

Dining Philosophers Revisited, Again

Kwok-bun Yue
University of Houston - Clear Lake
2700 Bay Area Boulevard
Houston, TX 77058

Abstract

This paper describes a problem in the solution of the dining philosophers problem by Gingras [2] that makes it inefficient, instead of the claimed maximal efficiency. A correct implementation is presented. Even then, the solution has other undesirable characteristics and is still not maximally efficient. Depending on the definition of efficiency, it may not be possible to attain maximal efficiency and be starvation-free at the same time. A better and simpler solution for general mutual exclusion problems, in which the dining philosophers problem is a special case, is presented. This solution can become symmetric if appropriate data structures are used.

Introduction

In a recent article in the SIGSE Bulletin, titled "Dining Philosophers Revisited", Gingras presented some results on the well-known dining philosophers problem [2]. He presented an asymmetric solution and a symmetric solution. He proved that the asymmetric solution, where all philosophers always prefer to pick up a fork with a particular hand first, are both deadlock-free and starvation-free. The article also pointed out, as an example of the subtlety required to solve the problem, that the solution in a popular operating systems book [6] is not starvation-free. The symmetric solution is then proposed as one that is starvation-free and also attains maximal efficiency.

In this paper, the dining philosophers problem is considered in the context of general mutual exclusion problems. A problem in Gingras's solution that actually makes it inefficient is identified and fixed. However, even with the fix, the solution still has some shortcomings and does not attain maximal efficiency. It is shown that maximal efficiency may lead to starvation and it is necessary to compromise between the absence of starvation and efficiency.

A simpler and better method for generating asymmetric starvation-free solutions for general mutual exclusion problems is presented. With appropriate data structures, the solution can be converted to a symmetric one where every process uses the same synchronization code.

The Problem

In the dining philosophers problem, there are five philosophers sitting at a round table, with a chopstick between each pair of neighboring philosophers. The philosophers only do two things: eating and thinking. To eat, a philosopher must pick up the two chopsticks closest to him. The problem is to synchronize the actions of the philosophers such that certain requirements must be satisfied.

The dining philosophers problem is an example of general mutual exclusion problems. If the action of a philosopher is a process, then eating is a critical section mutually exclusive to the critical sections of neighboring philosophers.

A simple way to represent mutual exclusion problems is by graphs [5]. A process is represented by a node and a mutual exclusion constraint between two processes is represented by an edge between the corresponding two nodes. For example, the dining philosophers problems can be represented by the mutual exclusion graph in Figure 1.

There can be many requirements for the solutions of mutual exclusion problems [2,7]. The fundamental one being the mutual exclusion constraint which states that no two neighboring processes can be in their critical sections in the same time. The solutions should also avoid deadlock and starvation and be efficient -- allows reasonable degree of concurrency. In the dining philosophers problem, deadlock occurs when every philosopher is holding a chopstick, not giving up it and waiting for the other chopstick. Starvation occurs

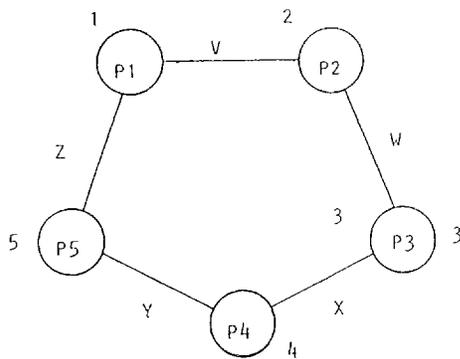


Figure 1 The Mutual Exclusion Graph for The Dining Philosophers Problem

when the two neighbors of a philosopher eat in turn, thus denying the philosopher to eat indefinitely.

Gingras's Solution

In [2], Gingras presented a solution (Figure 2) to the dining philosophers problem using monitor [3] in Concurrent Euclid [4] that is symmetric, deadlock-free, starvation-free and achieves maximal efficiency.

```

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

var DINER :
  monitor
  exports (THINK,HUNGER)
  var Line : array 1..N of 0..M (hunger wait line)
  var LineLen : 0..M := 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 M) or
    (j=(i+1) mod M)))
  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 M)) and
    (not Eating((i+1) mod M)) )
  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

```

```

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(1)) 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

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, ar LineLen,
    var Line, EnLine, Next)
  begin EnLine(i)
    Next
    if not Eating(i) then wait(Phil(i))
    end if
  end HUNGER

```

Figure 2 Gingras's Solution

Essentially, Gingras's solution is to use a waiting Line for the philosophers waiting to eat. A waiting philosopher is put at the end of the Line. A philosopher waiting in the Line is allowed to eat if (1) no neighboring philosopher is eating and (2) the head of the Line is not its neighbor. The second condition is added to avoid starvation.

In Figure 2, the most important procedure is Next which finds the next philosopher in Line that should be allowed to eat to do so. The procedure Next is called in two locations: in the procedure THINK when a philosopher has just finished eating and thus its neighbors may be allowed to eat; and in the procedure HUNGER when a philosopher wants to eat and thus wants to check whether he is allowed to do so.

The problem with Gingras's solution is that the procedure Next finds only one waiting philosopher in Line who should be allowed to eat to do so. However, there may be more than one philosopher in Line that should be allowed to eat. For example, in Figure 1, suppose philosophers P1 and P3 are hungry and waiting in Line and philosopher P2 is eating. If philosopher P2 has just finished eating now and calls the procedure Next within the procedure THINK, only one of the philosophers P1 and P3 will be allowed to eat whereas both should be allowed to eat.

A correct implementation

The correct implementation of Next is of course to allow all ready philosophers

to eat. This is done by examining the entire Line starting from the Head. Figure 3 is the correct implementation of the procedure Next. The only necessary change is to add the condition "when k > LineLen" to the exit statement to force iteration.

```

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(1)) and
              Ready(Line(k)))
        then Eating(Line(k)) := true
             i := Line(k)
             DeLine(k)
             Signal(Phil(i))
             exit when k > LineLen (change here)
        else exit when k = LineLen
             k := k + 1
        end if
      end loop
    end if
  end Next

```

Figure 3 The correct implementation of Next

This solution can be used for general mutual exclusion problems. The only necessary change is the function NonAdj and Ready to reflect the topology of the problem.

The resulting solution is a faithful implementation of the strategy:

A process is blocked or remains blocked if and only if it has a neighbor in critical section or it is the neighbor of the process blocked for the longest time.

This and other strategies to construct solutions for graphical mutual exclusion problems are discussed in [7].

Even with the correction, Gingras's solution has other undesirable characteristics.

- o The code is too long.
- o A lot of variables are needed. In particular, N Boolean variables (i.e. Eating), N conditions (i.e. Phil) and a line of length N is necessary, where N is the number of processes in the problem.
- o The time for executing Next is O(N).
- o Since the solution uses a monitor and a monitor allows only one process inside it at a time, this may represent a significant bottleneck in synchronization for complex problems with a lot of processes.

Furthermore, the solution does not provide maximal efficiency, as claimed by Gingras. It is very difficult to measure the degree of concurrency. One possible way of defining maximal efficiency is to

maximize the current number of processes in critical sections. If this is so, then the following strategy will ensure maximal efficiency.

A process is blocked or remains blocked if and only if one or more of its neighbors are in their critical sections.

It is easy to see that this strategy is more efficient, defined as above, than that of Gingras. However, the strategy is not starvation-free. Thus, maximal efficiency, defined as above, can be contradictory to the absence of starvation. [7] described other methods for measuring degree of concurrent activities.

A Simple Solution

There are many other strategies, with relative merits, to construct solutions for general mutual exclusion problems [8]. In this section, a simple method, based on [8], of generating solutions for general mutual exclusion problems is discussed.

Our method is based on Dijkstra's semaphore [1]. A semaphore S is a global non-negative integer variable that can be accessed only through the two primitive functions P and V (called down and up in [2,6]):

```

P(S): if S > 0 then S <-- S - 1
      else wait.
V(S): if S > 0 then S <- S + 1
      else signal a process waiting at
           P(S) to complete P(S).

```

In our method, there is a semaphore, with an initial value of 1, associated with each edge in the graph describing the mutual exclusion problem. Let N(p) be the set of all neighbors of node p in the graph. Let entry(p) and exit(p) be the synchronization code for process p, executed immediately before and after its critical section. The Pascal-like code for enter(p) and exit(p) of every node p in the graph can be obtained by the following algorithm where the symbol & is the string concatenation operator.

[1] Arbitrarily label the values 1 to N to the N nodes in the graph.

```

[2] for every node p in the graph do
  entry(p) <- "";
  exit(p) <-- "";
  work_set <-- N(p);
  while work_set is not empty do
    Let q be the node in work_set
      with the smallest value.
    Let S be the semaphore associated
      with edge joining p and q.
    entry(p) <-- entry(p) & "P(" & S
      & ");"
    exit(p) <-- exit(p) & "V(" & S
      & ");"
  end while
end for

```

The code generated by the algorithm does not have the undesirable features of Gingras's solution.

As an example, consider the graph G in Figure 4 that has six nodes, a, b, c, d, e and f, labelled with values 1, 2, 3, 4, 5, 6 respectively. There are 7 edges with associated semaphores T, U, V, W, X, Y and Z. The synchronization code for the 6 nodes generated by the algorithm is shown in Figure 5.

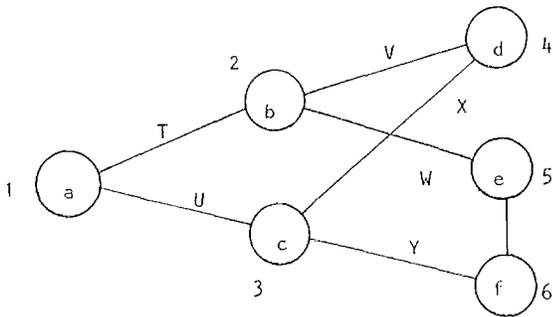


Figure 4 A mutual exclusion graph

Node p	entry(p)	exit(p)
a	P(T); P(U);	V(T); V(U);
b	P(T); P(V); P(W);	V(T); V(V); V(W);
c	P(U); P(X); P(Y);	V(U); V(X); V(Y);
d	P(V); P(X);	V(V); V(X);
e	P(W); P(Z);	V(W); V(Z);
f	P(Y); P(Z);	V(Y); V(Z);

Figure 5 Solution for Figure 4 generated by the algorithm

As another example, Figure 6 is the code generated by the algorithm for the dining philosophers problem of Figure 1, where the value associated with every node and the semaphore associated with every edge are also shown. This is similar to Gingras's asymmetric solution where philosopher p1 is a 'lefty' and all other philosophers are 'righties'.

Node p	entry(p)	exit(p)
P1	P(V); P(Z);	V(V); V(Z);
P2	P(V); P(W);	V(V); V(W);
P3	P(W); P(X);	V(W); V(X);
P4	P(X); P(Y);	V(X); V(Y);
P5	P(Y); P(Z);	V(Y); V(Z);

Figure 6 The solution for the dining philosophers problem of Figure 1 generated by the algorithm

The algorithm can actually be considered as a generalization of Gingras's asymmetric solution. Informally, deadlock and starvation is avoided because the algorithm guarantees that there is at least a 'lefty' and a 'righty' in every cycle in the graph. For a formal proof and a detailed discussion, please refer to [8].

The solutions in Figures 5 and 6 are asymmetric in the sense that each process does not have the same synchronization code. A symmetric solution can be generated if appropriate data structure is used.

Suppose there are N nodes and M edges. As an example, Figure 7 contains the necessary data declarations and code.

```
S[1..M] : semaphore;
E[1..N,1..M] : Boolean;
{E[i,j]=True iff the edge j is
incident to the node i.}
```

```
Synchronization code for node i:
entry(i):
  for j := 1 to M do
    if E[i,j] then P(S[j]);
exit(i):
  for j := 1 to M do
    if E[i,j] then V(S[j]);
```

Figure 7 A symmetric solution

More efficient data structures can be used. For example, linked lists may be used to improve performance for sparse graphs.

Conclusion

In this paper, we have demonstrated and fixed a problem in a symmetric solution by Gingras of the dining philosophers problem. Even after the correction, this solution has some undesirable characteristics. The other asymmetric solution by Gingras is

generalized so that it can deal with any mutual exclusion problem, not just the dining philosophers problem. By using simple data structures, the solution can be converted to a symmetric one where all processes have the same synchronization code.

References

- [1] Dijkstra, E.W. Cooperating Sequential Processes, in Programming Languages, Genuys, F. Ed., Academic Press, New York, 1968.
- [2] Gingras, A.R., Dining Philosophers Revisited. SIGCSE Bulletin, vol.22 No.3 (1990), 21-28.
- [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, Reading, Massachusetts, 1983.
- [5] Page, I.P. & Jacob, R.T., The Solution of Mutual Exclusion Problems which can be Described Graphically, The Computer Journal, vol.32 No.1 (1989), 45-54.
- [6] Tanenbaum, A.S. Operating Systems: Design and Implementation, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1987.
- [7] Yue K. & Jacob R.T., Starvation-Free Semaphore Solutions to Mutual Exclusion Problems, Proceedings of the 1987 ACM Southern Central Regional ACM Conference, Lafayette, Louisiana (Nov. 1987), 127-141.
- [8] Yue K., Semaphore Solutions for General Mutual Exclusion Problems, PhD dissertation, University of North Texas, Denton, Texas, 1988.

BIBLIOGRAPHY-- continued from page 59

Anneliese von Mayrhauser. *Software Engineering: Methods and Management*

San Diego, Calif.: Academic Press, 1990. ISBN 0-12-727320-4. 864 pages. \$49.95. Includes exercises.

Table of Contents

- Part 1: Methods
- 1. Introduction
 - 2. Problem Definition
 - 3. Functional Requirements Collection
 - 4. Qualitative Requirements
 - 5. Specifications
 - 6. Design: Strategies and Notations
 - 7. Software System Structure Design
 - 8. Detailed Design
 - 9. Coding
 - 10. Testing
 - 11. Operation and Maintenance

- Part 2: Management
- 12. Management by Metrics
- 13. Feasibility and Early Planning
- 14. Models for Managerial Planning
- 15. Project Personnel
- 16. Software Development Guidelines

Richard Wiener and Richard Sincovec. *Software Engineering with Modula-2 and Ada*

New York: John Wiley & Sons, 1984. ISBN 0-471-89014-6. 451 pages. Includes exercises.

Reviewed in *Computing Reviews*, October 1985.

Table of Contents

- 1. What is Software Engineering? A Top-Down View
- 2. Software Requirements and Specifications
- 3. Programming Languages and Software Engineering
- 4. General Principles of Software Design
- 5. Modular Software Development Using Ada
- 6. Modular Software Construction Using Modula-2
- 7. Programming Methodology
- 8. Software Testing
- 9. A Case Study in Modular Software Construction

INTERNET-- continued from page 54

facilities of C using additional (nonstandard) Pascal procedure names and corresponding library routines is being considered. Providing a name server for students to register and request services, thus providing a means of "broadcasting" (and connecting to) some new service, is planned. Also, an adequate means of handling INET file identifiers is being sought. At the least, the initialization of these identifiers will be moved to the *constant* section of the source code.

An early decision was made to support only character based input/output using *read(ln)* and *write(ln)*. Extensions will allow the use INET files for output of *integers*, *reals*, and *booleans*, as is required by the Pascal standard. The system will also allow access to INET files through *file pointers* using pointer notation and the Pascal procedures *get* and *put*.

In summary, the PIP system can be effectively used to introduce or reinforce concepts of networks and telecommunications using a language quite similar to Pascal. Very few changes to Pascal syntax are introduced, and PIP retains all of the features of standard Pascal. Very little class time is needed to introduce PIP. The system can be implemented on 4.2+ BSD UNIX systems with minimal effort.

REFERENCES

- Cooper, Doug. *Standard Pascal User Reference Manual*. W.W. Norton & Company, New York. 1983.
- Leffler, Samuel, Robert Fabry, and William Joy. "A 4.2BSD Interprocess Communication Primer." Computer Systems research Group, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley. 1986.
- Tanenbaum, Andrew. *Computer Networks, 2nd*. Prentice Hall, Englewood Cliffs, NJ. 1988.