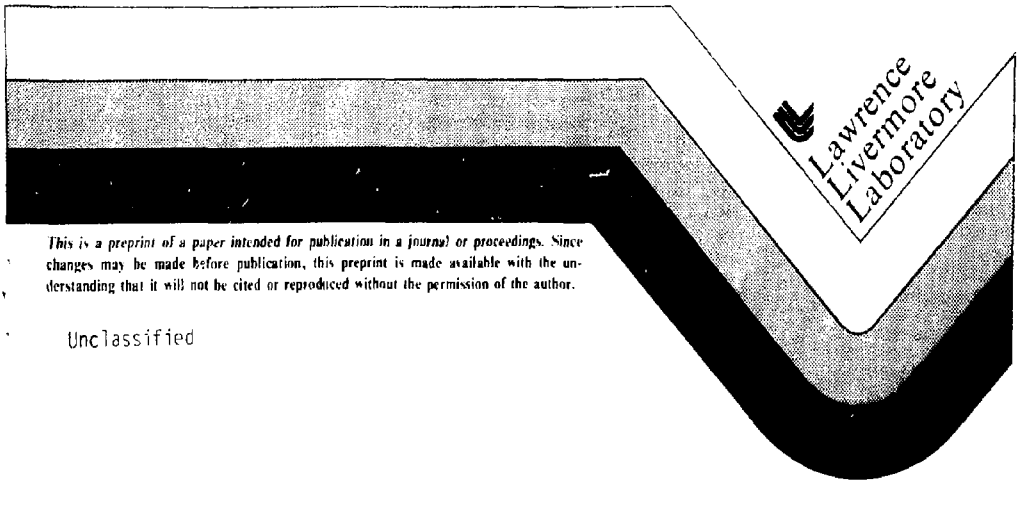
UCRL- 86122
PREPRINT
CONF-811040--26

NOTIFICATION OF CHANGE IN A DATABASE

Eron L. Nelson

This paper was prepared for submittal to
9th Symposium on Engineering Problems
of Fusion Research
Palmer House
Chicago, Illinois
October 26-29, 1981

October 5, 1981



Unclassified

NOTIFICATION OF CHANGE IN A DATABASE*

Bron C. Nelson
Lawrence Livermore National Laboratory
P.O. Box 5511, L-535
Livermore, CA 94550

Summary

The Supervisory Control and Diagnostics System for the Mirror Fusion Test Facility is an event driven system; tasks that handle specific events are active only when those events occur. One method of monitoring and generating events is the data base notification facility; a task can request that it be loaded and started by the dbms if a data element is touched or goes outside of a specified range. The motivations for this facility (along with an example of its use and some specifics regarding how it is done) are presented.

*"Work performed by LLNL for DOE under contract number W-7405-ENG-48."

Introduction

The Supervisory Control and Diagnostics System (SCDS) of the Mirror Fusion Test Facility must be able to recognize and respond to a variety of events. This paper will focus on one event in particular: the writing of data into the database. SCDS supports a large number of status displays and when the data changes the new values may need to be reflected in these displays. SCDS has chosen to solve this problem through the concept of data base triggers: the program that drives the display can request to be notified by the data base when any of the displayed values change.

There are three major reasons why this approach to the update problem was chosen. Most importantly is that it solves the problem; the displays can be kept current and correct. Second, the program that drives the display will be in core and running only when necessary (rather than, for instance, periodically polling the data of interest). Third, it provides a considerable amount of decoupling. A status display can be created and modified in isolation. No other task needs to know anything about the display, and the display need not worry about where the data are from or what task wrote it. It should be noted that 'writing into the data base' is a sufficiently general event that almost any process can use data base triggers to drive it. Status displays serve as a concrete example.

Data base Triggers

In SCDS, data base triggers are called 'dbnotifies' and can be set with various scopes. A data base table is viewed as an array of records, with the records as the rows of the table, and the record elements as the columns (known as attributes) of the table. The user can monitor a whole table, a single row across all attributes, a single attribute across all rows, or (most commonly) a single attribute of a single row. Dbnotifies that are set on a single attribute may be conditioned on the value written, with notification being sent only when the value goes outside a specified range. (In the interest of speed, more complicated 'query like' conditionals are not supported).

Inter Process Communication System

The ability to pass messages between tasks is provided by the Inter Process Communication System (IPCS).[2] Messages can only be received by tasks that are active. Since the host operating system does not support virtual memory, if a message is to be delivered to a task that is not currently active, IPCS will load and start the task in order to deliver the message. This provides the perfect mechanism for an event driven system. SCDS tasks are in core only when they are needed to respond to a message, and since the machines only have about a half megabyte of core each, it is important that tasks not be in memory when they are not needed.

Overview

When a task starts a dbnotify, the data base puts a packet of information into a linked list associated with the particular relation. This packet contains (among other things) a description of the portion of the table watched, and the 'mailing address' of the requesting task. Whenever any task writes into that table, the list is searched to see if any other task cares. If so, a message is generated and sent using IPCS.

Example

Consider the following example: Suppose an operator invokes a display module (call it DM34567) and it shows the pressure reading in a pipe. DM34567 reads and formats the data, and displays it on a monitor screen. In addition it executes a statement such as

DBNOTIFY START ON pipetable (pipe12).pressure WITH 2.

Here, 'pipetable' is the name of the database table desired, and 'pressure' is the attribute name. The 'WITH' clause is a user defined integer that is returned in the dbnotify message (its purpose is to aid the user in distinguishing which dbnotify generated the message in case the user has several outstanding concurrently). DM34567 now goes end of task and leaves memory. Whenever anyone writes to 'pipetable' the dbnotify list is checked. Eventually, someone writes to the data element DM34567 cares about. A message is generated and sent to a special core resident message routing task: the DBMS task. The DBMS task now passes the message to DM34567. (The reason for this 'extra' step is to handle non-fatal errors in a nice way. Since there is no virtual memory, it may well be the case that DM34567 will not fit into core at the exact moment that a message is passed to it. IPCS returns an error in this case. Clearly the sending task should try to send the message again later, but the task doing the writing should not have to hang up waiting to pass the message. Thus, the DBMS task handles the message passing, and will retry periodically until it succeeds.) DM34567 receives its message, reads the current pressure value out of the data base, updates the display, and leaves memory again. Eventually, the operator decides to look at something else, and so sends DM34567 a 'stop' message. The monitor screen is cleared and DM34567 executes a

DBNOTIFY STOP ON pipetable (pipe 12).pressure WITH 2.

This deletes the dbnotify so that messages are no longer sent to DM3456/ when that data element is written.

Dbnotify Problems

The worst problems with implementing dbnotifies are cleaning up: When a program abnormally terminates, finding and deleting its outstanding dbnotifies is quite difficult. But the trickiest problem is getting the DBNOTIFY STOP command to have the desired effect. After deleting the dbnotify so that no new messages will be sent, the DBNOTIFY STOP must ask the DBMS task to delete any messages previously generated by the dbnotify that have not yet been sent, and then look in the requesting task's message receiving area and delete any messages from the dbnotify that have been received but not yet acted upon.

From a user point of view, the worst problem is that dbnotifies are too slow. Part of the problem is that they have proved so popular that users set dozens at a time and the linked list grows very rapidly which slows things down tremendously. However, most of the speed problem can be blamed on the data base memory management schemes currently in use. In the most closely timed test as of this writing, a 7 second delay was observed between the time a pressure reading changed, and the time that change was reflected on the status display. Between 2 and 5 seconds of that time was spent in the database. In order to improve this, the data base paging mechanism is being rewritten and dbnotifies will switch over to a b-tree structure rather than a simple linked list.

Other Solutions

There are several other solutions to the problem of keeping status displays current that deserve mention. Instead of having individual tasks that drive displays a single task could drive all displays and be informed everytime someone did a write. Such a task quickly becomes quite large, is almost continually active, and would be continually modified as displays are created and changed. Alternately, the task that wrote the data could be responsible for informing all tasks that were interested. This approach creates a considerable communication problem. Whenever a display is modified, the corresponding data writers must also change. Communication quickly becomes quite complex and modification very tricky. The easiest alternative is to have each display periodically re-read and redisplay the data. This solution simply ties down too much of the available computer resources with continual and unneeded activity.

Conclusions

The dbnotify facility has been running quite successfully for almost a year. It neatly solves the problem of keeping status displays current, with the advantages that displays are event driven and decoupled from other tasks. It is sufficiently general that it is useful for other types of process control and coordination. The only change currently planned is to improve the speed.

References

1) Bobrow, D. G., Winograd, T. "An Overview of KRL, a Knowledge Representation Language," Cognitive Science, Vol. 1, No. 1, Jan. 1977.

2) McGoldrick, P. R. IPCS User's Manual, Lawrence Livermore National Laboratory, Internal Report, Dec. 1980.