SLAC-PUB-949
(MISC)
August 1971

A MULTIPROGRAMMING SYSTEM FOR PROCESS CONTROL*

Sam Howry

Stanford Linear Accelerator Center
Stanford University, Stanford, California 94305

Abstract

This paper describes an operational small computer multiprogramming system developed for the control of the Stanford Two Mile Linear Accelerator (SLAC). The system has many features of larger systems such as dynamic memory allocation and interprocess control, but does not have to handle typical batch type jobs which need large arrays and many other system resources. This difference results in a drastic reduction in the complexity of both design and implementation. The accelerator control problem is discussed in terms of what requirements are imposed on the system by the environment. Then the basic subsystems are described with sufficient examples to show the reader other areas where such a system may be applicable.

I. Introduction

The purpose of this paper is to describe a relatively abstract multiprogramming system, called DS for Disk System, the implementation of which is proving to be an ideal vehicle for a large process control application. Such a system must provide for the orderly execution of many simple functions such as display of status and analog data, and the activation of control buttons. In a non-automated control room, these functions are performed, independently and potentially in parallel, by operators using many separate electronic interface boxes. In order to justify itself economically, a computer must do much more; it must collect and coordinate the data and functions, allowing carefully measured amounts of interaction between them and thereby producing higher level data acquisition and control processes. This interaction must not be permitted to destroy the modularity inherent in the old electronic boxes, however. It must be possible to program new functions and processes with only minimum concern for those already embedded in the system. Finally, it is obvious that the computer system must be reliable, capable of running several weeks without crashing in the case of the SLAC operation. Through multiprogramming, the DS system provides the appearance of independent and parallel execution of functions. Modularity is facilitated by the disk file system for programs, each of which is limited to a self-contained page size. New programs may be created on-line by use of an interactive text editor. Simple and uniform design of the DS system itself resulted in a compact, reliable implementation, which was a big factor in making the system acceptable for accelerator operations.

The accelerator control room computer is a Digital Equipment Corporation PDP-9 with an 8K memory, console teletype and a million word fixed head disk. There is also a 3 KC synchronous duplex data transmission link to an SDS 925 computer in another control room. The DS system is currently being implemented on the SDS 925 as part of a project to consolidate SLAC's two control rooms using these two machines.[1,2]

II. Operating Environment

The system on which DS operates is not a typical multiprogramming configuration. As indicated above, there are few standard peripherals. Instructions that start a program can be typed in at the console teletype in the usual manner, but the main "user" of the system is the accelerator itself. Over 5000 accelerator status bits are continuously read into the computer in a long sequence, spread out over a period of .7 second. These must be checked for validity and stored into common data buffers. Any status change is cause for the processor to initiate a task into the system. The mechanism by which this is done is discussed later in the paper.

At a rate of 360/sec, pulse trains of 144 bits are selected from memory and delivered through a computer data channel to drive various accelerator devices. Occasionally tasks may be initiated by the operator to alter the values of these bits, or to change the selection sequence. A large number of relays, about 2000, are available to the system, some of which select among several hundred slowly changing analog signals. Programs exist to sequence, select, close, timeout and open these relays. Tasks which use these programs may be initiated by an operator on the teletype or data link, or by the accelerator via status changes. The relay interface device is slow however and system management of this resource is required. A Direct Memory Access (DMA) channel on the PDP-9 can be activated to read 90 selected values digitized from "sample and hold" signals of rapidly changing analogs.

Operation is characterized by a normally quiet accelerator (several status changes a minute) punctuated by high bursts of changes, which may result in up to 30 tasks suddenly in the system. Most of these are quickly processed or combined, while a few may linger on for a while. Other tasks are built into the system to occur periodically, or at specified times of day. Operators can start specific functions from the TTY such as requests for maintenance data summaries and routine calculations. There are special tasks when the system overloads--accelerator status changes coming in too fast to be processed. Tasks may involve data acquisition and logging, equipment testing, level monitoring, closed loop control and other functions.

From the above description, it can be seen that there is a wide spectrum of tasks. However, most are simple to program if considered singly: a small amount of CPU execution, mixed with real time waits, then termination.

III. Overall System Architecture

The DS system consists of a compact resident program coupled with a disk file subsystem. The disk file structure for the SLAC PDP-9 computer is identical to that of DEC's Advanced Monitor System[3]. Binary and source sequential files are on the disk, in the form of chained blocks, together with a common directory and tag bits. The resident program contains code which emulates twenty DS primatives--pseudo instructions which will be described below--and a subprogram which relocates the binary files into core memory pages. The resident program also controls the dynamic memory allocation and the task scheduler. Additional core memory is taken up by programs specialized to the SLAC environment, such as the interrupt drivers, and associated data buffers. Accelerator interface input, data link I/0 and TTY output

*Work supported by the U. S. Atomic Energy Commission

(To be presented at the Third ACM Symposium on Operating Systems Principles, Stanford University, Stanford, Ca. (October 1971).)

data are buffered by separate circular storage mechanisms of fixed lengths. A resident program filters accelerator status input data, putting the results into fixed common arrays. These arrays are available to all users of the system. A fixed region on the disk, dedicated for use outside the file system, is also available for common storage, primarily data logging. These blocks are accessed directly by the use of DS primitives.

IV. Dynamic Memory Management

The dynamic memory management scheme is extremely simple. At system load time, what remains of core memory is used to create two linear singly linked lists. The cell structure is illustrated in Figure 1. The small cell list is the free list for all system and user requirements (although the user refers to it only through DS primitives). Because of the pointers, there is only 50% utility of this storage but there are several compensating effects. There is no fragmentation, so no garbage collection programs are required. Since it is more convenient, users will tend only to acquire single cells when needed, rather than looking ahead and ordering a block of store (although users can also reserve a buffer from the large cell list--see below). A less obvious advantage is that the critical period, when a cell is transferred between this list and some other list, requires only 6 instructions (change 3 pointers). As pointed out by Mills[5] most smaller computers lack context switching capabilities and therefore interrupt driver programs must limit their effect on the task queues. Although DS also prohibits task changing except at prescribed points, it could mask off the interrupts during this short critical period, thereby making it possible for interrupt drivers to at least allocate/ deallocate dynamic store. It will be shown below that this is equivalent to allowing task initiation/ termination at the interrupt level.

The large cell linked list provides a structure for data buffers and areas for resident images of program pages. The choice of 520 words per cell was made so that two 256 word blocks (the basic unit in the disk file system) can be accommodated in one buffer, allowing the file oriented tasks to be more efficient. Also, it is a nice size for program pages. If a binary file can fit into a large cell it is called loadable. After loading, the second word of the cell contains the page name i.e. - the file name of the binary program. User tasks can also reserve and free buffers from the large cell list, by means of DS primitives, for array and file processing. In this case, the second word contains special marks to denote a reserved or released buffer. The third word of the cell contains an internal time, representing when the page was last used. This number is used to optimize the resident page overlays. The large cell list is ordered in decreasing time (except for buffers, which may be anywhere in the ordering). Cells are never detached from the large cell list; only reshuffling by the system is permitted.

V. Tasks and Programming

A basic unit of DS is the task. It is a linear linked list of varying length, made up of cells detached from the small cell list. This is the "seed", containing all system and problem related information necessary for the task function to execute properly. The system controls the first four cells of the task, and the remainder is completely under user control. The user portion of the task is called the argument list. Convenient, fast system primitives permit the user to select, insert, delete and change cells of the argument list. Two of the four system cells contain the current location of the task--page name and a number giving relative location within the page. The other system cells contain the scheduled execution time t_j, in which case it is an active task, or an event name, in ASCI characters, if the task is blocked. It is important to note that a task does not include a program. No distinction is made in this paper between a task and what is usually referred to as its Task Control Block (TCB). The collection of all loadable binary files on the disk make up the program data base, and is available to all tasks. This is true even though each page is a separate assembler output containing no external label definitions other than the file name.

At any point in time, each task is either:

1. On the active task queue, (time driven)
2. On the blocked task queue, (event driven)
3. Executing.

At most, one task may be executing, in which case, the system is in user mode. During this time the task has full control of cells in its argument list, as well as any local variables in its page, and the common data buffers. Flow proceeds sequentially as dictated by the executing program except for the interrupt drivers. These latter programs are small and limited for the most part, functionally little more than sophisticated channels, putting data into and out of fixed buffers. Control is always returned to the interrupted task. Since there is no provision (other than a system timeout trap) for the system to unilaterally suspend user mode, the programmer must do this frequently to prevent input buffer overflow and output buffer underflow due to unprocessed I/O. There are several system primitives available which do this. Each of these provide a system break -- a point where each task is on one of the two queues. The system break-causing primitives also allow manipulation of the four system cells of the task, such as branching (with or without return linkage) to another page, forcing a wait for a specified real time interval or a specified event name, and termination.

The strategy of the programmer in this system is as follows: between any two consecutive breaks all locations in his page are 'safe' and re-entrant code is not necessary. These locations are called local variables. However, across any break all important temporary values must be put into the user's argument list because local variables may be altered by another task using the same program page, or because the program page itself may be reloaded into a different large cell by the system task scheduler (see Section VI below). This means that breaks should be programmed at "good stopping places", i.e., where there are relatively few temporary variables to be protected. The result is that all program pages are automatically re-entrant since interference by other tasks can occur only at the programmed system breaks. This strategy puts a premium on "thin" programs -- those having a minimum of parameters at selected points not too far apart in time. For large applications codes such as occur in numerical analysis, thin programs probably are difficult to write. Creating a compiler which generates thin object code would be a real challenge! But there are areas where this can be done and the simple accelerator tasks described earlier certainly are included. The DEC keyboard Text Editor[4] was rewritten into DS with system breaks no more than 25 msec apart, and a maximum argument list of 16 variables plus two file records. Most file oriented programs encountered in business applications are probably comparable to the editor in this regard.

VI. Task Scheduling

Current time, counted in units of 1/360 sec since initial system load time, is used in the scheduling of tasks. The active task queue is always ordered as to increasing scheduled execution times t_j of the task:

$$t_1 \leq t_2 \leq \ldots \leq t_N$$

Tasks T_k put onto the queue with no special dealy requirements (such as would happen with a jump to a new page) are assigned t_k=current time. When the system is ready to enter user mode, t_1 is compared with current time t. If $t < t_1$ the system is idle, otherwise an attempt is made to enter the location of the first task. If the task's current page is resident, control is transferred to the location and user mode begins. Otherwise a page roll in is

initiated and the task is reinserted at time $(t_1 + \Delta t)$ back into the queue. Keeping track of absolute time rather than having a hardware timer(s) clocking time between active tasks is different than most operating systems[5,6]. Functionally, the two schemes are identical so long as the scheduler "keeps up with its work." However, in peak load times it can occur that, say

$$t_1 \le t_2 \le \ldots \le t_j \le t \le t_{j+1} \le \ldots \le t_N$$

The quantity $L(t)=(t-t_1)$ is a real time measure of system load and can be used in scheduling new tasks. In some systems where priority information is known about a task, $L(t)$ and the priority could be used to compute an amount of extra time Δt added to the nominal scheduled time t_k the sum being the actual scheduled time. This means that higher priority tasks have a good chance of slipping ahead of lower priority tasks at each system break, but once current time becomes greater than the actual scheduled time of any task, that task's position in the active queue becomes fixed, guaranteeing that even low priority jobs eventually execute.

VII. Inter-Process Communication

Inter-process communication means that two tasks, instead of being programmed to act separately, must now be made to consider each other's existence. A degenerate form is when one task initiates another, and there is a DS primitive to do this. The primitive converts a specified array of local variables of the executing program page into a list and puts it onto the active or blocked queue, thereby defining a new task. Because of this identification of a task with its list, inter-process communication is simply a mechanism for exchanging information between the list representing the executing task, and some other list on the queues. Since, it is difficult to single out a task on the active queue, DS limits communication to the tasks on the blocked queue where the executing task can make the specification by referring to the event name of the blocked task. The currently implemented forms of task communication involve different calls of the .EVNT primitive:

Call	Description
.EVNT $(\ell;0)$ JMP NONE	unblock--unblock the first blocked task with event name ℓ. i.e. - put this task onto the active queue. If there is such a task, (the 'communication successful' case) indicate this to the executing task by skipping the next instruction on return to the calling program. If there is no such task on the blocked queue (the 'communication unsuccessful' case) do not skip the next instruction on return.
.EVNT $(\ell; n, x_1, \ldots x_n)$ JMP NONE	unblock and copy from caller-Do as above, and in addition insert n new cells, with values $x_1, x_2 \ldots x_n$ at the head of the argument list of the newly activated task. $x_1, x_2 \ldots x_n$ are local variables in the same page as the .EVNT primitive.
.EVNT $(\ell;-n, x_1 \ldots x_n)$ JMP NONE	unblock and copy to caller-If the blocked task has n cells in its argument list, then do as first case above, and in addition put into local variables $x_1 \ldots x_n$ the values of the first n cells of the newly activated task. Otherwise, the communication is not successful; do not activate any task and do not skip an instruction on return.

It is also possible for the system itself to call these primitives during a system break. In this case, there is no executing task and the variables $x_1 \ldots x_n$ are resident locations in DS rather than local variables.

A simple example of communication is given in Figure 2 where a task A unblocks a common process P known to be blocked on event name ℓ, and waits for a result from it. Here .U is a DS primitive which calculates an event name unique to the calling task, in this case task A. It does this by simply returning the absolute core location of the first system cell of the list defining the task. The primitive .WAT(ℓ) puts the executing task onto the blocked queue, waiting for event name ℓ. It is possible to put tasks at either the head or tail of the blocked queue, giving the queue a FIFO or FILO capability with respect to blocking and unblocking tasks with the same event name. Of course, this is important only when more than one task is blocked on the same event name.

A somewhat more interesting case is the mailbox scheme used to illustrate the IPC Facility of MULTICS[7]. A generating task A is asyncronously sending single word messages to a receiving task B, both understanding that there is a buffering task M blocked on eventname 'M'. The calling sequences are given in Figure 3. The buffering task is the "mailbox" which accepts messages from sender(s) and gives them on demand to a (presumably unique) receiver. The mailbox task is normally on the blocked queue, waiting for eventname 'M'. Its argument list is in the canonical form shown at left of Figure 4, namely an integer $N \ge 0$ followed by N messages. Either a sender or a receiver can activate 'M', and on execution the mailbox reduces itself back to canonical form, adjusting its argument list to reflect the message absorption or emission which occurred when it was activated. The program to do this is given in Figure 5. The Boolean expression integer $[x]$ is used in the program to distinguish messages from integers. INSERT and DELETE are examples of list manipulating DS primitives mentioned in Section V above, allowing cell exchanges between the argument list of the executing task and the small cell (free) list. Occasionally, either task A or B may fail to find the mailbox on the blocked queue because the other asyncronous process has just activated it. In this case, the communication is unsuccessful and the caller should wait until M becomes available (eventname 'ME' in Figures 3 and 5). The real time between the activation of M and the corresponding event 'ME' is the mailbox's dead time. It is interesting to note that absence of M from the blocked queue is equivalent to a raised semaphore variable s(P), in the sense of Dijkstra[8], signaling that a queue of asyncronous processes exists, each waiting enter a critical section P of program.

More elaborate communication situations can be handled, without significant change to the sender and receiver programming, by extending the mailbox program. Such situations include: management of a critical resource, such as the accelerator control relay output interface in the SLAC system mentioned in Section II above, the implementation of a many pronged, JOIN primitive, or even communication involving controlled access data retrieval.

VIII. Conclusion

Small computer multiprogramming systems can be very effective in applications areas where the dynamic data can be grouped into many relatively small lists, such as

those encountered naturally in message switching and line concentrator functions[5]. The paper shows that a large process control problem can be fit into this model by extending the communication between users and system through primitives. Once any system has been cast into this mold, the resulting message-like (i.e., linear) quality of the data structures can be utilized in a variety of ways. One interesting possibility would be to apply tools of the data communications field to these messages in a multiprocessing situation.

Acknowledgments

The author would like to thank Dr. Ken Mallory, who wrote the text editor, contributing many useful ideas along the way, and Vic Waithman who wrote most of the applications programs, insisting on the detail so necessary in an operational system.

References

1. K. Breymayer, T. Constant, K. Crook, J. Hall, T. Huang, D. Reagan, T. Sandland, W. Struven, "SLAC Control Room Consolidation Using Linked Computers," Particle Accelerator Conference, Chicago, Illinois, March 1971 (SLAC-PUB-866).

2. S. Howry, R. Johnson, J. Piccioni, and V. Waithman, "SLAC Control Room Consolidation - Software Aspects," Particle Accelerator Conference, Chicago, Illinois, March 1971 (SLAC-PUB-871).

3. Digital Equipment Corporation, Advanced Software System, Programmers Manual, Order No. DEC-9A-MACO-D.

4. Digital Equipment Corporation, Utility Programs Manual, Order No. DEC-9A-GUAB-D.

5. David L. Mills, "Multiprogramming in a Small Systems Environment," Second ACM Symposium on Operating System Principles, Princeton, New Jersey, October 1969.

6. Rajani M. Patel, "Basic I/O Handling on Burroughs B6500," Second ACM Symposium on Operating System Principles, Princeton, New Jersey, October 1969.

7. Michael J. Spier and Elliot I. Organick, "The MULTICS Inter-Process Communication Facility," Second ACM Symposium on Operating Systems Principles, Princeton, New Jersey, October 1969.

8. E. W. Dijkstra, "Co-operating Sequential Processes" (in: Programming Languages, NATO Advanced Study Institute, Edited by Dr. F. Genuys; Academic Press, London and New York, 1968).

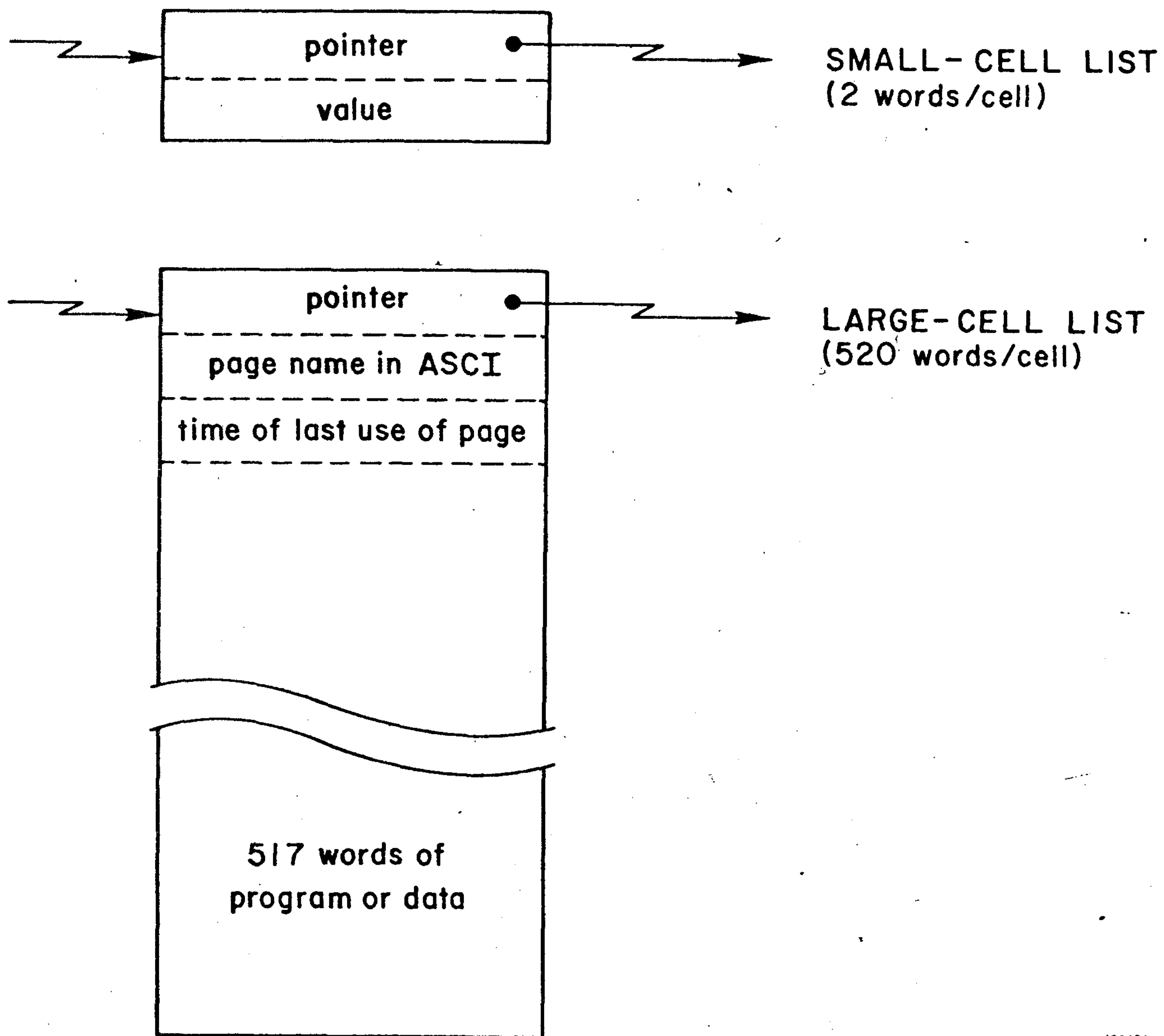


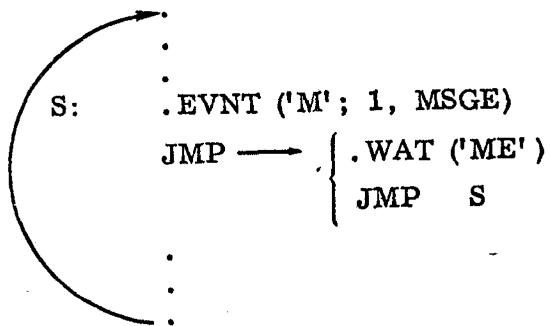
FIG. 1--Dynamic storage list structures.

A	P
<u>executing task code</u>:	<u>activated task code</u>:

```
A                               P
executing task code:            activated task code:

u ← .U
.EVNT (ℓ; 2, u, x)              PROC:  .WAT (ℓ)
JMP     NONE                                 .
    .                                        .
    .                                        .
    .                           (calculate f(x))
.WAT (u)                                     .
    .                                        .
    .                                        .
    .                           .EVNT (u; 1, f(x))
    .                           JMP     NONE1
                                JMP     PROC
```

FIG. 2--Simple inter-process communication.



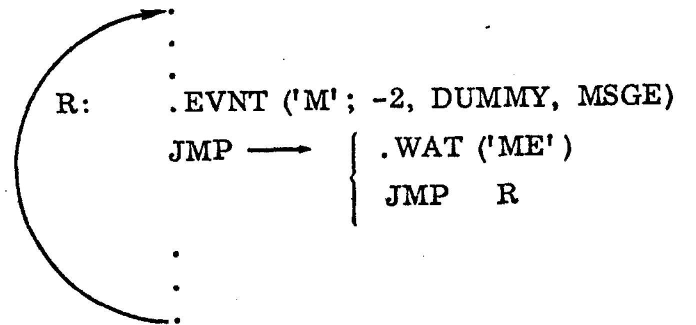


FIG. 3--Activation of mailbox.

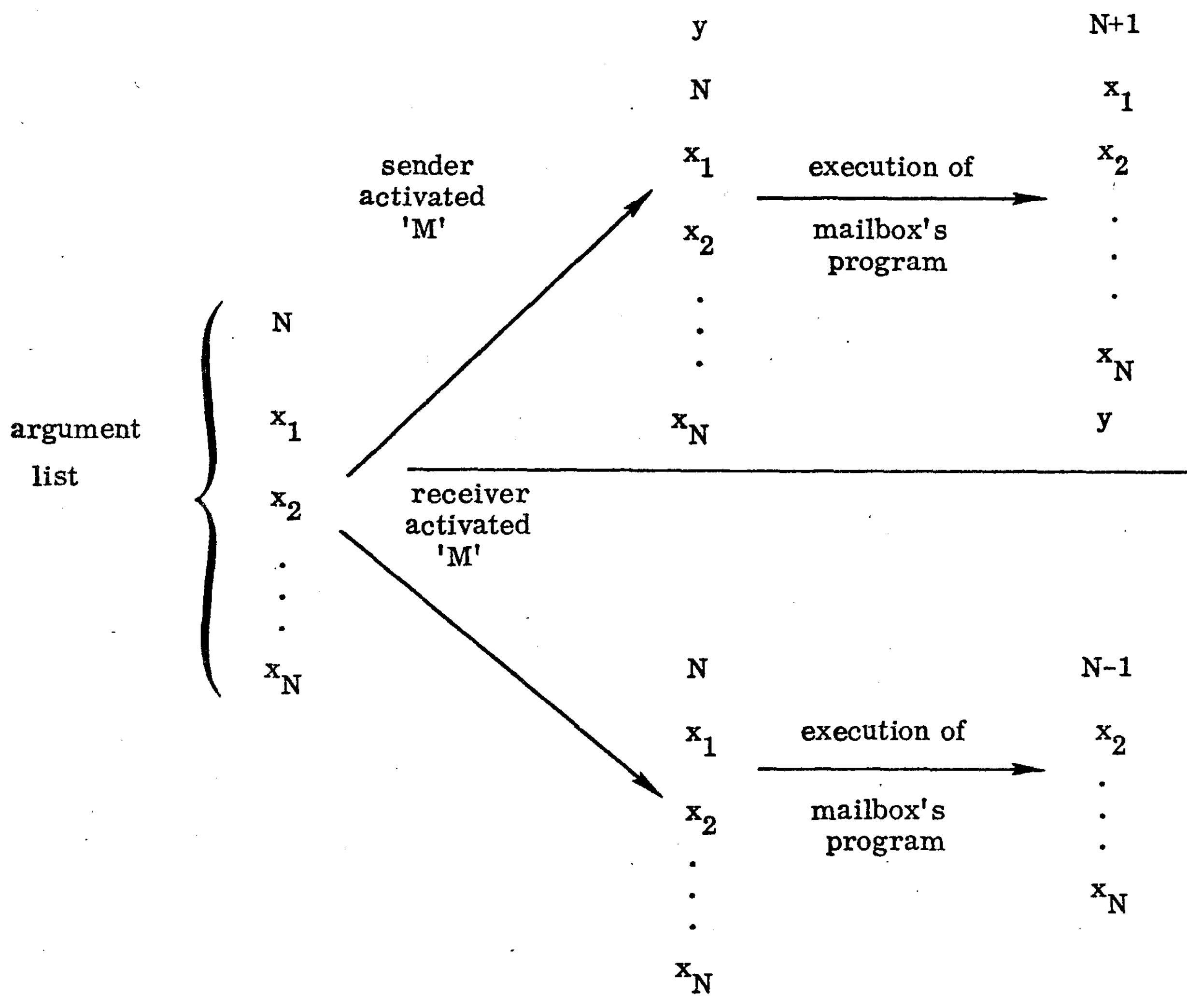


FIG. 4--Mailbox task states.

CODE FOR THE MAILBOX TASK

(conditional and assignment statements in ALGOL):

```
M1:   .WAT(M);
      if integer [arg1] then begin comment-receiver activated mailbox;

                  arg1 <-- arg1 - 1 ;
                  (DELETE arg2) end

      else begin comment - sender activated mailbox;

                  y <-- arg1          ;   comment arg1=message;
                  arg2 <-- arg2+1     ;   comment arg2=count;
                  n <-- arg2          ;
                  (INSERT AFTER argn: NEW CELL, VALUE y);
                  (DELETE arg1) ;

end;

.EVNT ('ME'; 0)  ;      comment--declares that the mailbox is ready;
NOP              ;
JMP    M1        ;
```

FIG. 5--Mailbox facility in DS system.