i makes two calls to PickUp to pick up first his left fork and then his right fork, waiting in each case until each is available. After eating, P_i returns both forks to the table. A righty is defined analogously.

The exercise is now generalized in the following two theorems to any combination of lefties and righties having at least one of each .

Theorem 1 Any seating arrangement of lefties and righties with at least one of each avoids deadlock.

Proof Assume the table is in deadlock, i.e., there is a nonempty set D of philosophers such that each P_i in D holds one fork and waits for a fork held by a neighbor. Without loss of generality, assume that $L_j \in D$ is a lefty. Since L_j clutches his left fork and cannot have his right fork, his right neighbor L_k never completes his dinner and is also a lefty. Therefore $L_k \in D$. Continuing the argument rightward around the table shows that all philosophers in D are lefties. This contradicts the existence of a righty. Therefore deadlock is not possible. ■

Theorem 2 Any seating arrangement of lefties and righties with at least one of each prevents starvation.

Proof Assume that lefty L_j starves, i.e., there is a stable pattern of dining in which L_j never eats. Suppose L_j holds no fork. Then L_j 's left neighbor must continually hold his right fork and never finishes eating. Thus L_j 's left neighbor R_i is a righty holding his right fork, but never getting his left fork to complete a meal, i.e., R_i also starves. Now R_i 's left neighbor must be a righty who continually holds his right fork, etc. Proceeding leftward around the table with this argument shows that all philosophers are (starving) righties. But L_j is a lefty: a contradiction. Thus L_j must hold one fork.

As L_j continually holds one fork and waits for his right fork, L_j 's right neighbor never sets his left fork down and never completes a meal, i.e., L_j 's right neighbor L_k is also a lefty who starves. If L_k did not continually hold his left fork, L_j could eat; therefore, L_k holds his left fork. Now carry the argument rightward around the table to show that all philosophers are (starving) lefties: a contradiction. Starvation is thus precluded. ■

It is left as an exercise for the reader to show that in any arrangement of righties and lefties maximal efficiency is not attainable by exhibiting table configurations where at most one philosopher eats while the rest must wait hungry.

The Footman Solution

One common solution is to allow at most four philosophers to eat or hunger at any time. This can be achieved by removing one chair from the table and requiring that a philosopher can contend for forks only if he is seated. Whenever a philosopher finishes a meal, he sets his forks down and stands up. If another philosopher is standing, he may sit first. If no other philosopher is standing, the philosopher just completing his meal may sit down again when hunger strikes. The coordination and distribution of the chairs may be handled by introducing a new agent, a footman, as is done in [3]. This more complex arrangement leads to an efficient correct solution at the cost of greater inter-philosopher communication.

A Proffered Solution

Instead of focusing on the forks individually, the following solution, written in Concurrent Euclid [4], looks at the actions of the philosophers themselves.

The round of activities of philosopher P_i are modeled by an endless loop within a process of the following form:

```
process PHILi
  imports (var DINER)
  begin var i: 0..M
    loop DINER.THINK(i) busy(A) {A=thinking delay}
      DINER.HUNGER(i)
      {EAT} busy(B) {B=eating delay}
    end loop
  end PHILi
```

The central controlling mechanism is the monitor DINER. It uses a waiting Line with the special property that all ad-

ditions are at the end of the line, but any waiting philosopher in line may leave the line, not just the one at the head of the line. The current length of Line is kept in LineLen. The boolean array Eating indicates which philosophers are presently eating. Finally the array Phil has one condition variable for each philosopher to effect eating delays.

The monitor DINER begins with four internal utilities. The boolean function NonAdj returns true if two given philosophers are not adjacent neighbors at the table. The boolean function Ready returns true if a particular philosopher's two neighbors are both not eating. The procedure EnLine adds a philosopher to the end of Line, while the DeLine removes a particular philosopher from within Line.

```

var DINER :                               {N=5; M=N-1}
  monitor
  exports (THINK,HUNGER)

  var Line : array 1..N of 0..M {hunger wait line}
  var LineLen : 0..N := 0      {# waiting in Line}
  var Eating : array 0..M of boolean {phils eating}
  var Phil : array 0..M of condition {wait queues}

FUNCTION NonAdj(j: 0..M, i: 0..M) {T if i,j not neighs}
  returns b: boolean = {note no phil is adj to self}
  begin return (not( (j=(i+M) mod N) or
                    (j=(i+1) mod N)))

  end NonAdj

FUNCTION Ready(i: 0..M) {T if no neigh is eating}
  returns b: boolean =
  imports (Eating)
  begin return ( (not Eating((i+M)mod N)) and
                (not Eating((i+1)mod N)) )

  end Ready

PROCEDURE EnLine(i:0..M) = {i to end of Line}
  imports (var LineLen, var Line)
  begin LineLen := LineLen + 1
        Line(LineLen) := i
  end EnLine

PROCEDURE DeLine(k:1..N) = {dele kth phil in line}
  imports (var LineLen, var Line)
  begin var j : 1..N
        j := k
        loop exit when j = LineLen {move up}
              Line(j) := Line(j+1) {followers}
              j := j+1 {to close gap}
        end loop
        LineLen := LineLen - 1
  end DeLine

```

The heart of the monitor is the procedure Next, which attempts to find the longest waiting member of Line who has the utensils to begin eating provided he is not adjacent to the philosopher at the head of the line. If one is found, that philosopher is removed from Line and set to eating. Note that the philosopher at the head of the line, the *head* philosopher, is not adjacent to himself.

```

PROCEDURE Next =
  imports (var Eating, var Phil, var LineLen,
          var Line, NonAdj, Ready, DeLine)
  begin var k : 1..N var i : 0..M
        if LineLen > 0 then {feed oldest elig}
          k := 1 {waiting phil}
          loop if (NonAdj(Line(k),Line(i)) and
                  Ready(Line(k)))
            then Eating(Line(k)) := true
                  i := Line(k)
                  DeLine(k)
                  Signal(Phil(i))
                  exit
            else exit when k = LineLen
                  k := k+1
            end if
          end loop
        end if
  end Next

```

The final two procedures are the only ones available to the several philosopher processes. The procedure THINK sets the calling philosopher to thinking after eating and tries to start another philosopher eating by calling Next. The final procedure HUNGER places the calling philosopher at the end of the waiting Line and checks which waiting philosopher can eat next. If this does not turn out to be the calling philosopher, then he is made to wait.

```

PROCEDURE THINK(i: 0..M) =
  imports (var Eating, Next)
  begin Eating(i) := false
        Next
  end THINK

PROCEDURE HUNGER(i: 0..M) =
  imports (var Eating, var Phil, var LineLen,
          var Line, EnLine, Next)
  begin EnLine(i)
        Next
        If not Eating(i) then wait(Phil(i))
        end if
  end HUNGER

end monitor {Diner}

```

Solution Verification

To show that the given solution is correct, we must verify that it meets the minimum requirements of any acceptable solution.

Mutual Exclusion: No two adjacent philosophers simultaneously use the fork placed between them. This cannot happen since no philosopher is permitted to eat if either of his neighbors is eating.

Deadlock Avoidance: If deadlock were present in some configuration, then each philosopher would be holding some resource, i.e., a fork, while waiting for some other resource, i.e., a second fork. However, a philosopher either holds two forks (while eating) or no fork (while thinking or waiting).

Starvation Prevention: Each philosopher enters the wait-

DINING PHILOSOPHERS-- continued on page 28