A DEADLOCK ABSENCE PROOF TECHNIQUE APPLIED TO A MULTIPLE COPY CONSISTENCY PROTOCOL*

by

Laura M. Haas
K. M. Chandy
J. Misra

TR-149 July 1980

* Work supported by NSF Grant MCS77-09812.

LIST OF FIGURES

Abstract

This paper proposes a general method for proving absence of deadlock in distributed networks of communicating processes, where messages are the means of communication. This method is simple and, in particular, easy to use. We demonstrate these properties by applying the method to one solution to an important problem in distributed database systems: Ellis' algorithm for maintaining consistency in multiple copies of databases.

Index Terms

Absence of deadlock, proof method, distributed systems, deadlock absence proofs, distributed databases, multiple copy consistency problem, message communication systems.

1. INTRODUCTION

It is important to prove absence of deadlock in algorithms for distributed systems. This paper proposes a very simple method for deadlock absence proofs in general, and applies the method to an ubiquitous problem in distributed databases: the multiple copy consistency problem.

Ellis [1], Thomas [2], Silberschatz and Kedem [3] and others have proposed algorithms in the data base area, and have proven absence of deadlock for their algorithms. Our goal is to present a general method for proving deadlock absence in message communication systems and to demonstrate the ease of using this method by analyzing one example. We have chosen the multiple copy consistency algorithm because of the fundamental role that the algorithm plays in distributed data bases. Our approach is applicable to a variety of protocols.

In section 2 we discuss our model of a network of processes. We present the multiple copy consistency problem in section 3 and introduce Ellis' elegant solution [1] to the problem. Theorems regarding process waiting behavior are presented in section 4. Section 5 introduces the concept of priority which is central to our deadlock-absence proofs. The proof of absence of deadlock and starvation is found in section 6. The conclusion is presented in section 7.

2. NETWORK MODEL

A network is a finite collection of processes which communicate with one another exclusively through messages. Messages are transmitted along communication lines. A line is directed from one process to one other process. Each line has a unique label. There may be an arbitrary (finite) number of lines between a pair of processes. Consider a line (with label) e directed from a process h_i to a process h_j. Line e is said to be incident on h_i and h_j. Process h_i is said to wait on e if it is waiting to send a message along line e. Similarly, h_j is said to wait on e if it is waiting to receive along e. A process can only wait on a line incident on it. A message may be transmitted along a line only if the two processes at both ends of the line are both waiting on the line.

There is no parallelism within a process: it is impossible to execute two statements within a process simultaneously. However, a process h may wait in parallel on an arbitrary set of lines E incident on h. Note that a message can be transmitted along only one of the lines in E, at a time. Thus, though a process may wait in parallel on many lines, messages must be transmitted to and from the process serially.

We shall ignore terminated and unborn processes in this paper. For a more complete discussion regarding these issues, see [4].

A process is in one of two states: executable or blocked. A process is said to be executable if some statement in the process, other than a message transmission statement, can be executed. Informally speaking, an executable process is one on which processing can proceed without the cooperation of other processes. A process is said to be blocked if it is not executable. Since we assume (in this paper) that processes do not

terminate, a blocked process must be waiting on a non-empty set of lines. A blocked process h waiting only on a set of lines E, must continue waiting on all lines in E until a message is transmitted along at least one line in E. Informally, a blocked process is one in which processing can proceed only with cooperation from other processes; in other words processing can proceed only after a message transmission.

We are not concerned here with the internal details of a process. Hence, we assume that the length of time an executable process remains continuously executable is finite. In other words we assume that all statements and loops, which do not involve message communication terminate.

Deadlock

A non-empty set of blocked processes H in a network N is said to be deadlocked at some stage in computation, if, at that stage, for any h_i in H, if h_i is waiting on some line e, and e is incident on h_j, then h_j is also in H and h_j is not waiting on e. It follows that all processes in H will be permanently blocked.

Examples of Deadlock

Networks are represented as labeled directed graphs with processes as vertices and lines as edges. The letter W at the junction of a process h and a line e indicates that h is waiting on e; N indicates not waiting. The letter "X" by a process h indicates that h is executable; "B" indicates blocked.

Processes 1, 2 and 3 are deadlocked in figure 1. They are not deadlocked in figure 2 because processes 1 and 2 are waiting on line c. The processes are not deadlocked in figure 3 because, though process 2 is blocked, it is waiting on line d, and d is incident on an executable process (viz process 3).

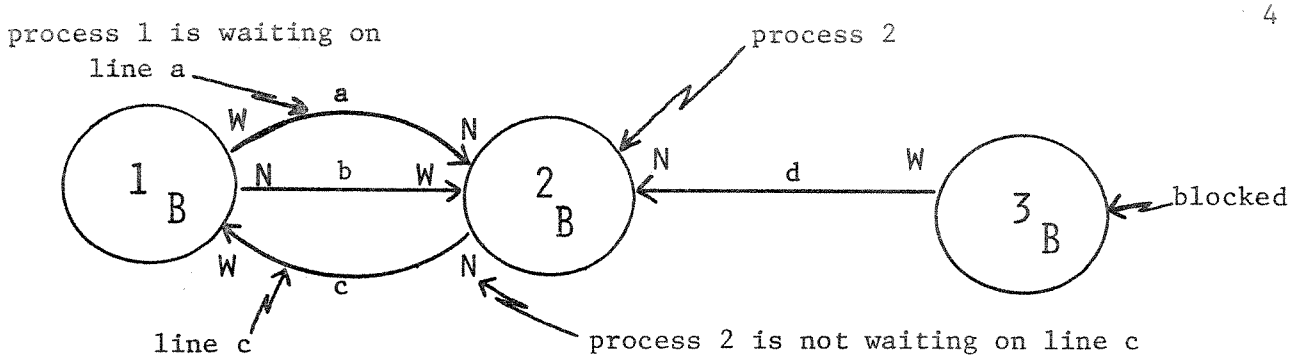


Figure 1. An example of deadlock

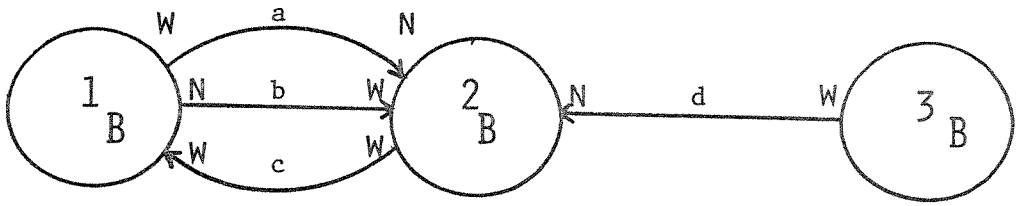


Figure 2. Not deadlocked

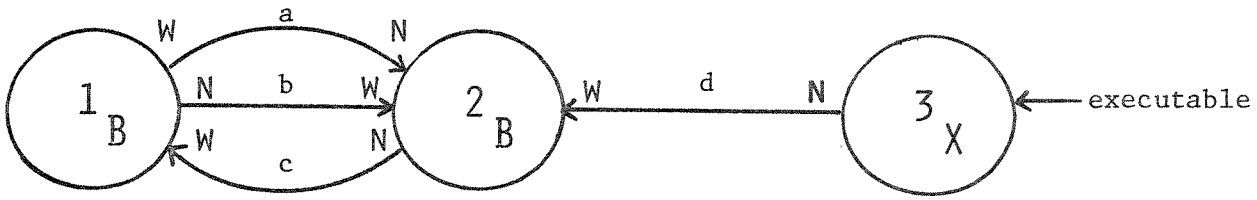


Figure 3. Not deadlocked

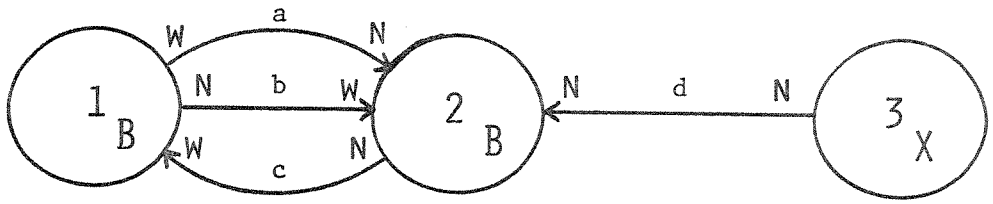


Figure 4. Processes 1 and 2 are deadlocked though 3 is executable

We cannot determine whether process 3 in figure 2 will ever become executable without proving properties about the internals of process 2. For example, processes 1 and 2 could send messages to one another, yet process 2 may never wait on the line d connected to process 3. Thus, though absence of deadlock does imply that at least one process can proceed, it does not imply that no process will be permanently blocked.

A network N is said to be deadlock-free at some stage of the computation if no set of processes H in N is deadlocked at that stage. N is deadlock-free if it is deadlock-free at every stage in every computation sequence.

A process P is said to be starved on a line e incident on it, if P is permanently waiting on e or if P waits on e infinitely often and no message is transmitted along e. P may be starved on e and not be deadlocked because P may be communicating along other lines. To prove absence of starvation we shall assume a fair-scheduler which determines the line along which communication will next take place. Two processes cannot both wait permanently on the same line if scheduling is fair. Fair scheduling also implies that two processes cannot both wait simultaneously on the same line infinitely often.

3. OVERVIEW OF ELLIS' ALGORITHM

A brief overview of the algorithm, is presented now. For simplicity in exposition we initially consider a simple network consisting of only 2 (two) users; the general network is considered later.

The simple network (figure 5) consists of 2 user processes (UPs), 2 control processes (CPs) and one data base process (DBP). The UPs represent users who request updates to control processes and wait for replies . The DBP represents the data base. The DBP gets update requests from CPs, carries out the update and responds with update-done messages. The CP is responsible for correctly managing an update. (The general network will consist of an arbitrary number of processes of each type.)

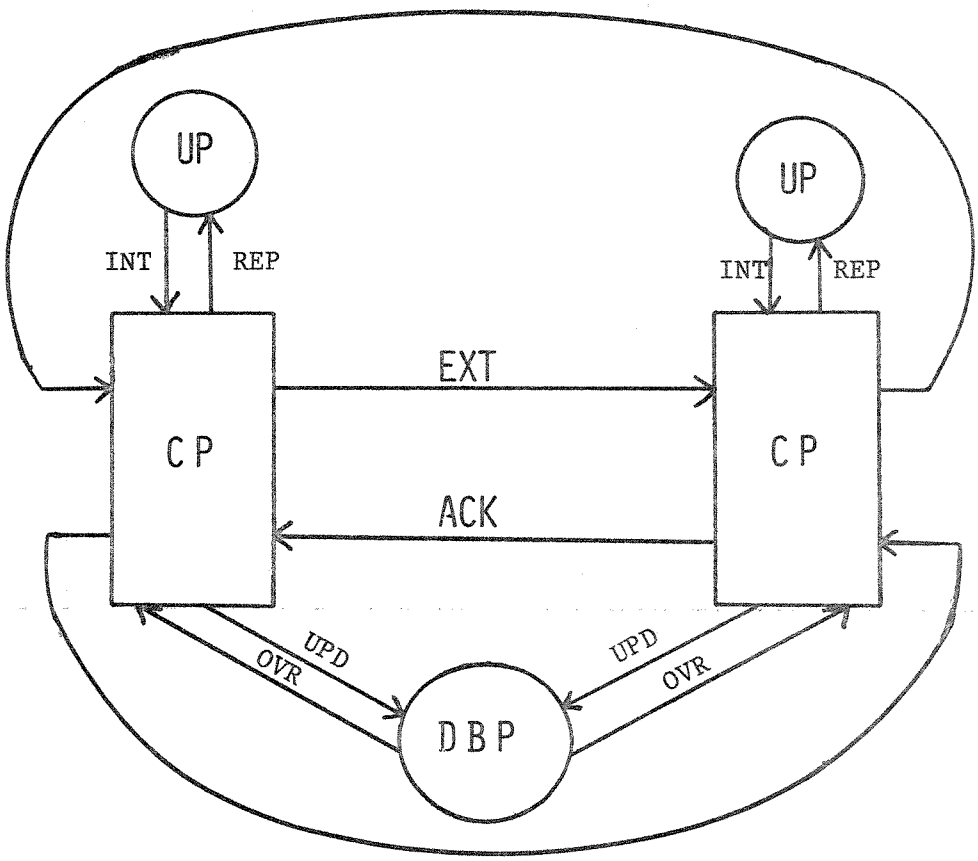


Figure 5. A simple network with 2 control processes and one data base process

User process

The operation of a user process is straightforward. Each UP is connected to one CP which is responsible for managing all of the UP's requests. Each UP repeats the following operating cycle. Initially a UP is in a "think"or executing state. After some time, the UP waits to send an update request (called an internal request by Ellis) to the control process which manages its request. After sending the message the UP waits to receive a reply from the CP; the reply may either be a DONE (request completed) or REJECT (request denied) message. (A UP whose request is denied may try to repeat the message.) The cycle of operation of a UP is shown in figure 6 .

Figure 6 . Cycle of operation of a user process

The time spent in an execution step is arbitrary but finite. Execution steps do not play a part in deadlock; hence it is convenient to focus attention on those steps in its cycle of operation in which a process may become blocked. In other words, we shall only consider those steps in the cycle in which a process is waiting for some

event. A diagram showing the sequence of points at which a process waits to communicate is called a waiting-cycle diagram. The waiting-cycle diagram for a UP is shown in figure 7.

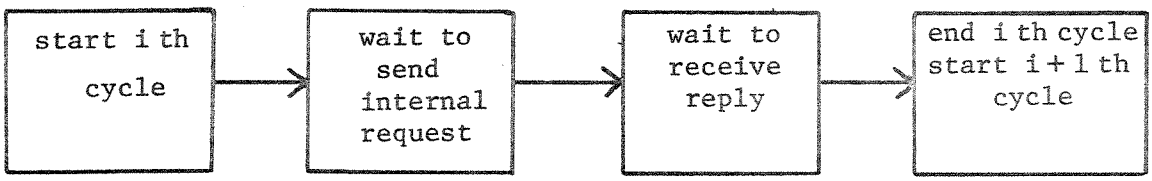


Figure 7. Waiting cycle diagram for a user process

Data Base Process

A data base process's operation is also simple. Its function is to receive updates from control processes, implement the updates in the order received and send update completed messages to control processes. A DBP begins by waiting in parallel to receive update

messages from all control processes. When it receives an update message from any control process, say CP m, the data base process ceases to wait for further update messages, implements the requested update on its copy of the data base and waits to send an update-over message to CP m. After sending the update done message to CP m, the DBP repeats its cycle, once again waiting in parallel to receive update messages from control processes. The waiting-cycle diagram of a DBP is shown in figure 8.

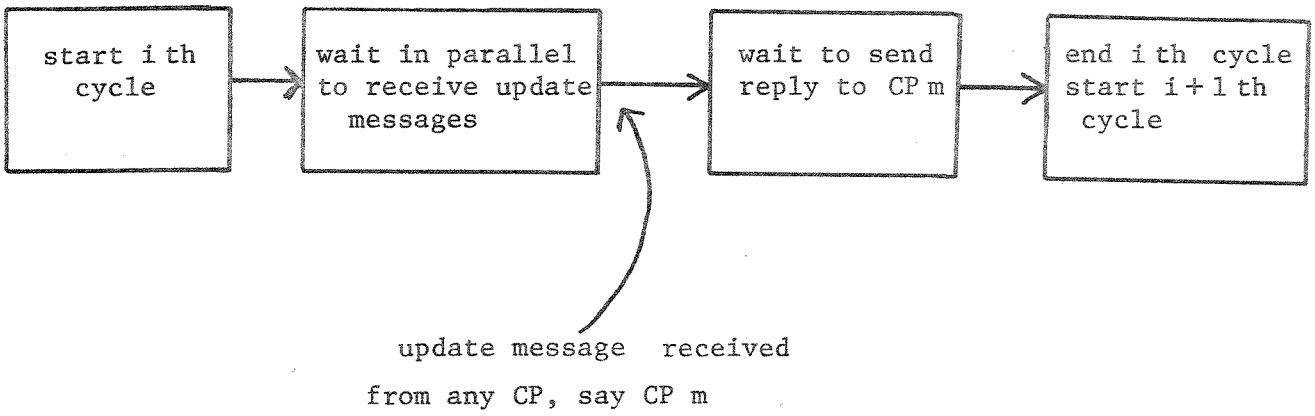


Figure 8. Waiting cycle diagram of a data base process

Control Process

A CP's behavior is described by two waiting cycles: the basic waiting cycle and the external response cycle. The basic waiting cycle describes how a CP handles requests received from its user process. The external response cycle describes how a CP handles requests made by the other CP.

The basic waiting cycle of a control process (figure 9) say CP 1.

Step 1 Initially, CP 1 waits for a request from the user process; at this point CP 1 is not waiting on any other line.

Step 2 On receiving the request, CP1 ceases to wait to communicate with the user process and begins to wait to send an "external request" message to the other CP, i.e. CP 2. The external request message asks CP 2 whether the internal request being processed by CP 1 conflicts with the internal request (if any) being processed by CP 2. If CP 2 responds with a positive ACK there is no conflict. If CP 2 responds with a negative ACK, there is a conflict.

Step 3 On sending the external request, CP 1 ceases to wait to send to CP 2 and begins to wait to receive an ACK from CP 2. If the ACK received is positive, CP 1 stops waiting for ACKs and goes to step 4 of the basic waiting cycle, else to step 5.

Step 4 If the ACK received is positive then

begin: (updating)

Step 4a CP1 begins to wait to send an update message to the data base processor.

Step 4b On sending the message, CP 1 ceases to wait to send to the DBP and begins to wait for an update-over reply

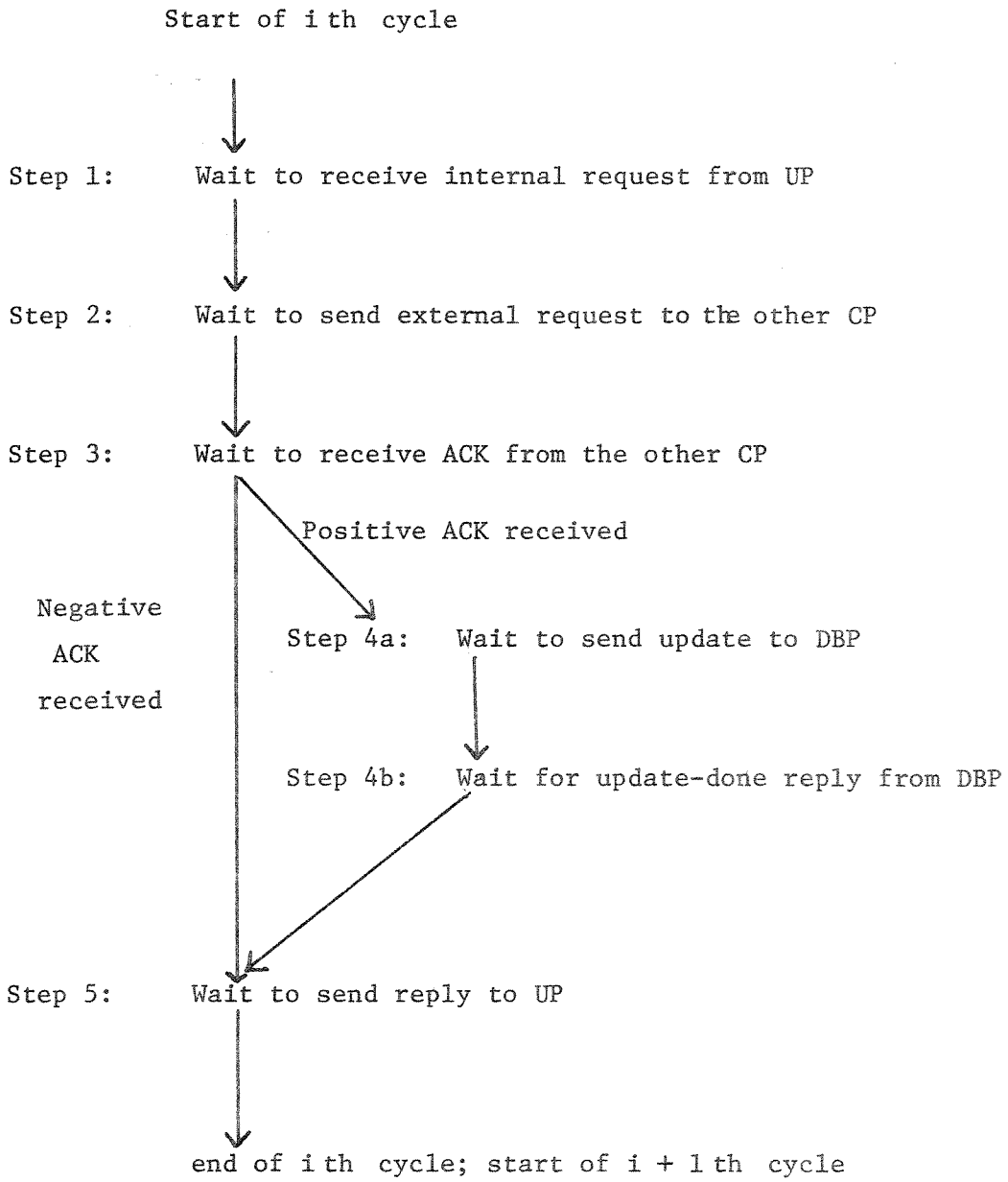


Figure 9. Basic waiting cycle of a CP

from the DBP. On receiving the reply, CP 1 ceases to wait for messages from the DBP and goes to step 5.

end: (updating)

Step 5 CP 1 waits to send a reply to the user process. The reply is a "DONE" message if the ACK that CP 1 received from CP 2 was positive; otherwise the reply is a "REJECT" message.

The cycle is complete on sending the reply.

We next describe the external response cycle.

External response cycle of a CP (figure 10)

In parallel with the basic waiting cycle, a CP also executes an external response cycle in which it waits to receive external requests from the other CP, determines if there is a conflict with the internal request (if any) that it is processing, and replies with a positive or negative ACK. If a CP is waiting to communicate with its UP, (i.e. if the CP is in steps 1 or 5 of the basic waiting cycle), then the CP is not currently processing an internal request. Ellis points out that there is an advantage in efficiency in a CP postponing sending an ACK until the CP is in steps 1 or 5 of the basic operating cycle, because in these states the CP can always send a positive ACK, since it is not processing an internal request in these states. However, if both CPs always postpone sending ACKs until they are in steps 1 or 5 of their basic operating cycles, deadlock could arise with each CP waiting for an ACK from the other. Deadlock can be avoided if a CP postpones sending ACKs only while it is updating (i.e. waiting to communicate with the data base process: step 4 of the basic waiting cycle).

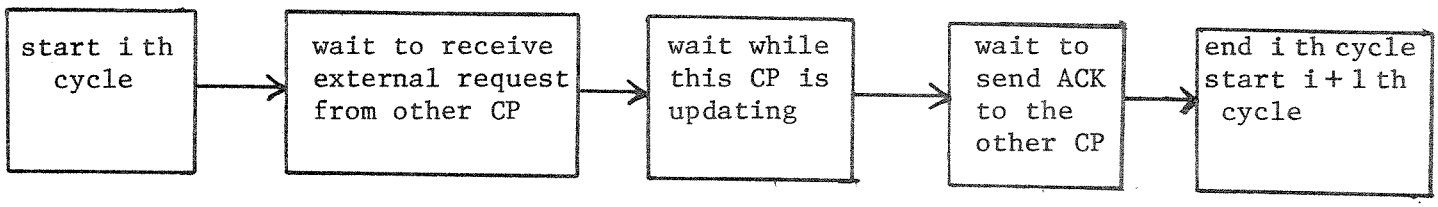


Figure 10. External response cycle of a CP

The basic waiting cycle and the response cycle of a CP proceed in parallel with no interdependence except that the transition into the "wait-to-send ACK" step in the response cycle cannot be carried out while the CP is in step 4 of the basic waiting cycle.

The general network

Now consider the case where there are an arbitrary (positive) number of user, data base and control processes. Let there be R UPs, K DBPs and M CPs. Each UP communicates with one CP which manages its requests.

The operation of a UP and a DBP are as described before. The basic operating cycle of a CP is modified in the obvious way to handle a multiplicity of processes. In step 1 the CP waits in parallel to receive an internal request from any one of its UPs. The sending of external requests to CPs and the receiving of ACKs from CPs (steps 2 and 3) are done in parallel for all CPs. The positive ACK branch is taken only if positive ACKs have been received from all CPs. The negative ACK branch

is taken only if ACKs have been received from all CPs but not all the ACKs received are positive.

Since there is an arbitrary number of data base processors, a CP must now ensure that updates are carried out by all DBPs. Hence steps 4a and 4b are done in parallel for all DBPs, and the final DONE reply to the UP is sent only after receiving update-done messages from all DBPs.

The CP's response cycle is not modified; however, each CP must execute several response cycles in parallel, one response cycle for every other CP. The names given to the lines in the network are shown in Table I.

LINE NAME	FROM	TO	MESSAGE	MEANING OF MESSAGE
INT_{rm}	UP r	CP m	Internal Request	Request to do update
REP_{mr}	CP m	UP r	Reply to internal request	Internal request is done or rejected
EXT_{mn}	CP m	CP n	External request	Can I implement this update?
ACK_{nm}	CP n	CP m	Acknowledgement to external request	Yes (ACK+) or No (ACK-)
UPD_{mk}	CP m	DBP k	Update request	Carry out update
OVR_{km}	DBP k	CP m	Over	Update has been implemented

Table 1. Names of lines in the network

4. PROCESS WAITING BEHAVIOR

Definitions

X < Y

Let X and Y be two distinct lines in the network. We use the notation X < Y to denote the condition that a message was transmitted on line Y after the last message transmitted on line X. At any given time during the execution of the network, if X and Y are lines incident on the same process P either X < Y or Y < X, because a process cannot communicate along two lines simultaneously.

WAIT[P,X]

Let X be a line incident on a process P. We denote the condition that P is waiting on line X by WAIT[P,X].

BLOCKED[P]

We denote the condition that process P is blocked by BLOCKED[P].

Initial Conditions

Initial conditions are specified as a partial ordering on all lines in the network. In Ellis' protocol, and many other protocols, the partial ordering specifying the initial condition may be derived from the conditions obtaining after complete traversals of the waiting cycles. For example, the initial conditions for Ellis' algorithm, using waiting cycles in figures 7 - 10, are displayed in table 2. These conditions merely state that at the start of the protocol we assume the convention that earlier internal requests were followed by external requests, ACKs, updates, update-over messages and replies to the user, in that order.

$INT_{rm} < EXT_{mn}$	all r, n, m
$EXT_{mn} < ACK_{nm}$	all m, n
$ACK_{nm} < UPD_{mk}$	all n, m, k
$UPD_{mk} < OVR_{km}$	all m, k
$OVR_{km} < REP_{mr}$	all r, m, k

Table 2. Initial conditions for the protocol

X precedes Y in P or P[X → Y]

Let X and Y be lines incident on a process P. We shall say that X precedes Y in P, denoted by P[X → Y] if

$$WAIT[P,Y] \rightarrow Y < X$$

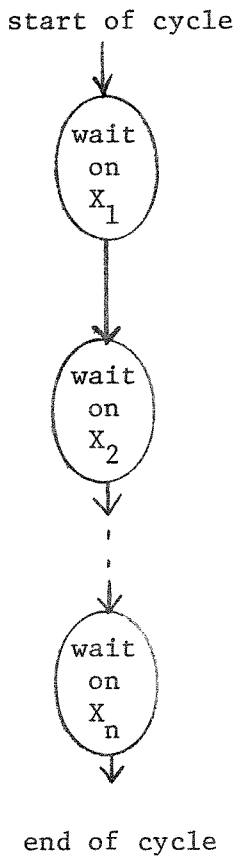
Example If P must communicate along X after a communication along Y and before P next waits on Y (and if this behavior is consistent with the initial condition) then P[X → Y], because if P is waiting on Y then it must have communicated along X after it last communicated along Y.

In figure 11, we have $P[X_i \rightarrow X_j]$ for any $i \neq j$, because if P were waiting on X_j then P must have communicated on every X_i, $i \neq j$, after it last communicated on X_j.

X Results in Y in P, or P[X ⇒ Y]

Let X and Y be lines incident on a process P. We shall say that X results in Y in P, denoted by P[X ⇒ Y] if

$$Y < X \text{ and } BLOCKED[P] \rightarrow WAIT[P,Y]$$



Initial condition: $X_1 < X_2 < .. < X_n$

Figure 11. Example illustrating $P[X \rightarrow Y]$

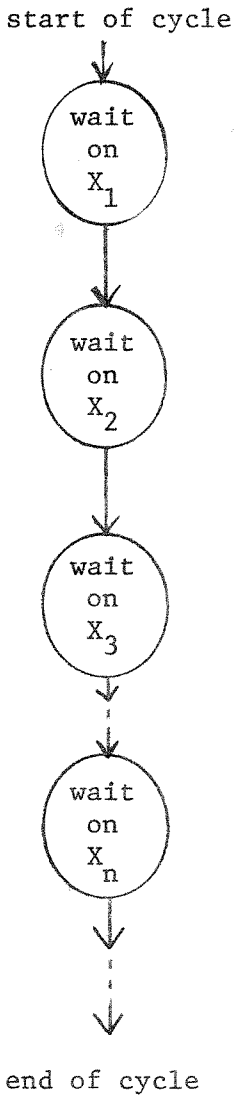
Example In figure 12, $P[X_i \Rightarrow X_{i+1}]$, $i = 1,..,n-1$ because after a communication along line X_i, P waits on line X_{i+1}.

P drives X

Let X be a line incident on processes P and P'. P is said to drive X if

$$BLOCKED[P'] \text{ and } WAIT[P,X] \rightarrow WAIT[P',X] .$$

In other words, if P drives X, it is impossible for P to be waiting on X while P' is blocked and not waiting on X.



Initial condition $X_1 < X_2 < .. < X_n$

Figure 12. Waiting diagram for which $P[X_i \Rightarrow X_{i+1}]$

Lemma

Let X and Y be lines each of which is incident on processes P_1 and P_2. Then

$$P_1\ [X \Rightarrow Y] \text{ and } P_2\ [X \rightarrow Y] \rightarrow P_2 \text{ drives } Y$$

Proof

WAIT $[P_2,Y] \rightarrow Y < X$

BLOCKED$[P_1]$ and $Y < X \rightarrow$ WAIT$[P_1,Y]$

Hence BLOCKED $[P_1]$ and WAIT $[P_2,Y] \rightarrow$ WAIT $[P_1,Y]$

Examples

It follows directly from the lemma that a CP drives all communication (replies and requests) with UPs:

$UP[INT_{rm} \Rightarrow REP_{mr}]$ and $CP_m[INT_{rm} \rightarrow REP_{mr}] \rightarrow CP_m$ drives REP_{mr}

$UP[REP_{mr} \Rightarrow INT_{rm}]$ and $CP_m[REP_{mr} \Rightarrow INT_{rm}] \rightarrow$

$UP_r[REP_{mr} \Rightarrow INT_{rm}]$ and $CP_m[REP_{mr} \rightarrow INT_{rm}] \rightarrow CP_m$ drives INT_{rm}

Note that UP_r also drives INT_{rm}.

Likewise, a CP also drives outgoing external requests (EXTs), outgoing ACKs, and update-over replies (OVRs) from DBPs. OVRs are also driven by DBPs.

Mutually exclusive waits. MUTEX $[P_1,X; P_2,Y]$

Let X and Y be lines, each of which is incident on processes P_1 and P_2. P_1 waiting on X and P_2 waiting on Y are said to be mutually exclusive, denoted by MUTEX$[P_1,X; P_2,Y]$, if WAIT $[P_1,X] \rightarrow$ NOT WAIT $[P_2,Y]$.

Lemma

$P_1[X \rightarrow Y]$ and $P_2[Y \rightarrow X] \rightarrow$ MUTEX $[P_1,Y; P_2,X]$

Proof

WAIT $[P_1,Y] \rightarrow Y < X$ (since $P_1[X \rightarrow Y]$)

WAIT $[P_2,X] \rightarrow X < Y$ (since $P_2[Y \rightarrow X]$)

Hence WAIT $[P_1,Y] \rightarrow$ NOT WAIT $[P_2,X]$

5. PRIORITIES

It is helpful in deadlock absence proofs to assign certain real numbers, called priorities to edges of the network. Let $g_N(e,S)$ denote the priority assigned to an edge e in network N, when the computational state of the network is S. Note that the priority assigned to an edge may change when the computational state of the network changes. g_N ( )

is called a priority function. It has been shown [4] that a necessary and sufficient condition for absence of deadlock is the existence of a priority function satisfying certain properties called proper priority conditions; we shall prove absence of deadlock in Ellis' algorithm by presenting a priority function which satisfies these properties. It should be emphasized that priority functions are concocted by the program prover exclusively for proof purposes.

In the following, we shall drop the subscript N when talking about parameters dealing with network N.

We define the propriety condition for a priority function g( ), process P, and line L incident on P as follows: The propriety condition H(g,P,L) holds if and only if for every computational state S in which P is blocked and not waiting on L there exists a line L' incident on P with $g(L',S) < g(L,S)$.

We use the following short-form to define the propriety condition:

H(g,P,L) holds if and only if

NOT WAIT [P,L] and BLOCKED[P] → $g(L,S) > g(L',S)$ for some L' incident on P.

A priority function g for a network N is said to be proper if the propriety condition H(g,P,L) holds for every process P in the network and every line L incident on P.

Example

Consider a network consisting of two processes A and B (figure 13). Process A initially waits to send out the number 0 (zero) to process B. After sending any message to B, A waits to receive a message from B. Upon receiving a message (assumed to be an integer), A adds 1 (one) to the integer received and waits to output the sum to B. B's operation

is identical to A's except that B begins by waiting to receive a message from A. Figure 14 shows the operation of processes A and B.

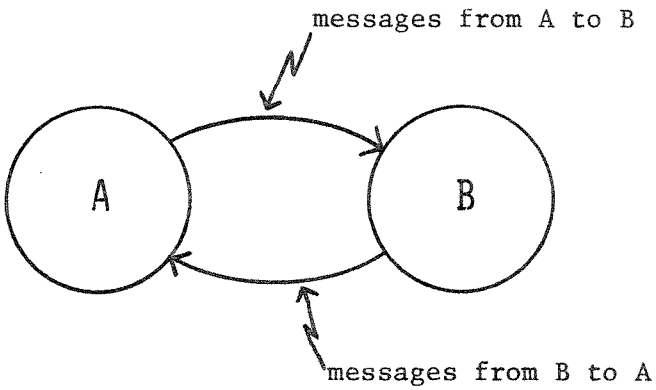


Figure 13. The network consisting of processes A and B.

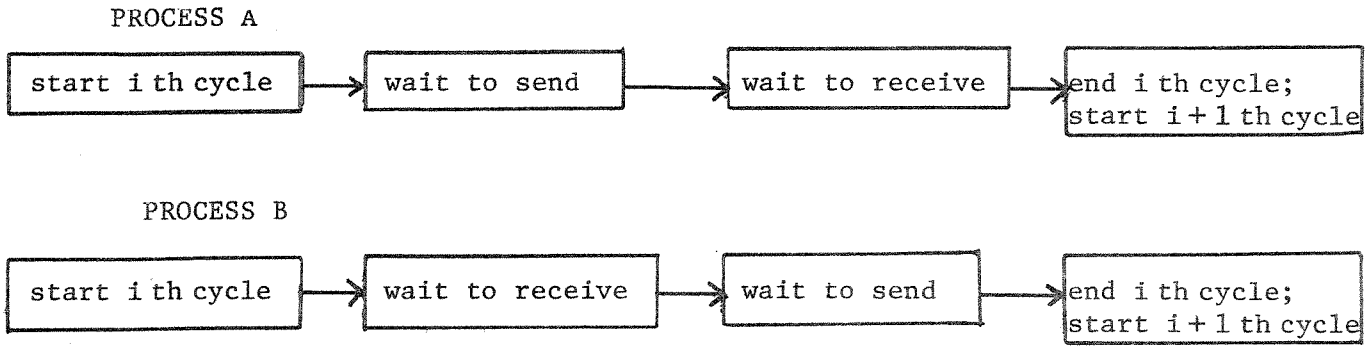


Figure 14. Operation of processes A and B.

We define a priority function $g_N(e,S)$ as follows. If process A is waiting to receive, then the line from B to A is assigned a priority of 0 (zero), and the line from A to B is assigned a priority of 1 (one) else the line from B to A is assigned a priority of 1 (one) and the line from A to B is assigned a priority of 0. It is easy to see that the priority function is proper.

Theorem 1

A network N is deadlock free if and only if there exists a proper priority function g( ) for N

Proof See [4].

Note: The theorem implies that there does not exist a set of proper priorities for the stages shown in figures 1 and 4. The reader may find it instructive to verify this fact for himself.

We have found that it is easy to derive a proper priority function from a deadlock-free algorithm, as we shall show for Ellis' algorithm [1]. Our method has also been applied to other important problems [5].

6. ABSENCE OF DEADLOCK AND STARVATION

We next present a priority function for Ellis' algorithm and then show that it is proper; only then shall we attempt to motivate our choice of the function. It is much easier to understand the reasoning underlying the choice of a priority function after going through one proof demonstrating that a priority function is proper.

The priority function is described in table 3. For each line and for all possible conditions of the processes incident on the line, the table specifies the priority of the line. For example, the first row states that the priority of the line from UP r to CP m is 1 (one) while CP m is waiting on that line.

Proof that the priority function is proper

We shall consider every process P, every line L incident on P and show that the propriety condition H(g,P,L) holds.

STEP 1: Propriety condition for CP m

The lines mentioned in the basic operating cycle of a CP are lines between CPs and UPs, CPs and DBPs, outgoing external requests and incoming

Row No.	Line	Condition	Priority
1	INT_{rm}	CP m waits	1
2		CP m does not wait and UP r waits	4
3		CP m does not wait and UP r does not wait	6
4	REP_{mr}	CP m waits	1
5		CP m does not wait	5
6	ACK_{nm}; EXT_{mn}	CP m waits	3
7		CP m does not wait	6
8	OVR_{km}	CP m waits	1
9		CP m does not wait	8
10	UPD_{mk}	CP m waits	2
11		CP m does not wait	7

Table 3. The priority function

ACKs; the lines not mentioned in the basic operating cycle (and mentioned in the external response cycle) are incoming external requests (EXTs) and outgoing ACKs.

L is a line in the basic operating cycle

In any given computational state S, a CP is at some point in its basic operating cycle and hence is waiting on some line L^* mentioned in the basic operating cycle if it is blocked. From table 3

$$g(L^*,S) \leq 3$$

For any line L mentioned in the basic operating cycle

NOT WAIT [CP,L] $\rightarrow g(L,S) \geq 4$ (table 3)

Hence, $g(L,S) > g(L^*,S) \rightarrow H(g,P,L)$

L is the outgoing ACK (ACK_{mn}) from CP m to CP n

BLOCKED [CP m] and NOT WAIT [CP m, ACK_{mn}]

$\rightarrow$ WAIT [CP m, EXT_{nm}] or WAIT [CP m, UPD_{mk}] or WAIT [CP m, OVR_{km}]

for some k (see response cycle - figure 10).

Case 1 WAIT [CP m, EXT_{nm}] $\rightarrow$ NOT WAIT [CP n, ACK_{mn}] (because MUTEX)

$\rightarrow g(ACK_{mn},S) = 6$ (see table)

Hence $g(ACK_{mn},S) > g(L^*,S) \rightarrow H(g,P,L)$

Case 2 WAIT [CP m, UPD_{mk}] $\rightarrow g(UPD_{mk},S) = 2$ (see table)

$g(ACK_{mn},S) \geq 3$ (see table)

Hence $g(ACK_{mn},S) > g(UPD_{mk},S) \rightarrow H(g,P,L)$

Case 3 Same argument holds for WAIT[CP m, OVR_{km}]

L is the incoming external request (EXT_{nm}) from CP n to CP m

NOT WAIT [CP m, EXT_{nm}] $\rightarrow$ NOT WAIT [CP n, EXT_{nm}] (since CP n drives EXT_{nm})

$\rightarrow g(EXT_{nm},S) = 6$ (see table)

$\rightarrow g(EXT_{nm},S) > g(L^*,S) \rightarrow H(g,P,L)$

STEP 2: Propriety condition for UP r

L is the internal request line(INT_{rm}) from UP r to CP m.

NOT WAIT [UP r, INT_{rm}] $\rightarrow$ NOT WAIT [CP m, INT_{rm}] (since CP m drives INT_{rm})

$g(INT_{rm},S) = 6$ (see table)

$\rightarrow$ $g(INT_{rm},S) > g(REP_{mr},S)$ (see table)

$\rightarrow$ $H(g,P,L)$

L is the reply line (REP_{mr}) from CP m to UP r.

NOT WAIT [UP r, REP_{mr}] $\rightarrow$ NOT WAIT [CP m, REP_{mr}] (since CP m drives REP_{mr})

$g(REP_{mr},S) = 5$ (see table)

BLOCKED[UP r] and NOT WAIT [UP r, REP_{mr}]

$\rightarrow$ WAIT [UP r, INT_{rm}] (from UP r operating cycle)

$\rightarrow$ $g(INT_{rm},S) \leq 4$ (see table)

$\rightarrow$ $g(REP_{mr},S) > g(INT_{rm},S) \rightarrow H(g,P,L)$

STEP 3: Propriety condition for DBP k

L is the update-over line (OVR_{km}) from DBP k to CP m.

NOT WAIT [DBP k, OVR_{km}] $\rightarrow$ NOT WAIT[CP m, OVR_{km}] (since CP m drives OVR_{km})

$\rightarrow$ $g(OVR_{km},S) = 8$ (see table)

$\rightarrow$ $g(OVR_{km},S) > g(UPD_{mk},S)$ (see table)

$\rightarrow$ $H(g,P,L)$

L is the update request line (UPD_{mk}) from CP m to DBP k.

NOT WAIT[DBP k, UPD_{mk}] $\rightarrow$ WAIT [DBP K, OVR_{kn}] for some CP n (see DBP operating cycle)

$\rightarrow$ WAIT[CP n, OVR_{kn}] (since DBP k drives CP n)

$\rightarrow$ $g(OVR_{kn},S) = 1$ (see table)

$g(UPD_{mk},S) \geq 2$ (see table)

Hence $g(UPD_{mk},S) > g(OVR_{kn},S) \rightarrow H(g,P,L)$

Absence of starvation

1. No process can be waiting permanently on a line with priority of 1 (one), because the process at the other end of the line must also be waiting on the line since one is the lowest possible priority (and the priority function is proper).
2. We now show that no process can wait permanently on a line of priority 2 or lower. Line L can have priority 2 only if 2 is a line from a CP to a DBP and the CP is waiting on L. If the CP waits permanently on L then the DBP never waits on L. This implies that the DBP waits permanently on a message-over line which has a priority of 1 (from earlier arguments). This is impossible.
3. Similar arguments applied to lines with priorities 3, 4, 5 and 6, in sequence, show that no process can wait permanently on a line of priority 6 or lower. Hence user and control processes cannot be starved, because the maximum priority of any line on which a user or control process is waiting is 6.

Intuition behind the definition of the priority function

The priority function is defined by considering, oddly enough, the proof which must be made to show that the function is proper. Originally, a set of values is "guessed" by choosing one or more "important" processes, and assigning each line a low priority when one of these processes waits on it, and a high one otherwise. These values are then adjusted until a valid proof is obtained, by considering for each line L and process P, what line(s) P will wait on when not waiting on L, and assuring that the priority of L is higher than these lines when NOT WAIT[P,L].

For Ellis' algorithm, we started with the basic operating cycle of the control process because it was at the heart of the algorithm and assigned a priority of one (1) to a line on which the CP was waiting, and any arbitrarily high value otherwise.

Analysis of the UPs showed that the internal request line should have a lower priority than the reply line, when the CP was not waiting to communicate with the UP and the UP was waiting to send a request. Similarly, if the UP were not waiting to send, the priority of the UP sending line had to be higher than the reply line, and these arguments led to rows 2 , 3 and 5 of the priority table. The same arguments applied to the DBP resulted in rows 9, 10 and 11 of the table.

A CP is not waiting to send ACKs while it is updating (see external response cycle). Hence the priority of the ACK lines must be greater than the priority of the lines between a CP and a DBP, while the CP is waiting to update. This lead to row 6 of the table.

Finally, it should be emphasized that these priorities are not unique. The numbers themselves are not important; their relative values are.

7. CONCLUSION

The value of our approach is its simplicity. Applications to other protocols have also been straightforward.

REFERENCES

[1] Ellis, C. A. "A Robust Algorithm for Updating Duplicate Databases," 2nd Berkeley Conference on Distributed Computing, May 1977.

[2] Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," ACM TODS, Vol. 4:2, June 1977, pp. 180-209.

[3] Silberschatz, A. and Z. Kedem, "Consistency in Hierarchical Database Systems," Journal of the ACM, Vol. 27:1, January 1980, pp. 72-80.

[4] Chandy, K. M. and J. Misra, "Deadlock Absence Proofs for Networks of Communicating Processes," Information Processing Letters, Vol. 9:4, November 20, 1979, pp. 185-189.

[5] Haas, L. M., "Proving Temporal Properties of Distributed Systems," Ph.D. Dissertation, Computer Science Department, University of Texas, Austin, Texas 78712 (in preparation).