4) Path P containing two nondiameter links is constructed as follows: $(I, J) \Rightarrow (I, M) \rightarrow (M, I) \Rightarrow (M, K) \rightarrow (K, M) \Rightarrow (K, L)$, where $M \neq I$ and $M \neq K$. The routing distance of P is $R_P = H(J, M) + H(I, K) + H(M, L) + 2$. From (3), $R_C = H(J, L) + H(I, K) + 2$. Since $H(J, M) + H(M, L) \geq H(J, L)$, we have $R_P \geq R_C$. Therefore, routing algorithm C provides the shortest path that contains two nondiameter links. Note that if M satisfies $H(J, M) + H(M, L) = H(J, L)$, path P has the same distance as the path by routing algorithm C and can be used as its alternate path.
5) It is obvious from the definition of routing algorithm B^*. □

REFERENCES

[1] K. Ghose and K.R. Desai, "Hierarchical Cubic Network," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 4, pp. 427–435, Apr. 1995.
[2] A. El-Amawy and S. Latifi, "Properties and Performance of Folded Hypercubes," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 1, pp. 31–42, Jan. 1991.
[3] A. Esfahanian, L.M. Ni, and B.E. Sagan, "The Twisted n-Cube with Application to Multiprocessing," *IEEE Trans. Computers*, vol. 40, no. 1, pp. 88–93, Jan. 1991.
[4] K. Hwang and J. Ghosh, "Hypernet: A Communication Efficient Architecture for Constructing Massively Parallel Computers," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,450–1,466, Dec. 1987.
[5] J.M. Kumar and L.M. Patnaik, "Extended Hypercube: A Hierarchical Interconnection Network of Hypercube," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 1, pp. 45–57, Jan. 1992.
[6] N.F. Tzeng and S. Wei, "Enhanced Hypercube," *IEEE Trans. Computers*, vol. 40, no. 3, pp. 284–294, Mar. 1991.
[7] S.G. Ziavras, "A Versatile Family of Reduced Hypercube Interconnection Network," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 11, pp. 1,210–1,220, Nov. 1994.
[8] K. Efe, "A Variation on the Hypercube with Lower Diameter," *IEEE Trans. Computers*, vol. 40, no. 11, pp. 1,312–1,316, Nov. 1991.
[9] D.R Duh, G.H. Chen, and J. F. Fang, "Algorithms and Properties of a New Two-Level Network with Folded Hypercubes as Basic Modules," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 7, pp. 714–723, July 1995.
[10] K. Ghose and K.R. Desai, "The HCN: A Versatile Interconnection Network Based on Cubes," *Proc. Supercomputing*, pp. 426–435, 1989.
[11] K. Ghose and K.R. Desai, "The Design and Evaluation of the Hierarchical Cubic Network," *Proc. 19th Int'l Conf. Parallel Processing*, vol. I, pp. 355–362, 1990.

A Comment on "A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors"

Ting-Lu Huang, *Member, IEEE*, and Chien-Hua Shann

Abstract—The circular list-based mutual exclusion algorithm proposed by Fu and Tzeng [1] is subject to a race condition that leads to a deadlock under subtle situations. An execution sequence evidences the race, and a modified version is provided. The performance of the original algorithm remains unchanged.

Index Terms—Critical sections, race conditions, deadlocks, atomic instructions.

1 INTRODUCTION

THE *Acquire_lock* procedure and the *Release_lock* procedure in an article [1] of this transaction are subject to a race condition which can lead to a disruption of the privilege consignation process under subtle situations.

The race condition is evidenced by tracing a few steps of a computation. The original algorithm is reproduced in Fig. 1, with a label for each statement. Suppose there are two processes that are competing for the root lock: the *header* and the *successor*. Each arrives at the top level with the value of `I->next` pointing to itself. Table 1 details a sequence of events that ends with process *successor* spinning forever, waiting for some other process to write FALSE to its *wait* bit, but no one will. This results in a deadlock, and no process may enter its critical section any more.

The cause of the problem can be explained by focusing on a process executing line A7 in any computation. Presumably, the action is meant to clear the bit for the later detection of a *wake-up* signal from someone else leaving its critical section. What can go wrong is that the signal may arrive earlier than the action of line A7 itself and, then, the action unknowingly overwrites the FALSE value which is meant to be detected as the signal. This is a typical race and could happen under subtle situations, like context switching of the *successor* process. Notice that the same type of race could also happen when a process executes line R8.

The problem is due to the lack of proper coordination between the waiting process and the signaling process associated with a *wait* bit. A modified version, which satisfies mutual exclusion and deadlock freedom, is given in Fig. 2. This is achieved simply by setting the *wait* bit TRUE soon enough in order to prevent the race condition as in Table 1. The modifications include:

1) substituting X2 for A2, X7 for A7, X8 for A8, X13 for A13, and
2) swapping line R7 and line R8.

Notice that the privilege consignation procedure for the global header without the *swap_and_compare* primitive also needs a modification to prevent the same problem. Specifically, those processes that need to wait for a *wake-up* signal are subject to the same race condition. A similar modification is suggested.

• *The authors are with the National Chiao Tung University, Department of Computer Science and Information Engineering, 1001 Ta-Hsueh Road, Hsin-Chu, Taiwan 30050, Republic of China.*
E-mail: {tlhuang, chshan}@csie.nctu.edu.tw.

Manuscript received 26 Aug. 1997.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105551.

```
A0    Acquire_Lock(Lock L, node *I) {
A1      int level;
A2      I->wait = FALSE;
A3      I->next = my_proc_id;
A4      for (level =1; level <= root_level; level++) {
A5        I->next = fetch_and_store (L[level], I-
>next);
A6        if (I->next != nil) {
A7          I->wait = TRUE;
A8          break;
A9        }
A10       if (level < root_level)
A11         I->next = fetch_and_store (L[level],
nil);
A12     }
A13     while (I->wait);
A14   }

R0    Release_lock(lock L, node *I) {
R1      if (I->next == nil) {
R2        while (1) {
R3          I->next = my_proc_id;
R4          swap_and_compare(L[level], I->next,
nil);
R5          if (I->next == my_proc_id)
R6            return;
R7          I->next->wait = FALSE;
R8          I->wait = TRUE;
R9          while (I->wait);
R10       }
R11     }
R12     I->next->wait = FALSE;
R13   }
```

Fig. 1. Acquire and release procedures of the circular list-based scheme given in [1].

```
A0    New_Acquire_Lock(Lock L, node *I) {
A1      int level;
X2      I->wait = TRUE;
A3      I->next = my_proc_id;
A4      for (level =1; level <= root_level; level++) {
A5        I->next = fetch_and_store (L[level], I-
>next);
A6        if (I->next != nil) {
X7          while (I->wait);
X8          return;
A9        }
A10       if (level < root_level)
A11         I->next = fetch_and_store (L[level],
nil);
A12     }
X13       /* empty line */
A14   }

R0    New_Release_lock(lock L, node *I) {
R1      if (I->next == nil) {
R2        while (1) {
R3          I->next = my_proc_id;
R4          swap_and_compare(L[level], I->next,
nil);
R5          if (I->next == my_proc_id)
R6            return;
R8*         I->wait = TRUE; /* swap line R7 and
line R8 */
R7*         I->next->wait = FALSE;
R9          while (I->wait);
R10       }
R11     }
R12     I->next->wait = FALSE;
R13   }
```

Fig. 2. Modified procedures of the circular list-based scheme.

TABLE 1
A SEQUENCE OF EVENTS THAT LEADS TO A DEADLOCK

process	actions	process state afterwards	description of the events

header	A5	`I->next=nil`
		`level=root_level`
		`L[root_level]=`*header*
		`I->wait=FALSE`
			I am the global header.

successor	A5	`I->next=`*header*
		`level=root_level`
		`L[root_level]=`*successor*
		`I->wait=FALSE`
			I should wake up *header* when I am done.

header	A6	`I->next=nil`	The test result is negative.
header	A10, A13, A14	`I->wait=FALSE`	I don't have to spin. Enter critical section.
header	R1	`I->next=nil`	I am the global header.
header	R2,R3	`I->next=`*header*	Prepare data for swapping.
header	R4	`I->next=`*successor* `L[root_level]=`*header*	Swap `L[root_level]` and `I->next`.
header	R5	`I->next=`*successor*	The test result is negative.
header	R7	`I->next->wait=FALSE`	Since `I->next->wait` points to the same bit as *successor's* `I->wait` does, the action is meant to wake up process *successor*.
successor	A6	`I->next=`*header*	The test result is positive.
successor	A7	`I->wait=TRUE`	I overwrote the FALSE value that process *header* had written as a wake-up signal.
successor	A8	`I->wait=TRUE`	Will go to A13.
successor	A13	`I->wait=TRUE`	I am block here since no process will write a FALSE value to `I->wait`.

Since the modifications involve only rearranging the order of a few statements, neither an extra computation step nor extra network traffic is incurred by such changes. As a result, the performance of the original algorithm remains unchanged.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Council of Taiwan under Grant NSC 86-2213-E-009-092. The authors would like to thank Prof. Ten H. Lai and the anonymous referees for their valuable comments and suggestions.

REFERENCES

[1] S.S. Fu and N.-F. Tzeng, "A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 6, pp. 628-639, June 1997.