Interprocess Communication in Real-Time Systems

P.G. Sorenson*
Department of Computer Science
University of Toronto
Toronto, Canada

Summary

A variety of solutions have been proposed for ensuring data integrity in nonreal-time systems (i.e. batch or on-line systems). A brief review is made of some of the techniques employed in these solutions. It is indicated why the data integrity problem is different in a real-time system than in a nonreal-time system. Two models of interprocess communication are presented and it is demonstrated that the models are sufficient to preserve data integrity in a real-time system.

I. Introduction

When designing a real-time system it is advantageous to associate one or more "computational processes" with each environmental activity which is to be computer controlled. A process can be considered to be an entity operating logically independent from all other processes in the real-time system. However, such an assumption would mislead and oversimplify the situation, for it is often necessary or advantageous to share information among processes. For example, consider a real-time system which controls a brine processing plant. One process might be dedicated to the control of the steam flow in the plant. A second process controls the flow of brine to the top of a regeneration tower. The process regulating the brine flow needs the steam state in order to function properly. Therefore, the "brine" process must access information acquired by the "steam" process, i.e. we have interprocess communication.

Interprocess communication has been and is a topic of great interest in the design of operating systems (Dijkstra[1965], Knuth[1966], Saltzer[1966], Dijkstra[1968], Bētournē,et al. [1969], Brinch Hansen[1970], and Habermann [1971]). It has been demonstrated that there are problems with attempting to ensure the integrity of data that are mutually accessible to a number of processes. The following illustrates why this is so. Returning to the brine example, let us assume that the steam state is represented by a vector of values. Suppose the steam process is blocked while in the midst of updating the steam state vector. Later the brine process is allowed to run. If the brine process requests the steam state, it receives an incomplete copy of the state vector. This erroneous information could falsify the action of the brine process and we say the integrity of the steam state data set has been violated.

There have been a variety of solutions proposed for ensuring the integrity of shared data in a multiprocessing system (Dijkstra[1965], Knuth[1966], Habermann[1971] and Gilbert and Chandler[1972]). In this paper we briefly review some of the formalism and techniques employed by these authors in their solutions. Next, it is indicated why the data integrity problem is different in a real-time system than in nonreal-time systems. Finally, we present two of our own interprocess communication models and demonstrate that they are sufficient to guarantee data integrity in a real-time system.

II. Modelling Process Interactions

Before discussing the types of interactions which can take place among processes, some mention must be given to the "dimensionality" of the interprocess communication. Throughout the remainder of the paper we use the two terms: message passing and data sharing. Message passing is a one-to-one type of interprocess communication (i.e. one process sends information which is received by another process). Data sharing is a many-to-many type of interprocess communication (i.e. many processes are releasing information which other and possibly the same set of processes are requesting). A data sharing scheme incorporates a message passing scheme if we restrict both the set of processes which release information and the set of processes which request information to sets of size one which do not contain common elements. In some of the interprocess communication schemes which we will discuss, restrictions are made on the dimensionality of the interprocess communication in an effort to provide a more structured and understandable scheme of data sharing.

Horning and Randell [1972] cite two categories of interactions among processes. Co-operation includes all interactions which are anticipated and desired. Interference is that interaction which is unanticipated or unacceptable. Relating these interactions to interprocess communication, we can speak of co-operative messages (or co-operative

*Current address:
Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan

shared data) and interfering messages (or interfering shared data). We introduce the following definition.

Definition 1: The integrity of a system is preserved (or guaranteed) if all data sharing within the system is co-operative.

In the earlier example, the integrity of the brine system cannot be guaranteed because the steam state information is unacceptable. We would like to devise a scheme which guarantees that the data sharing is always acceptable.

Dijkstra [1965] originally posed the data integrity problem and offered a partial solution (see Knuth [1966]). In a later paper (Dijkstra [1968]), a more detailed and elegant solution is given in terms of two synchronization primitives, P and V. In his solution, Dijkstra points out that all operations performed on shared data by a given process must be completed mutually exclusive of all other processes (i.e. if process A is operating on shared data $\mathcal{D}$ then no other process is allowed to operate on the same shared data before A has completed all of its operations on $\mathcal{D}$). The sections of program which contain the operations on the shared data are referred to as critical sections.

Others have suggested and used synchronization primitives which are very much like Dijkstra's P and V (e.g. BLOCK and WAKEUP (Lampson [1968], and WAIT and SIGNAL (Habermann[1971])). However, all of these synchronization operations are primitive operations. They facilitate the writing and demonstration of correct programs through the use of critical sections, but they are somewhat removed from or are independent of the sending or receiving of the messages. Brinch Hansen [1970] and Sevcik, et al. [1972] have used "higher level" operations such as SEND and WAIT on message buffers and SEND and RECEIVE on facilities (where a facility might be a message resource). Hoare [1972] has suggested the use of a monitor which regulates the message passing (or data sharing) resources of the system. Synchronization within the monitor is achieved by WAIT and SIGNAL instructions. The following example illustrates how such higher level "primitives" might be used to solve the data integrity problem for the brine system.

```
steam: PROCESS;
          .
          .
       FOR i := 1 UNTIL n DO
          READ steamstate(i) from device(i);
       SEND (steamstate,mlbx1);
       /*send steam state via the mailbox mlbx1*/
          .
          .
END steam;

brine: PROCESS;
          .
          .
       RECEIVE (steamstate,mlmx1);
       /*receive the steam state from mailbox mlbx1*/
       FOR i := 1 UNTIL n DO
          valveset(i) := steamstate(i) * function(i);
          .
          .
END brine;
```

In order that the processes proceed correctly and co-operative data sharing occurs, the SEND and RECEIVE operations must be performed mutually exclusive of all SEND and RECEIVE operations on the same shared data.

In this section thus far, we have introduced the notion of integrity and have presented, without detail, a variety of models which have helped to solve the problems of preserving shared data integrity within a system. All of the models are valid in operating systems designed for on-line or batch environments. There are two reasons why the models are not adequate for a real-time environment.

1. In the models given, a process can be blocked for a substantial amount of time waiting to enter a critical section to release or request shared data. In a nonreal-time system, this can be accomplished without disastrous effects on the system environment. By the inherent nature of real time, processes cannot afford the luxury of synchronization (i.e. the situation "I'll wait for your message, wake me up when you send it" cannot be allowed in a real-time system). The information a process, say A, is waiting for may be from a process, say B, that is proceeding much slower than A. By waiting, the progress of A is, in effect, reduced to that of B.

2. In a real-time system, it is often the case that the process receiving a message does not care which message it receives as long as it is the latest one sent. If the data received is "old", the action of the receiving process may jeopardize the system. Hence, the integrity of any shared data is time dependent in a real-time system. In a later section in this paper we investigate in greater detail the notion of time dependent data.

III. An Interprocess Communication Model for Real-Time Systems

The following definitions are used through the remainder of the paper.

Definition 2: Any process that assigns information to a shared data set is a releasing process.

Definition 3: Any process that acquires information from a shared data set is a requesting process.

For an interprocess communication model to satisfy the two conditions stated at the end of the previous section:

(i) a process must not wait for another process when either releasing or requesting data, and
(ii) the most recent information to be released must be received by the requesting processes.

These two criteria enable us to define more rigorously what is meant by the preservation of integrity in a real-time system.

Definition 4: The integrity of shared data in a real-time system is preserved if all data sharing is co-operative and any information received by a requesting process is the latest to be sent by the corresponding releasing process or processes.

We investigate two methods for attaining integrity. The first method involves explicit control over the progress at which the data sharing processes are allowed to proceed. The method is presented

because it is easy to understand and because it characterizes the problems surrounding interprocess communication in a real-time system. A more elegant model is proposed which involves the control of a number of shared buffers ("data set copies") by the issuing of RELEASE and REQUEST commands. This alternative model is free of two deficiencies present in the first model. These deficiencies become evident in the discussion to follow.

A. Regulating Process Speed to Achieve Data Integrity

To achieve co-operative data sharing a process must be allowed to send information without interference from another process. Generally, interference takes place

(i) when two or more processes attempt to send, concurrently through a common variable (mailbox, buffer, or data set), information to the same set of requesting processes, or

(ii) when a requesting process accepts a partially completed set of shared data.

By controlling the progress of each process so that no requesting process or any other releasing process can interfere with a process during the time it is releasing information, co-operative data sharing can be achieved.

To properly examine the model of interprocess communication by process speed regulation a number of terms must be introduced.

Definition 5: A process interval is some part of the activity of a process during its "lifetime". (For example, if p is a releasing process, then we can refer to the "process interval of p over which a message is sent").

Definition 6: The framesize (or response time) f_i for a real-time process p_i is the maximum amount of time(wall clock time) allowed p_i before it must be completed.

Definition 7: The quantum size q_i is the amount of processor time allotted to real-time process p_i during one activation.

Definition 8: The dynamic frame size $f_i(t)$ at time t is the amount of wall clock time remaining before the current frame for real-time process p_i has expired.

Definition 9: The dynamic quantum size $q_i(t)$ is the amount of processor time remaining in quantum q_i at time t when executing p_i.

Definition 10: The speed of a real-time process p_i at time t is the quotient $q_i(t)/f_i(t)$.

We see from Definition 10 that a process can be effectively speeded up (slowed down) by reducing (increasing) the dynamic frame size.

In the theorem to follow, we make reference to the "least time to go" scheduling discipline.*

*Fineberg and Serlin[1967], Lubran and Roberts [1972], and Liu and Layland [1973] all have suggested the least time to go discipline for scheduling real-time processes. Sorenson [1973] has demonstrated that least time to go scheduling is optimal for an objective function which characterizes process activity in any real-time environment.

If, at time t, $p_1,p_2,\ldots,p_n$ are processes with dynamic frame sizes of $f_1(t),f_2(t),\ldots,f_n(t)$ where $f_i(t) \leq f_j(t)$ for $1\leq i<j\leq n$, then the processes are scheduled by least time to go if they are given service in order $p_1,p_2,\ldots,p_n$. We see how this discipline plays a central role in the following theorem.

Theorem 1: Assume the processes of the real-time system are scheduled according to the least time to go discipline. Let P_Q and P_L be sets of requesting and releasing processes which interact through a shared data set $\mathcal{D}$. If the time to go is made equal for all processes, P_Q and P_L, at the beginning of the process intervals in which they request or release through $\mathcal{D}$, then the integrity of $\mathcal{D}$ is preserved within the real-time system.

Proof: Let p_i be a process, $p_i \varepsilon P_Q \cup P_L$. Let t_0 be the time at which p_i enters a process interval in which it requests or releases information through $\mathcal{D}$. Assume the time to go for p_i is set at f (implying that the frame end time is set at $t_0 + f$). At time t_1, before p_i completes the process interval in which it is acting upon $\mathcal{D}$, some other process, $p_j \varepsilon P_Q \cup P_L$, begins a process interval in which it requests or releases data through $\mathcal{D}$. The time to go for p_j is set at $t_1 + f$. Since processes are scheduled by least time to go, p_i has priority over p_j (i.e. $t_0 + f < t_1 + f$). Therefore, p_i completes all transactions referring to $\mathcal{D}$ before p_j is allowed to request or release information through $\mathcal{D}$. Hence, p_j can never interfere with p_i and we have co-operative data sharing.

Note that $\mathcal{D}$ always contains the latest data sent (i.e. when a process releases new information the old copy of $\mathcal{D}$ is destroyed whether it was received or not). Since p_i and p_j were chosen arbitrarily, the integrity of the shared data $\mathcal{D}$ is shown to be preserved in the real-time sense. Δ

Theorem 1 tells us that interacting processes can be synchronized by controlling the speed of each process activity. In this manner we have violated to some degree one of the criteria stated at the beginning of this section, namely, a process must not wait for another process when either requesting or releasing shared data. This problem can be avoided, in part, by momentarily setting the time to go for each interacting process such that the average speed of the "fastest" interacting process is maintained during intervals of interaction on $\mathcal{D}$.* By speeding up slower processes to maintain the speed of the fastest process, the process demands of the remaining processes are neglected temporarily. This neglect leaves the system suspect of overloading. (Overloading is a condition in which one of

*Speed is a very dynamic measure, depending on the system load at any given moment in time. There is, consequently, no obvious way of calculating "the average speed of the fastest process". An approximation may be achieved by first calculating a quantum time needed to execute the fastest process over the process interval in which the interaction takes place. Based on this quantum estimate, a frame size may be computed such that the speed over the process interval is equivalent to the initial speed of the fastest process.

the real-time processes is unable to complete its computation before its current frame expires, i.e. $q_i(t)>0$ and $f_i(t)=0$.) Because interactions due to interprocess communication can take place rather unpredictably in a parallel processing system, any attempt to analyze system overload while taking into account these temporary speed-ups, is an almost impossible endeavour.* Instead we present an alternative model which satisfies the criteria for shared data integrity in a real-time system and also eliminates process speed-ups and permits system overload analysis.

B. A Controlled Data Set Model to Achieve Data Integrity

It may not be desirable or even possible to regulate process interactions using frame size control, due to the additional system load created by temporarily reducing the frame sizes of interacting processes. An alternative method is to use what will be referred to as controlled, shared data sets (henceforth called controlled data sets).

Definition 11: A controlled data set $\mathcal{D}$ is a set $\{D_1,D_2,\ldots,D_Z\}$ of data set copies in which each copy has the potential to contain information from a releasing process which may later be accessed by a requesting process.

Each data set copy of a controlled data set $\mathcal{D}$ can be in one of four states:

(i) If D_i is in a sending state (s) then all updating by a releasing process is made on D_i and a requesting process is not allowed to access D_i.

(ii) If D_i is in a completed state (c) then D_i is the latest data set copy to be updated by a releasing process of $\mathcal{D}$. New data cannot be added to D_i. The information in D_i may be accessed by any requesting process that is not already accepting data from $\mathcal{D}$. In this state a user count is associated with D_i. The user count is initially zero and is incremented by one each time a new requesting process accesses the data set copy. Only one data set copy can be in the completed state at one time.

(iii) If D_i is in the receiving state (r) then only those requesting processes which accessed D_i in the completed state may continue accessing D_i. New data may not enter D_i. When a requesting process finishes with D_i, the user count for D_i is decremented by one.

(iv) If D_i is in the available state (a) then all requesting processes have finished with the current copy of D_i (i.e. the user count is zero). Any data set copy in the available state can be used again by a releasing process wishing to begin a new update on $\mathcal{D}$.

When $\mathcal{D}$ is initialized, all data set copies start in an available state except for one,

* Sorenson [1973] has investigated the problem of determining system overload given the initial quantum and frame size estimates for the processes of the system. It has been established that if

$$\sum_{i=1}^{n} q_i/f_i \leq 1,$$

system overload can be avoided. However, if we attempt to accommodate a model which allows occasional process speed-ups, overload analysis is unrealizable.

which starts in the completed state. The data set copy initially in the completed state contains the initial value of $\mathcal{D}$. All user counts are set to zero initially.

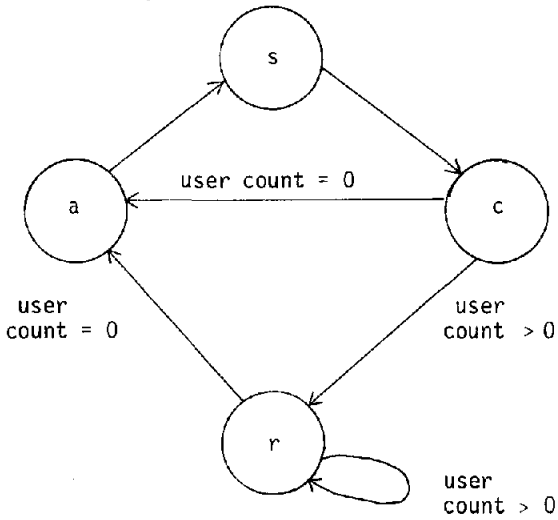

Fig. 1 State Transition diagram for data set copy D_i

Possible state transitions (under mapping S) for a data set copy are as follows (refer to Fig.1);

(i) Transition $S(a) \rightarrow s$

When a releasing process encounters a release type of command, a data set copy, say D_i, is selected from the available pool. The state of D_i changes from available to sending. The process may now update D_i.

(ii) Transition $S(s) \rightarrow c$

Once the releasing process signals that it is finished releasing information to D_i, D_i goes into the completed state. At the same time the data set copy that was previously in the completed state is forced into either a receiving state or an available state. Now that D_i is the latest data set copy to be updated, any data requested from $\mathcal{D}$ is acquired through D_i

(iii) Transition $S(c) \rightarrow r$ or a

Upon completion of a new data set copy, D_i is moved from a completed state to a receiving state or an available state. If a requesting process has not finished accessing D_i (i.e. the user count > 0) then the state transition for D_i is from completed to receiving. However, if all requesting processes have finished with D_i, then D_i becomes available.

(iv) Transition $S(r) \rightarrow r$ or a

If D_i is in state r and a requesting process finishes with D_i, then the user count is decremented by one. If the count is zero, D_i is sent to the available pool. If the count is greater than zero, a requesting process is yet to complete its transaction on D_i. Therefore D_i remains in state r.

Theorem 2: Assume there are N releasing processes and M requesting processes interacting asynchronously on a controlled data set $\mathcal{D}$. Then to ensure the integrity of shared data, $\mathcal{D}$ must contain at least $Z = N + M + 1$ data set copies.

Proof: Looking at the allowable state transitions for a controlled data set, we see that at least one data set copy must be in the completed state at one

time. Hence $Z \geq 1$.

Suppose all N releasing processes are active at time t (i.e. all have started sending information). In order to prevent the loss of any data set copy and to ensure co-operative data sharing, each of the releasing processes must have a separate data set copy. Therefore $Z \geq 1 + N$.

Assume the M requesting processes are active at time t also, and all have started to receive data prior to the completion of the data set copy currently in the completed state. In addition, it is possible that each requesting process commenced receiving at times $t_1, t_2, \ldots, t_M$ such that a different completed data set copy was available at each of $t_1, t_2, \ldots t_M$. To ensure that each requesting process is able to acquire a coherent message, a data set copy must be supplied for each process. Therefore $Z \geq N + M + 1$.

It is easy to show that the N + M + 1 data set copies are sufficient. Since all the processes which interact with $\mathcal{D}$ are busy at time t, any additional data set copies (i.e. $Z > N + M + 1$) must be in available states. We see (by viewing the state transition diagram) that no matter what transition takes place for any of the N + M + 1 unavailable data set copies, one of the data set copies must enter an available state. Hence, the need for additional data set copies is not justified. Hence $Z = N + M + 1$.

Δ

Theorem 3: Subject to the condition that $\mathcal{D}$ contains N + M + 1 data set copies, the controlled data set model for interprocess communication is sufficient to preserve shared data integrity in a real-time system.

Proof: Let P_L and P_Q be sets of releasing and requesting processes for controlled data set $\mathcal{D}$. Assume $p_i \varepsilon P_L$ enters a process interval in which it releases data through $\mathcal{D}$. A data set copy, say D_k, is selected from the available pool and is assigned to p_i (by Theorem 2 we know that one such data set copy must be present). By the restrictions on the controlled data set model, no other process can access D_k until it is released by p_i. Therefore, when releasing shared data, no interference can take place.

Assume $p_j \varepsilon P_Q$ enters a process interval in which it receives data through $\mathcal{D}$. The data set copy currently in the completed state is assigned to p_j. Let this data set copy be called D_k. Data is acquired through D_k while in the completed state or in the receiving state if D_k undergoes a transition. Data sharing interference is impossible since new data cannot be released to either a completed or receiving data set copy. A requesting process always acquires the latest, coherent message provided in $\mathcal{D}$ (i.e. the information in the completed data set copy at the time p_j begins receiving). Therefore, by Definition 4, the integrity of the shared data is preserved and the controlled data set model is sufficient in a real-time sense.

Δ

The requirement that $\mathcal{D}$ contains N + M + 1 data set copies is unrealistic in view of the type of message passing that occurs in real-time systems. Real-time processes are "data driven" (i.e. their activity is heavily dependent upon the current conditions in the real-time environment). Rarely is it the case that two or more processes are needed to collect, then release, essentially the same type of information to a common set of requesting processes. If the information to be released by two processes describes two logically different entities or situations, then two controlled data sets should be used, not one.*

Let us further examine interprocess communication in a real-time system. It is often the case that once a releasing process completes work on a shared data set $\mathcal{D}$, it activates a requesting process (or processes) of $\mathcal{D}$. (Typically, the requesting processes are responsible for calculating long-term objective functions for the system) Meanwhile, the releasing process begins updating a new data set copy and need not complete this task before the requesting process (or processes) finish with the old data set copy. In such a situation, the releasing and requesting processes are effectively synchronized.

Definition 12: A releasing process p_i and a set of requesting processes P_Q are synchronized in real time if the P_Q are activated explicitly by p_i and p_i releases at most one message for every activation of p_j, for all $p_j \varepsilon P_Q$.

Theorem 4: If the releasing process p_i and the set of requesting processes P_Q are synchronized in real time through controlled data set $\mathcal{D}$, then $\mathcal{D}$ need only contain two data set copies to preserve the integrity of the data shared by p_i and P_Q.

Proof (by cases): Because p_i and P_Q are synchronized in real time, there are only four cases to consider.

Case 1: p_i not releasing, P_Q not requesting
Obviously, zero data set copies are required.

Case 2: p_i releasing, P_Q not requesting
One data set is required for p_i, and it is in the sending state.

Case 3: p_i not releasing, P_Q requesting
One data set copy is in the completed state containing data just sent by p_i. (P_Q cannot be requesting unless p_i previously was active and had completed the sending of the shared data.)

A second data set copy is in the receiving state corresponding to the data which the processes P_Q originally started to receive. Note that all requesting processes are operating on the same data set copy since p_i releases at most one set of data for every activation of each $p_j \varepsilon P_Q$. Therefore, in total, two data set copies are required for this case.

Case 4: p_i releasing, P_Q requesting
One data set copy is required by p_i and the copy is in the sending state.

A second data set copy is in the completed state. Because the releasing and requesting processes are synchronized in real time, all releasing processes acquire information from the completed

* In the model, we have allowed for multiple releasing processes primarily because the general case is no more complicated, other than the additional data set copies, than a model with only one releasing process. If we should desire multiple releasing processes, the mechanisms are there for us to use.

data set copy. (That is, if it is possible for a subset of P_Q to receive data from a data set copy other than the completed copy, then p_i must be releasing more than one data item per activation for some p_j, $p_j \in P_Q$.) Since there are no other processes of concern, case 4 requires two data set copies.

All possible cases have been considered and at most two data set copies are needed. By Theorem 4, the controlled data set model is sufficient to preserve the integrity of shared data. Therefore, we conclude that $\mathcal{D}$ requires only two data set copies to preserve the integrity of data shared by processes synchronized in real time. Δ

Theorem 4 points out that if processes interact synchronously, the size of the controlled data set can be reduced appreciably (i.e. from M + 2 to 2 data set copies).

IV. Co-operative Data Sharing in a Real-Time Sense

Before concluding an analysis of real-time interprocess communication, a remark is in order with reference to co-operative data sharing in a real-time sense. The following program is an example illustrating how two processes can be made to interact co-operatively. The processes operate on a common variable x which is initially set to zero.

```
proc1: PROCESS;          proc2: PROCESS;
         :                        :
       P(mutex);                P(mutex);
         x := x + 1;              x := x + 1;
       V(mutex);                V(mutex);
         :                        :
END proc1;               END proc2;
```

Without the synchronization primitives, P and V, operating on the semaphore "mutex", it is possible that the value of x is 1 after each process has executed "x := x + 1;". Of course the correct value for x should be 2. In this instance, we refer to x as being real-time independent. That is, independent of when in real time each process operates on x, the correct answer should be 2. Notice what happens when the same example is formulated using a controlled data set $\mathcal{D}$.

```
proc1: PROCESS;*         proc2: PROCESS;
         :                        :
       REQUEST(x,D);            REQUEST(x,D);
         x := x + 1;              x := x + 1;
       RELEASE(x,D);            RELEASE(x,D);
         :                        :
END proc1;               END proc2;
```

It is possible for proc1 to receive x (x = 0) at time t_0 and to proceed to increment the value of x. Before proc1 is able to send a new value of x, proc2 is scheduled to run and it receives x (x = 0) at time t_1. Both processes are allowed to proceed, each eventually releasing a value of x equal to 1 - a logically incorrect value! The point to be made is that releasing and requesting through a controlled data set can not be applied without discretion to data that is real-time independent.

Let us consider data that is real-time dependent. Assume in the above example that x is a measure of some condition in the real-time environment. Suppose proc1 is the only process which broadcasts this condition to other processes (i.e. in the example, RELEASE(x,$\mathcal{D}$) is considered to be removed from proc2). Then it is reasonable that x should be zero at time t_1 for indeed that is the state of the environment at t_1.* The entity which the value of x is representing has become real-time dependent.

There may be instances when real-time independent data sharing must take place in a real-time system. The following sample program shows how this might be accomplished. ($\mathcal{D}1$, $\mathcal{D}2$ are controlled data sets; x, x1, x2 are all initially zero).

```
proc1: PROCESS;                  proc2: PROCESS;
         :                                :
       x1 := x1 + 1;                    x2 := x2 + 1;
       REQUEST(x,D2);                   REQUEST(x,D1);
         x := x + x1;                     x := x + x2;
       RELEASE(x1,D1);                  RELEASE(x2,D2);
         :                                :
END proc1;                       END proc2;
```

The value of x in both proc1 and proc2 will be correct for a given request time (i.e. the value released is logically correct for the state of the system at the time x is last received). That is, if proc1 and proc2 are each activated alternately ten times beginning with proc1, then the value of x for proc2 is twenty (as it should be). If the same process activity takes place for the example previous to the one above, the value of x could be any integer value between 10 and 20 inclusive.

V. Conclusion

At the beginning of this paper, we introduced some of the interprocess communication schemes that have been developed for nonreal-time systems. It was pointed out why these schemes are insufficient to handle interprocess communication in a real-time sense. The notion of shared data integrity in real-time systems was examined. Two models were presented both of which are sufficient to ensure the preservation of this integrity. The first model involves the control of process scheduling such that processes, interacting through shared data, are prevented from interfering with each other. An implied process speed-up creates problems, however. Noninteracting processes are slowed down at the expense of interacting processes and system load analysis becomes almost impossible.

A second model, the controlled data set model, captures the essence of data sharing in real-time systems. Processes, operating on coherent messages, are not forced to wait or slowdown. Interprocess communication takes place through a number of data set copies. Control over each data set copy is

* REQUEST(x,$\mathcal{D}$) means assign the contents of the completed data set copy to x. RELEASE(x,$\mathcal{D}$) means transfer the value of x to a data set copy in the sending state.

* While proc1 is updating, the state of the environment is zero; therefore it is reasonable for proc2 to read zero.

imposed through a set of allowable state transitions. It was shown that the number of data set copies in a controlled data set can be substantially reduced for certain types of process interaction. Finally, the notions of real-time independent data and real-time dependent data were identified and were discussed with reference to real-time interprocess communication.

REFERENCE LIST

Bétourné, C., et al. "Process management and resource sharing in the multiaccess system 'ESOPE'", Comm. ACM, Vol. 12, Dec. 1970, pp. 727-733.

Brinch Hansen, P. "The Nucleus of a Multiprogramming System", Comm. ACM. Vol. 13, No. 4, April 1970, pp. 238-241.

Dijkstra, E.W. "Solution of a problem in concurrent programming control", Comm. ACM, Vol. 8, No. 9, Sept. 1965, p. 569.

Dijkstra, E.W. "The structure of the 'THE'-multiprogramming system",Comm. ACM, Vol. 11, No. 5, May 1968, pp. 341-346.

Gilbert, P. and Chandler, W.J. "Interference between communicating parallel process", Comm. ACM, Vol. 15, No. 6, June 1972, pp. 427-437.

Habermann, A.N. "Synchronization of communicating process", Comm. ACM, Vol. 15, No. 3, March 1972, pp. 171-176.

Hoare, C.A.R. "Nucleus for a Structured Multiprogramming System", unpublished, 1972.

Horning, J.J. and Randell, B. "Process Structuring", ACM Computing Surveys, Vol. 5, No. 1, March 1973, pp. 5-30.

Knuth, D.E. "Additional comments on a problem in concurrent programming control", Comm.ACM, Vol. 9, No. 5, May 1966, pp. 321-322.

Lampson, B.W. "A scheduling philosophy for muti-processing systems",Comm. ACM, Vol. 11, No. 5, May 1968, pp. 347-359.

Liu, C.L. and Layland, J.W. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", JACM, Vol. 20, No. 1, Jan. 1973, pp. 46-61.

Lubran, J.F. and Roberts, J.D. "Some observations on least time to go scheduling", The Computer Journal, Vol. 15, No. 1, 1972.

Saltzer, J.H. "Traffic control in a multiplexed computer system", (th.), Report MAC-TR-30, Proj. MAC, MIT, Cambridge, Mass., 1966.

Sevcik, K.C., et al, "Project SUE as a Learning Experience", Proceedings of the AFIPS FJCC, Anaheim, Calif. Vol. 41, Dec. 1972.

Sorenson, P.G. "A Design Methodology for Real-Time Systems", Ph.D. Thesis, University of Toronto, Toronto, Canada, 1973 (expected)