Data Base Contamination and Recovery

Murray Edelberg

Sperry Research Center
Sudbury, Massachusetts 01776

ABSTRACT

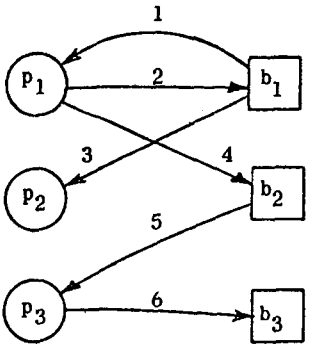
An approach to dealing with the contamination problem in the context of a simple model of a multi-process data base environment is described. We present an algorithm which, given a specification of a set of data transfers which are judged to have been possible contaminators, tracks the possible spread of contamination among processes and data blocks. This algorithm can be used as a diagnostic tool for recovery. Also, for environments in which it is feasible to rerun processes, we describe an algorithm which determines a recovery strategy. The strategy consists of processes to be rerun, blocks to be restored to a prior image, and appropriate prior images for these blocks.

I. INTRODUCTION

Bachman [1] observes that there are two basic challenges to the integrity of data in a computing environment in which multiple processes access and update shared data. One is interference, the other is contamination.

Interference occurs when two or more processes, each of which would be correct if running alone, interact to produce incorrect results. Contamination involves the propagation of undetected errors among processes and data. In this paper we are concerned exclusively with the contamination problem.

As a simple example, consider the figure below, in which the circles represent



processes, squares represent data blocks, and labeled directed lines represent data transfers which occur in the order suggested by the labels. If process p_1 transfers incorrect data to block b_2 via the transfer labeled 4, then transfers 5 and 6 may propagate the error, thereby affecting process p_3 and block b_3.

If errors are detected soon after they occur then opportunity for contamination is limited. In some applications it may be possible to detect errors committed

by a process before that process completes. In such cases it is possible to eliminate contamination entirely by locking all data items updated by a process until that process completes correctly. However, any error which escapes early detection can be propagated after completion of the offending process.

Recovery from contamination may involve processes and blocks other than those that have been contaminated. In the figure above, if a corrected version of process p_1 is to be rerun then it may be necessary to restore block b_1 to its image prior to transfer 2. Furthermore, if b_1 is restored then it may also be necessary to rerun process p_2.

In this paper we describe an approach to dealing with the contamination problem in the context of a simple model of a multi-process data base environment. We present an algorithm which, given a specification of a set of data transfers which are judged to have been possible contaminators, tracks the possible spread of contamination among processes and data blocks. This algorithm can be used as a diagnostic tool for recovery. Also, for environments in which it is feasible to rerun processes, we describe an algorithm which determines a recovery strategy. The strategy consists of processes to be rerun, blocks to be restored to a prior image, and appropriate prior images for these blocks.

II. THE MODEL

Our multi-process data base model consists of a set $P = \{p_1, p_2, \ldots, p_m\}$ of processes, a data base $B = \{b_1, b_2, \ldots, b_n\}$ partitioned into blocks b_i, $i = 1,2,\ldots,n$, and a sequence $T = (\tau_1, \tau_2, \ldots, \tau_k)$ of process/block data transfers.

At any given instant there is a subset of P which comprises the active processes. Once a process becomes active it remains so until completion, at which time it departs the system, perhaps to be replaced by another process. Execution of active processes is interleaved on one or more processors. Processes are independent; that is, there are no precedence constraints governing execution scheduling of active processes. Processes are considered indivisible for rollback purposes. No previous internal process state information is saved.

The data base B resides on one or more mass storage devices. Data base blocks are permanent subdivisions of B. All interactions between processes and the data base are handled by a data management system. A process issues commands to the data management system which in turn transfers information between that process and one or more data base blocks in response to each command.

There are two kinds of data transfers that can take place between a process p_i and a block b_j, one for each direction of data movement. We use the notation $\tau = (p_i,t,b_j)$ to represent a transfer τ from p_i to b_j initiated at time t, and (b_j,t,p_i) for a similar transfer in the reverse direction.

Transfers are indivisible; once begun they run to completion. At any given instant in time there can be at most one transfer in progress to or from a given block. The transfer sequence T reflects a lock-for-update mechanism. For each transfer (p_i,t,b_j) there is another transfer (b_j,t',p_i), where $t' < t$, such that no transfer involving b_j occurs in the interval $[t',t]$. The transfer sequence $T = (\tau_1,\tau_2,\ldots,\tau_k)$ is ordered by initiation time.

III. CONTAMINATION

We say that a transfer τ_r predates a transaction τ_s, written $\tau_r <\cdot\ \tau_s$, if there is a (not necessarily contiguous) subsequence $T' = (\tau'_1,\tau'_2,\ldots,\tau'_q)$ of T, where $\tau'_i = (x_i,t_i,y_i)$, $i = 1,2,\ldots,q$, and $q \geq 1$, such that:

(1) $\tau'_1 = \tau_r$ and $\tau'_q = \tau_s$

(2) $t_i < t_{i+1}$, for $1 \leq i < q$

(3) $y_i = x_{i+1}$, for $1 \leq i < q$.

If τ_r predates τ_s then we say that τ_s postdates τ_r and write $\tau_s \cdot> \tau_r$.†

† It can be shown that $<\cdot$ is a partial ordering relation on the set of transfers in T.

It should be clear that every transfer sequence $T' = (\tau'_1, \tau'_2, \ldots, \tau'_q)$ along which an error can propagate satisfies conditions (2) and (3) above. On the other hand, every sequence T' satisfying (2) and (3) is not necessarily a sequence along which an error can or will propagate. Consider two consecutive transfers τ'_i and τ'_{i+1} in a sequence T' satisfying (2) and (3). If $y_i, x_{i+1} \in P$ then τ'_{i+1} may overlap τ'_i in time or τ'_{i+1} may be functionally independent of τ'_i. If $y_i, x_{i+1} \in B$ then perhaps only a portion of block x_{i+1} is actually delivered to or utilized by process y_{i+1}.

We refer to a transfer τ_r which propagates an error as a <u>contaminator</u>, and adopt the following rule concerning the identification of contaminators:

> If τ_r is a contaminator and $\tau_r <\cdot \tau_s$
>
> then τ_s is a contaminator.

In other words, we assume that any transfer which postdates a contaminator is itself a contaminator. Although this assumption is not always valid, as the preceeding paragraph suggests, it allows us to identify contaminators without the need to interpret processes and leads to an efficient contamination tracking algorithm, at the possible cost of mistakenly including some non-contaminators.

Let S be a given set of <u>contaminator-seeds</u> drawn from T. Define $\overline{S}$ to be the set of transfers which postdate transfers in S. $\overline{S}$ includes S and all contaminators generated by S.

A process or block which participates in one or more transfers in $\overline{S}$ is said to be <u>contaminated</u>. A contaminated process p_i (or block b_j) is a <u>contaminant</u> if there exists a transfer $\tau = (p_i, t, b_j)$ (or (b_j, t, p_i)) in $\overline{S}$ such that τ is not predated by any other transfer in $\overline{S}$. In the following section we give a one-pass contamination tracking algorithm which, given T and S, identifies all contaminated/contaminant processes and blocks.

IV. CONTAMINATION TRACKING

Input to the algorithm consists of the transfer sequence $T = (\tau_1, \tau_2, \ldots, \tau_k)$ and a set S of contaminator-seeds. The algorithm uses three arrays: process-status [1:m], block-status [1:n] and block-time [1:n]. Initially,

$$\left.\begin{array}{l} \text{process-status } [i] = 0, \text{ for } 1 \le i \le m \\ \text{block-status } [j] = 0 \\ \text{block-time } [j] = \infty \end{array}\right\} \text{ for } 1 \le j \le n.$$

The final contents of these arrays are:

process-status [i] =

$$\begin{cases} 2 & \text{if process } p_i \text{ is a contaminant} \\ 1 & \text{if process } p_i \text{ is contaminated} \\ 0 & \text{otherwise} \end{cases}$$

block-status [j] =

$$\begin{cases} 2 & \text{if block } b_j \text{ is a contaminant} \\ 1 & \text{if block } b_j \text{ is contaminated} \\ 0 & \text{otherwise} \end{cases}$$

block-time [j] =

$$\begin{cases} t, \text{ where } t \text{ is time of earliest transfer at which block } b_j \text{ becomes either a contaminant or contaminated, if either has occurred} \\ \infty \quad \text{otherwise.} \end{cases}$$

Algorithm X

X1. [initialize] $r \leftarrow 0$.

X2. [get next transfer] $r \leftarrow r+1$. $\tau \leftarrow \tau_r$. Choose appropriate substep of step X3. according to whether τ is a contaminator-seed ($\tau \in S$) and whether τ is of the form (p_i, t, b_j) or (b_j, t, p_i).

X3. [update arrays]

(a) $\tau \in S$, $\tau = (p_i, t, b_j)$

if process-status [i] = 0 then
process-status [i] ← 2

if block-status [j] = 0 then
block-status [j] ← 1
block-time [j] ← t

(b) $\tau \in S$, $\tau = (b_j, t, p_i)$

if process-status [i] = 0 then
process-status [i] ← 1

if block-status [j] = 0 then
block-status [j] ← 2
block-time [j] ← t

(c) $\tau \notin S$, $\tau = (p_i, t, b_j)$

if process-status [i] > 0 then
if block-status [j] = 0 then
block-status [j] ← 1
block-time [j] ← t

(d) $\tau \notin S$, $\tau = (b_j, t, p_i)$

if block-status [j] > 0 then
if process-status [i] = 0 then
process-status [i] ← 1

X4. [check for end] If r < k then go to step X2 else stop.

Algorithm X could be used at the time an error situation is detected as a diagnostic tool for recovery. Specification of contaminator-seeds could be done manually, based on available information such as the nature of the detected error(s), knowledge of recent update activity, and spot checks of the

data base. This specification might be made in the form of templates such as:

(a) $(p_i,-,-)$
(b) $(b_j,t,-)$, $t \geq t_1$
(c) $(p_i,-,b_j)$
(d) $(-,t,-)$, $t_1 \leq t \leq t_2$.

Alternatively, an algorithm similar to Algorithm X might be used to generate all transfers which predate a given set of recent transfers known to be in error. The transfer sequence T would be processed in reverse order, back to a specified point in time, and the resulting set of transfers would be taken as contaminator-seeds and used as input to Algorithm X.

V. RECOVERY FROM CONTAMINATION

For applications in which it is feasible to rerun processes, the following algorithm provides a recovery strategy. Input to the algorithm consists of the transfer sequence $T = (\tau_1,\tau_2,\ldots,\tau_k)$ and the three arrays process-status [1:m], block-status [1:n], and block-time [1:n] as produced by Algorithm X. The final contents of these arrays are:

process-status [i] =

- 2 if process p_i is to be rejected
- 1 if process p_i is to be rerun
- 0 otherwise

block-status [j] =

- 2 if block b_j is to be restored to an image prior to block-time [j]
- 1 if block b_j is to be restored to its image at block-time [j]
- 0 otherwise

block-time [j] =

$$\begin{cases} t, & \text{where } t \text{ is the time to which block } b_j \text{ is to be restored, if necessary} \\ \infty & \text{otherwise.} \end{cases}$$

Algorithm R

R1. [initialize] r ← 0. another-pass ← 0.

R2. [get next transfer] r ← r+1. $\tau \leftarrow \tau_r$. Choose appropriate substep of step R3 according to whether τ is of the form (p_i,t,b_j) or (b_j,t,p_i).

R3. [update arrays]

(a) $\tau = (p_i,t,b_j)$

```
if process-status [i] > 0 then
   if t < block-time [j] then
      block-status [j] ← 1
      block-time [j] ← t
```

(b) $\tau = (b_j,t,p_i)$

```
if block-time [j] < t then
   if process-status [i] = 0 then
      process-status [i] ← 1
      another-pass ← 1.
```

R4. [check for end of pass] If r < k then go to step R2.

R5. [check for another pass] If another-pass = 1 then go to step R1 else stop. □

Algorithm R may be used in conjunction with an audit file to recover from contamination. For each transfer $\tau = (p_i t,b_j)$ in T the audit file contains a before image entry $\alpha = (t,b_j,B_j(t))$, where $B_j(t)$ is the image of block b_j just prior to transfer τ. For each j such that block-status [j] = 1, the appropriate audit file entry α is retrieved using indices t = block-time [j] and b_j, and block b_j is restored to its image $B_j(t)$. For each j such that block-status [j] = 2, the audit file is searched for an entry $\alpha = (t,b_j,B_j(t))$ such that: (a) $t \leq$ block-time [j], and (b) t is maximal. If no such audit file entry

exists then either some back up copy of block b_j will be needed, or some special measures will be necessary to reconstruct an appropriate image for block b_j.

It can be shown that Algorithm R has the following properties:

(a) for each process p_i such that process-status [i] > 0, if a transfer (p_i,t,b_j) appears in T then block-time [j] ≤ t.

(b) for each process p_i such that process-status [i] = 0,

(1) if a transfer (b_j,t,p_i) appears in T then t ≤ block-time [j].

(2) if a transfer (p_i,t,b_j) appears in T then t ≤ block-time [j].

From (a) we see that blocks are restored to images that existed prior to any update activity of processes which are rejected or rerun. From (b) we see that all interaction between the remaining processes and restored blocks has taken place prior to the time to which blocks have been restored.

No claim of efficiency is made for Algorithm R. Indeed, it is possible to avoid the necessity for several complete passes of the transfer sequence T by:

(a) modifying T to include an indicator identifying the point at which each process begins, and

(b) in step R3.(b), replacing the assignment another-pass ← 1 by a backup in T to the point at which process p_i begins.

VI. CONCLUSIONS

The transfer sequence T cannot be maintained indefinitely. Truncation of T is not only necessary, but also a means of controlling the amount of system resources allocated to integrity maintenance. Of course, truncation of T restricts contamination tracking and recovery to the period of time spanned by T.

For some applications it may be infeasible to rerun processes. In this case the contamination tracking method of Section IV. could be used as a diagnostic tool in deciding what forward corrective action is required to remedy the problem.

The recovery strategy described in Section V. could possibly be adapted to run concurrently with ongoing data base activity to avoid excessive down time. Affected data base blocks could be taken offline as they are discovered, and placed online after they have been restored.

For some applications it may be desirable to use the transfer sequence T to preserve the original ordering of transfers for rerun processes as far as possible.

This paper has dealt with some logical aspects of contamination and recovery for an idealized multi-process data base model. Various refinements or extensions may be of interest. One concerns refinement of the "postdates" relation, which specifies paths along which contamination propagates, in order to further restrict the identification of contaminated processes and blocks. Another involves recovery strategies for environments in which it is infeasible to rerun processes.

ACKNOWLEDGMENT

This work has benefited from discussions with my colleagues Mr. James L. Black and Dr. Eugene Ott.

REFERENCE

1. C. W. Bachman. The Programmer as Navigator, Comm. ACM, Vol. 16, No. 11, Nov. 1973, 653-658.