Published: 03/09/67
(Supersedes: BD.8.00, 08/29/66)

Identification

Overview of the Interprocess Communication Facility
B. A. Tague

Purpose

In order to permit parallel processing, each user job and many multics system tasks execute as collections of separate processes called process groups, rather than as single processes. This organization places a premium on effective and efficient control communication between processes. It is the purpose of the interprocess communication facility to establish such control communication. This section describes the facility--both the modules that comprise it and their general functioning.

Note to the Reader

The discussion that follows assumes familiarity with the notion of a process as it is implemented in multics, and with the multics file system. Some acquaintance with the traffic controller and interrupt handling facilities of the central supervisor is desirable. The structure of user process groups which perform various system chores is important to the discussion. A reading of overview sections of the MSPM in these areas should be adequate for the first part of this document which discusses general concepts.

The scheme of supervisor protection, the addressing hardware, and a more thorough knowledge of the file system are required in the section titled "Block Diagram of the Interprocess Communication Facility" and the sections that follow. These later sections describe the implementation of the facility.

Introduction

The basic mode of communication between two processes (refered to here as the sending process and the receiving process) is conceptually simple: the sending process places message information in a segment accessible to both itself and the receiving process; the receiving process reads the message information from the segment. In addition, the traffic controller provides two entry points,

block and wakeup, which make it possible for the receiving process to suspend operation while awaiting a message (by calling block), and for the sending process to cause the receiving process to resume running when a message is in the common segment (by calling wakeup). Several points of this basic procedure which are fundamental considerations in the design and use of the interprocess communication facility should be carefully noted.

First and foremost, communication depends upon the receiving process being explicitly programmed to receive a message. There is no interrupt primitive of a "now here this" variety available to the sending process. The call wakeup (process X) to the traffic controller merely causes process X to be rescheduled if it is currently blocked, and does nothing if the process is currently scheduled to run (ready), or running. Enforced communication can be accomplished only by enforcing the programming of calls to procedures in the receiving process designed to receive messages. A process may have its execution suspended by another process, but, on being resumed, the suspended process will continue to do whatever it was engaged in before suspension. If that does not include looking in interprocess communication segments, no interprocess communication will take place.

Another set of constraints on interprocess communication arises from the use of shared data segments for communication. Such segments must be known to both the sending and receiving processes, which implies a prior communication of the identity of the segment to be used between the processes. The segment used may be a programmed constant of the sending and receiving procedures in each process; the segment name may be passed to a process when the process is created; or, it may be passed in an interprocess message. In any case, there must be some communication between the processes which establishes those segments to be used for communication. Equally important, a sending process must have (at least) write permission for the segments used, and a receiving process must have (at least) read permission. (In practice, it will usually be necessary for both sender and receiver to have both read and write permission in order to operate interlocks which prevent conflicts between processes sharing the segment.)

Finally, there is the question of who is to be permitted to communicate with whom. Validation mechanisms must be established which permit a receiver some control over who can send him messages, but which also permit reasonable guarantees that communication important to system control can be accomplished.

Events and Event Channels

Fundamental to the interprocess communication facility is the notion of an event. An event is anything recognized during the execution of one process that is of interest to some other process, or perhaps some other procedure of the first process. For example, the completion of the task of collecting the characters of an input line from a typewriter is an event which might be recognized by a device manager process and be of interest to a working process. An event is a unique occurrence; it happens exactly once. If the device manager process of the example recognizes several successive line completions, each completion would be a separate event. The primitive interprocess message is a signal from one process to another that an event has occurred.

Events are signalled by the interprocess communication facility over event channels. Conceptually, an event channel is simply a first-in-first-out queue of bit strings associated with a process that wishes to receive event signals. A bit string called an event id is added to the channel queue by the sending process each time it signals an event. The event id is a unique bit string generated by the sending process for each event signalled. In the example of a device manager process signalling line completions to a working process, a sequence of ten completed lines would cause the device manager to signal ten completion events; each event would be represented by a unique event id in the channel queue. Moreover, the event ids generated could also be stored with the respective lines that generated them in the line buffers, which would permit the receiving process to discover the order in which the lines were typed by reading the order of event ids in the channel queue. The receiving process can read the event queue of a channel by executing a call which returns the queue of event ids and resets the queue to zero length.

An event channel always has exactly one receiving process, the process that creates the channel, but the same event channel may be used by several sending processes. A process may create as many event channels as it needs. Usually, a different event channel would be created for each kind of event whose occurrence is to be signalled.

A module called the event channel manager is provided to maintain event channels. It provides calls which create event channels, which place event ids in event channel queues, which read and reset event channel queues, and which delete event channels no longer needed.

Event Channel Types and Modes

A process must specify three things when creating an event channel:

which processes are permitted to signal over the channel, the sending processes;

the channel type:

the channel mode.

There are three event channel types: intrauser channels, interuser channels, and device signal channels. There are also three event channel modes: event queue mode, event count mode, and event time mode.

The channel type controls which processes have access to the channel by determining the data bases in which the channel data is stored. The data base in which the channel data is stored, in turn, determines the protection ring (or rings) where the channel data is stored. Also, different data bases have different processes on their access control lists. Intrauser event channels only permit communication between processes of the same process group since the table containing intrauser channel data is accessible only to this user process group. The sending process id specified for an intrauser channel must belong to the same process group as the receiving process that creates the channel. If the sending process id specified is zero, any process of the group that knows the channel id and its receiving process id can signal over the channel. Interuser event channels permit communication between any two processes. If the sending process id specified is zero

any process at all that knows the channel id and its receiving process id can signal over the channel, however, the device signal channels are very special. They are maintained in "wired down" core areas for the purpose of signalling by hard core procedures that cannot tolerate the page faults that are possible in referencing the paged data bases in which other types of event channels are kept. Because they are in "wired down" core, it is important that they not consume much space; therefore, they do not queue event ids. They always operate in the event time mode described in the next paragraph.

The mode of a channel determines the kind of signals sent over the channel. In event queue mode, the signalling method is that described earlier: each call to signal over the channel causes an event id (including time of occurrance) to be appended to the channel's event queue. Resetting the channel empties the queue. Since the queueing of event ids requires space and time, the receiving process may specify the event count mode, in which case an event count is kept rather than an event queue. Each call to signal over the channel increases the event count by one. Resetting the channel zeroes the count. As another alternative, the receiving process may specify the event time mode, and a single clock time replaces the event queue. A call to signal over the channel stores the current clock time in the channel. Resetting the channel resets the time to zero. If several signals are sent before the channel is reset, only the first signal stores the time. The event time mode is always the mode of device signal channels.

Process Synchronization

The basic purpose of interprocess communication is to coordinate the computations of independent, but related, processes. A typical situation is presented in the following example: a working process wishes to request output of the results of the first part of a computation and continue with the second part of the computation while the I/O is being done, but does not want to request output of the results of the second part of the computation until the first I/O request is completed. The user process group has a separate device manager process for just such a purpose. The device manager process can supervise the execution of I/O requests while the working process continues with other computations.

Two event channels are used to synchronize the two processes of this (simplified) example. One event channel would be created by the device manager and its channel id given to the working process with the understanding that an event in this channel will be interpreted as a request for some I/O by the device manager. A second channel would be created by the working process and its channel id given to the device manager with the understanding that an event in the channel will be interpreted as signalling the completion of the requested I/O task by the device manager.

The creation of the two event channels does only part of the desired job. The working process wants to stop computing and wait for the first I/O request to be completed before placing a second I/O request. This requires that the working process call the block entry of the traffic controller. The block entry is not accessible in the user ring (it can be called only from the administrative and hard core rings); therefore, the working process would like to be able to call a procedure which would result in a call to block. Similarly, the device manager process needs to be able to call wakeup. Moreover, there is a need for a procedure to coordinate the wakeup sent by the device manager with the event in the channel used to signal I/O request completion.

The coordination of calls to block and wakeup with event channel signalling is the responsibility of the wait coordinator module of the interprocess communication facility. It provides two basic calls for user programs. First, there is a call to be executed by the sending process that signals an event over a designated event channel, and then sends a wakeup to the receiving process for the channel. Second, there is a call to be executed by the receiving process that stores a list of event channel ids and calls block. Any wakeup signal received when the process is blocked this way in the wait coordinator causes the process to check the event channels on the stored list. If any channel has an event id in its event queue, the procedure returns to the caller. Otherwise, it calls block again. Remember that a process may be awakened by a signal over a channel on which it is not currently waiting.*

---

*This description of the wait call is simplified for clarity. In order to be sure that no wakeups are missed, the event channels must also be checked before block is called. See section BO.6.05 for details.

Interprocess Calls

Consider again the example of a device manager process running, say, a typewriter. Such a process will typically spend a large part of its life in the blocked state waiting for either a new request for I/O from a working process, or an interrupt from the GIOC announcing some communication from the typewriter. Both of these kinds of signals will be received via event channels. A device manager will typically have a large number of event channels for which it is either sender or receiver, and when it goes blocked it will be waiting upon signals from many different channels. When it is awakened, the device manager process will want to execute different procedures depending upon which event channels have nonempty event queues. A natural way of looking at a signal sent to a device manager process is as an interprocess call--a request from the sending process that a particular procedure be called by the receiving process. The interprocess communication facility provides two kinds of interprocess call: the event call and the givecall.

The event call mechanism is part of the wait coordinator module. A call to the wait coordinator associates an event channel with a procedure name supplied by the caller in a table maintained by the wait coordinator. Whenever an event is signalled over that channel and the process is waiting (i.e. has called block via the wait coordinator), the process wakes up in the work coordinator which calls the procedure associated with the channel in the table. The procedure is called with the event channel id as the only argument. The calling process must place any other data needed by the receiving process from the caller into a data segment accessible to the receiver. The procedure called can be programmed to look in that shared data segment. The event call offers limited service at low overhead.

The givecall is handled by a module called the call passer. It is simpler to use, but involves higher overhead than the event call. The caller supplies the id of the called process, the name of the called procedure, and its argument list. The call passer places the symbolic procedure name and argument list (actually modified pointers to the arguments) into a special data base associated with the call passer

module of the called process. It then signals an event over a channel whose id is found in the data base used to pass arguments. Normally, a procedure expecting a givecall will be waiting for a signal over that channel. When the called process wakes up, it calls a procedure of the call passer which takes the procedure name and arguments out of the data base, constructs a call to the named procedure, and executes it. On return, the call passer signals an event over an event channel created for that purpose by the givecall procedure of the calling process. The caller, after issuing the givecall, can either continue computing or wait on this event channel for the return signal.

A primary use for the givecall is process initiation. When a new process is created, it is laid down in the blocked state ready to receive a givecall from its creator. This enables a process that spawns a new process to control the procedure that is first executed by the new process. Among other things, the initial givecall to a newly created process can pass as arguments the names of data segments to be used in future interprocess signalling.

Block Diagram of the Interprocess Communication Facility

Figure 1 shows the modules and data bases of the interprocess communication facility, their positions with respect to the rings of protection, and the paths of possible calls. The diagram also includes the traffic controller in order to show its relationship to the facility. The preceding paragraphs have mentioned the three modules in the administrative ring: the event channel manager, the wait coordinator, and the call passer. The discussion which follows will outline the functioning of those modules and discuss their associated data bases. It will also treat the modules in the hard core ring which have not yet been mentioned.

The interuser event table manager and the device signal table manager can be looked upon as extensions of the event channel manager. They manipulate event channels in the hard core ring. The interuser event table manager creates and maintains interuser event channels used in communication between process groups. Since different process groups belong, in general, to different users, care must be taken lest improper signalling by a process in one group cause trouble in another group. Signalling between processes fo the same group can be carried out

in the administrative ring because improper signalling within a group can only disrupt one user. Adequate protection for signalling between groups requires that the channel data be maintained in the hard core ring. The device signal table manager is in the hard core ring for these same reasons, but is separate from the interuser event table manager because the channels it manages are a special type. In particular, they reside in "wired down" core. This means that some of the procedures which maintain the channel data must also reside in "wired down" core.

Event Channel Creation

A process that wishes to be notified of some event, a receiving process, must create an event channel over which it can be signalled. This is accomplished by a call to the event table manager giving the channel type, the channel mode, and the sending process id. The event table manager examines the type of channel and determines whether it is to maintain the channel data in the administrative ring (intrauser channel), or whether the channel data is to be maintained in the hard core ring by the interuser event table manager (interuser channel). (The device signal channel is a special case which will be discussed later.)

If the channel is an intrauser event channel, the event table manager:

1. Obtains a unique bit string to be used as channel id.

2. Makes up an entry in the event table which contains a pointer to the first entry of the event queue, the sending process id, the channel mode, and other data associated with the channel.

3. Makes up an entry in the event channel index consisting of the event channel id, the channel type, and the location of the channel data entry created in step 2.

4. Returns the event channel id to the caller.

Both the event table and the event channel index are per process data bases. The event table has one entry for each intrauser event channel for which the process is the receiving process. The event channel index has one

entry for each channel for which the process is receiver. Any event channel which the process is to wait upon must be represented in the event channel index for the process--that is, a process can only wait for signals over channels for which it is the receiving process. Both tables must be accessible to a sending process in the administrative ring. Given only the receiving process id, a sending process can construct the name of either table in order to gain access to it. Naturally, the sending process must have read and write permission in the administrative ring for the segments involved.

If the channel being created is of the interuser type, it is maintained in the hard core ring by the interuser event table manager. The event channel manager calls that module to perform steps 1 and 2. The channel id generated in step 1 is returned by the interuser event table manager to the event channel manager in the administrative ring which performs steps 3 and 4 as in the intrauser case. The entry described in step 2 is made up in the interuser event table becuase the entry must be accessible to a sending process of another process group. The interuser event table contains one entry for each interprocess event channel for which the process is receiver. It is accessible to any sending process, but only in the hard core ring. Given only the receiving process id, a sending process can construct the name of the receiver's interuser event table in order to gain access to it.

Channel Creation in the Hard Core Ring

There are two kinds of channel that can be created without leaving the hard core ring: an interuser channel, or a device signal channel. The interuser channel is created by a direct call to the interuser event table manager. This call sets up the channel just as if the create call had come from the event channel manager in the administrative ring. The channel id and the location of the channel data in the interuser event table are returned to the hard core procedure creating the channel. These two items can then be passed to an administrative or user ring procedure which wishes to wait on the channel. This procedure must call the event channel manager with the channel id and location of the channel data as arguments to create an event channel index entry for the channel. The process may now wait upon the channel like any other.

While having a channel created in the hardcore ring is optional for interuser event channels, it is the only way a process can use a device signal channel. A device signal channel always exists in the sense that the space for the channel data is allocated in "wired down" core at system initialization time. A process wishing to use a device signal channel must have been handed a device index by a hard core procedure that controls the channel data space located by the device index. This usually occurs because the process is being given control of some peripheral device. If the process is to wait on the device signal channel, the event channel manager is called to make up an event channel index entry for the channel. The event channel manager is given the device index, which is the location of the channel data in the device signal table, and it returns an id for the channel after making up an entry in the event channel index. The process can now wait upon this channel like any other.

Event Channel Signalling

In order to signal over an intrauser or interuser event channel, a sending process must be given a channel key by the receiving process which created the channel. A channel key is an array which consists of the channel id and the receiving process id. It is stored by the receiving process in a segment accessible to the sending process or processes; it must be supplied as argument to the event channel manager when an event is to be signalled. The event channel manager uses the receiving process id to determine whether or not the receiving process is in the same process group. If it is not, the channel must be an interuser event channel. If it is, either the channel is an intrauser event channel, or a process of the group is signalling over an interuser channel (it is convenient to permit this). Either way, the event channel index of the receiving process is accessible to the sending process and the channel type can be obtained from there.

In the case of an intrauser channel, the sending process event channel manager:

1. Retrieves the event table index of the receiving process and looks up the entry for the event channel id.

2. Obtains the channel type and channel data location from the event channel index entry.

3. Retrieves the event table of the receiving process, and looks up the channel entry, provided that the event channel type is intrauser.

4. Updates the event table entry as required by the channel mode, provided that either the sending process id of the entry is that of the executing process, the sending process id of the entry is zero, or the executing process is the receiving process of the channel.

5. Returns the event id, event count, or event time generated by the update to the caller.

6. Calls wakeup for the receiving process if requested by the caller.

In the case of an interuser channel, the event channel manager calls the interuser event table manager, passing the event channel id and receiving process id as arguments. The interuser event table manager performs steps 3 through 6 as given for the intrauser channel above, except that the interuser event table is referenced in steps 3 and 4. Steps 1 and 2 are unnecessary because the channel id can be used to locate the channel data in the interuser event table. Also, the event channel manager passes the argument returned in step 5 back to its original caller.

The wait coordinator can be called to signal an event rather than calling the event channel manager directly. This results in precisely the same action, but a list of channel keys can be passed in a single call. In addition, exactly one wakeup will be sent for each different receiving process on the list of channels.

Signalling over device signal channels is restricted to hard core ring procedures. The reader will find a discussion of how a hard core procedure sends signals over a device signal channel in section BQ.6.04.

Reading, Resetting, and Deleting Event Channels

In order to read an event channel, the user calls the

event channel manager with the channel id as argument. The event channel manager looks up the channel id in the event channel index of the process. The channel type tells the event channel manager whether it can read the channel directly from the event table (intrauser channel), or whether it must call the interuser event table manager (interuser channel) or the device signal table manager (device signal channel) to read the channel data from the hard core ring tables. Depending upon the channel mode, the event queue, count or time is read from the channel data. The caller also indicates whether or not the channel is to be reset after reading.

Event channel entries are deleted from the event channel data bases in either of two ways: by a specific delete call to the event channel manager, or by the demise of a process which causes the demise of its per process data bases.

Only the process which creates a channel can delete it. Device signal table entries are not deleted, they are merely reallocated to different uses; however, the event channel index entries for device signal channels are deleted by either delete calls or by the demise of a process.

Event Call Table

The event call table is a per process table maintained by the wait coordinator module. It contains one entry for each event call currently in existence for the process. Each entry contains an event channel id and a procedure entry point. Whenever control returns from block to the wait coordinator because of a wakeup, each event channel that is listed in an entry of the event call table is read out (but not reset). If the channel contains any signals the procedure listed for that entry is called.

Note that the event call table provides a way of storing all those event channel ids on which a process is currently expected to wait which the user whould prefer not to have to explicitly list in wait calls.

The Work Queue and the Call Passer

The work queue is a per process data base maintained by the call passer. When a process is created, its work

queue contains only an event channel id and is accessible to the creating process for both reading and writing in the administrative ring. A signal on this channel causes the created process to execute a procedure of the call passer named accept_call. If the creating process issues a givecall, a procedure name, and an argument list for that procedure are placed in the work queue of the created process and an event is signalled over the channel whose id is at the head of the work queue. This causes accept_call to call the procedure with the arguments supplied, and to send a signal over an event channel supplied by the caller when the called procedure returns.

Note that there are two event channels involved in signalling for a givecall. One channel is created by the called process when that process is created. Its channel id is the first item in the work queue of the called process. It is used by the calling process to signal that the arguments for an interprocess call have been placed in the work queue. The other channel id is placed in the work queue of the called process by the calling process along with the arguments of the call. It is used by the called process to signal the return from the call to the calling process.

A process can issue a givecall to a second process at any time the work queue of the second process is accessible. Repeated givecalls result in an ordered list of calls in the receiving process work queue, and a corresponding queue of event ids in the event channel whose id is at the head of that work queue. Repeated execution of accept call will clear the work queue by executing the calls in order; however, the work queue can also be cleared without executing the stacked calls by resetting the event channel.

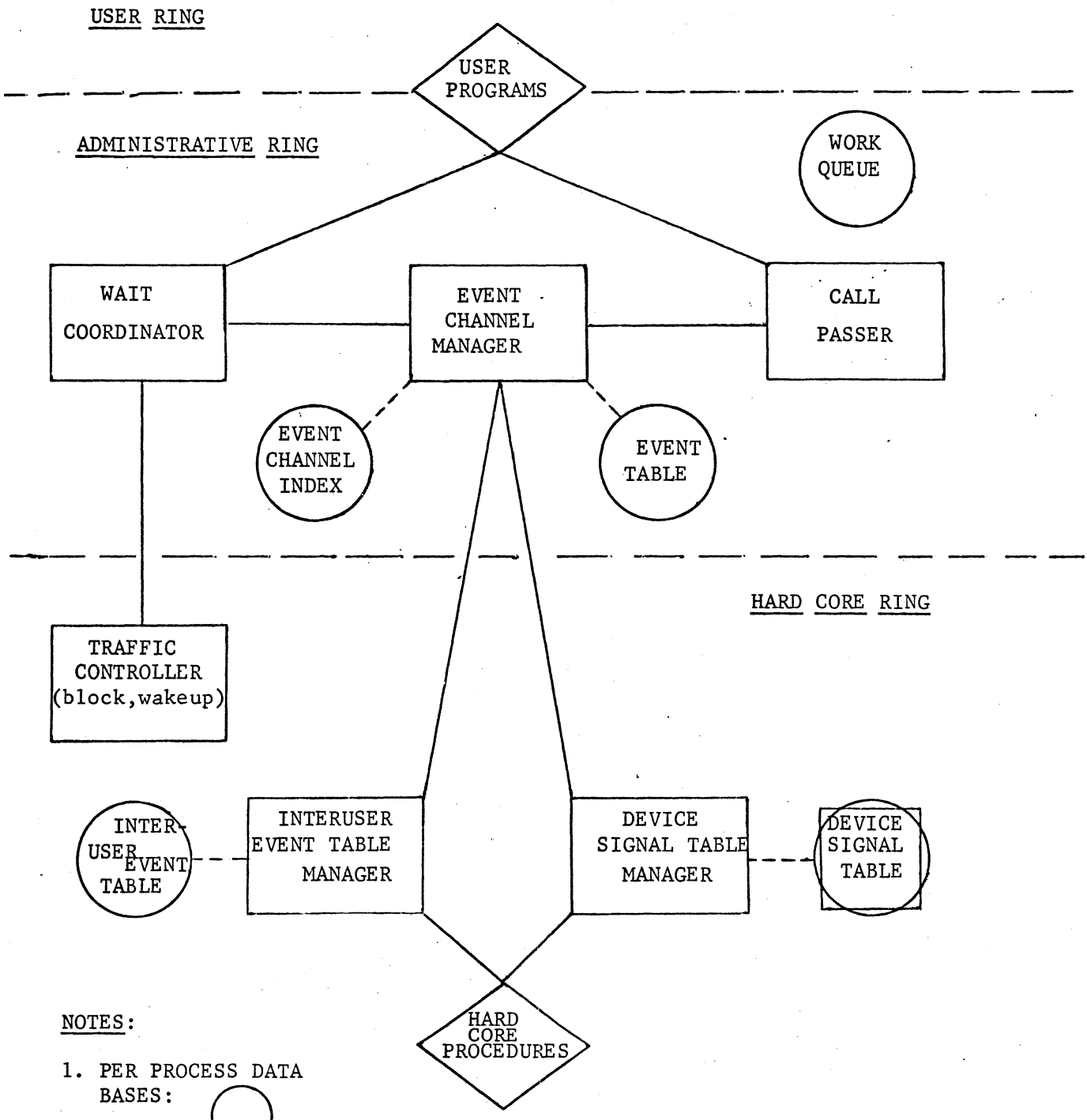

FIGURE 1. BLOCK DIAGRAM OF THE INTERPROCESS COMMUNICATION FACILITY