M0114
July 29, 1969

THE MULTICS INTERPROCESS COMMUNICATION FACILITY*

Michael J. Spier
Massachusetts Institute of Technology
Project MAC

and

Elliott I. Organick
Visiting Professor of Electrical Engineering
Massachusetts Institute of Technology

Summary

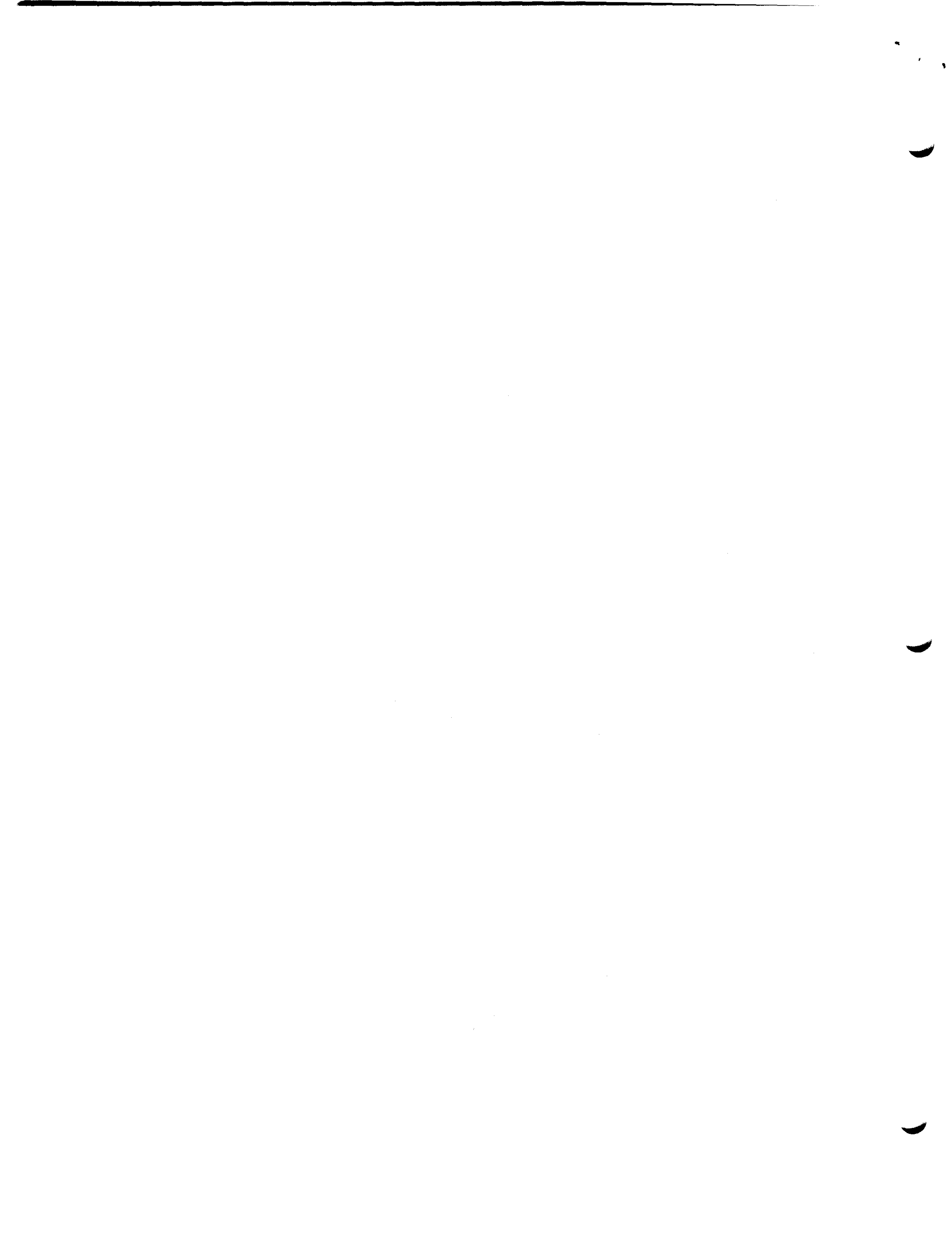
Essential to any multi-process computer system is some mechanism to enable coexisting processes to communicate with one another. The basic inter-process communication (IOC) mechanism is the exchange of messages among independent processes in a commonly accessible data base and in accordance with some pre-arranged convention.

By introducing several system wide conventions for initiating communication, and by utilizing the Traffic Controller it is possible to expand the basic IPC mechanism into a general purpose IPC facility. The Multics IPC facility is an extension of the central supervisor which assumes the burden of managing the shared data base and of respecting the IPC conventions, thus providing a simple and easy way for the programmer to use the interface.

The following paper describing the implementation of the Multics Interprocess Communication Facility is a preprint of a paper to be presented at the Second ACM Symposium on Operating System Principles, to be held in Princeton, New Jersey in October, 1969.

---

* Work reported herein was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.



1. Introduction

This paper describes the Inter-Process Communication (IPC) facility which was developed for the Multics (Multiplexed Information and Computing System) system and which is part of the latter's central supervisor program. Not unlike other central functions of the supervisor, the IPC facility began with an implementation, followed by successive redesign/re-implementation cycles during which we gained increased understanding of the function and found an increasing number of applications for it.

An IPC capability of one form or another must and does exist in any multi-programmed computer system. It is an essential requirement for any parallel-processing computation, as the examples below will suggest.

There is a marked trend in modern system design towards a multiplexing of most hardware resources to the end that a single job (task, computation) may behave, to all outward appearances, as if it were an independent computer. In this paper we refer to such a virtual (pseudo) computer as a process. Later, we shall present our definition for the term "process". For the present we shall bootstrap by using this term to present a preliminary sketch of IPC and its uses.

An important characteristic of a process is its degree of independence of fellow processes. Full independence may easily be achieved by containing all operations of a single-purpose job (task, computation) within a single self-sufficient process. Such an implementation, however, tends to be wasteful in terms of CPU or other resource usage. An example would be a computation that includes slow input output (I/O) steps. A more efficient implementation would be achieved if the computation merely initiates its I/O requests letting the actual I/O operations be executed by some other, perhaps dedicated process. This arrangement enables the computation to proceed to the point at which it actually needs the result of the I/O operation. At this point the computation (i.e., the

first process) should be able to wait for the notification that the awaited event has occurred. Moreover, for efficient CPU resource management, the first process should be able to abandon its hardware processor until the event occurs. This type of implementation of a single computation involves two cooperating processes.

Another example is that of two processes which operate on a single buffer. Process A puts items into the buffer; process B removes items from it. By virtue of their independence of one another, no assumptions about the relative speeds of process A and process B should be necessary. Thus if process A is "faster" then process B, the former will eventually fill up and overflow the buffer. If, on the other hand, process B is the faster, a point will be reached when B will attempt to extract a non-existant item from an empty buffer. This problem is easily solved by making process A wait for the "buffer not full" event whenever the overflow condition is detected, by making process B wait for the "buffer not empty" event whenever the buffer is empty, and by making processes A and B notify one another of the "buffer not empty" and "buffer not full" events respectively when it is appropriate that they should do so.

In the foregoing example one process is permitted to execute asynchronously of the other, to the extent that its relative independence allows. Cooperating processes synchronize or coordinate their activities only when necessary (i.e., when they enter their respective "critical sections", as Dijkstra has put it[1]). Now picture an arbitrary computation that involves n (≥2) cooperating processes. What system-provided facilities can be provided that will assist the designer of such a computation to effectively

program the required coordination among these n processes when n is large? This is the problem that has been attacked and effectively solved in the Multics IPC effort. The current solution, albeit one of perhaps many, will hopefully be of interest to the reader of this paper.

As evolved in Multics, the IPC capability is now a completely generalized, and effective modular unit. This development was made possible by the availability, in the Multics system, of certain other capabilities, namely the ability to freely share data bases among (or protect them from) different users, the ability for several of them to access such a shared data base by referring to it under a single symbolic name[6], the ability to achieve mutual exclusion among competing computations and the availability of efficient processor multiplexing capabilitiss. In the past the problems involved in providing all those functions may have seemed so insurmountable that they may have completely obscured the actual issues of IPC and may have prevented a general purpose IPC function from ever evolving beyond the stage of ad hoc implementations.

The Inter-Process communication facility is itself oblivious to the type of, or reason for, the communication that it supports between processes. The facility provides wait and notify services. It does so without ever compromising a process' independence, without restricting the number of possible events, without restricting the number of communicating processes and without differentiating between types of events. It works for hardware-originated events (e.g., interrupts) as well as for software originated (e.g., "buffer not empty/not full") events. This last ability has proven to be useful because it allows the intermixing of hardware and software originated events so as to give them a common

interpretation without ever actually having to know their origin. An example taken from the actual Multics implementation is the "process termination" event which may be triggered by any of the following causes: the logout command, console hangup (power off), general system shutdown, fatal error condition within the same process, out of funds condition, preemption by a higher priority user, etc. Any of these occurrences causes a dedicated system process to be notified that a "process termination" event has occurred for a given process. The 'et cetera' following the list of occurrences implies that the list is by no means closed. Any other condition may be added to that list in the future without affecting, in any way, any of the existing implementations of this event.

2. Background

Before proceeding, we offer several background concepts and definitions to explain the framework (Multics system environment) in which the IPC operates.

The concept of "process" is an abstraction. It has been defined in different ways [2,3,4,5,8] depending upon the computer and information system model one wishes to explain. We begin here with a definition for process which, though perhaps abstract to an extreme, may ultimately facilitate development of the intercommunication model that is wanted. We way that a process is a discrete progression, in time, of discernable changing states. If each discernable state change is called an event, we may abbreviate the foregoing by saying that a process is a (discrete) sequence of events[4]. Two processes normally communicate with one another when some of the events of one are, or may be, of interest to the other[1,4,8,9]. Our definition allows us to think of the process as being an abstract execution point whose progression manifests itself in the form of changing states which may be observed (i.e., events). The word discrete suggests that the observer is free to choose different units ("grains") of time.

It is convenient to think of a multi-processing system (one that distributes its resources over a group of processes) as a collection of cooperating processes that exist within a special world, or private universe. This approach allows us, later on, to talk about a universe that is external to that of a multi-processing system.

To the programmer, the notion of process is (intuitively) easily understood to be the equivalent of "program in execution"[5]. This maps nicely into the above definition if we say that the "grain of time" is a hardware

instruction cycle and if the states which we observe are those of the hardware registers. Let us name it a "hardware-level" process. Alternatively, we may choose a larger grain of time. An example is a program coded in some high-level language (e.g., FORTRAN or PL/I etc.) wherein, as far as the programmer is concerned, the process' progression is of interest only as it passes from one source statement to another, and where the observations relate to changes in the values of source-language variables.

In the Multics system, the word "process" has a well defined meaning. It is a "hardware-level" process whose address space is a collection of named segments, each with defined access, and over which a single execution point is free to "roam" i.e., fetch instructions and make data references [3,4,6]. A central supervisor module sees to it that at most one execution point is ever awarded to an address space.

Interprocess communication implies an exchange of data communications among cooperating processes; this data exchange must take place in a shared data base and implies read/write access in that data base for all communicating processes. By design, sharing a data base (for whatever reason) presents no difficulty in the Multics system. After agreeing on the segment to be shared for storage and retrieval of messages each of the cooperating processes is free to reference the segment by its distinct file system name. Such a reference causes the same segment (same physical copy in primary memory) to be accessed by each process[6].

To control multi-process access to a single data base and to guard against reference to data that may be briefly in inconsistent states, there is need for a locking mechanism to insure mutual exclusion among potentially interfering processes[1,5,8]. In this paper, whenever we discuss shared data bases, we assume that availability of such a mechanism (later referred to as functions lock and unlock), based on a hardware test and set instruction, and its application whenever necessary.

IPC is conceived, primarily, as a standard means of communication among processes. However, a process which is engaged in input output operations (I/O) is, in fact, communicating with some independent dedicated hardware processor (commonly known as an I/O Channel) which is external to the process' universe. I/O channels are capable of producing signals indicating completion or trouble. The signals are transmitted to the system in the form of processor interrupts (device signals). We may think of the processor and the I/O channel as being two cooperating processes which exist in a common universe.

Device signals are unpredictable and may arrive at any given time to interrupt any currently-executing process. Logically, however, they "belong" to the process that is currently "responsible" for the interrupting I/O channel.

In a multiplexed computer system, it is of interest to be able to distinguish between the concepts of "hardware" processor and "virtual" (i.e., pseudo, or software) processor[3,4,8]. A program that is written for a computer (as opposed to process) may make assumptions about the (hardware) processor's speed of execution. If, however, the program is written for a process, even though the actual execution is done by the very same hardware

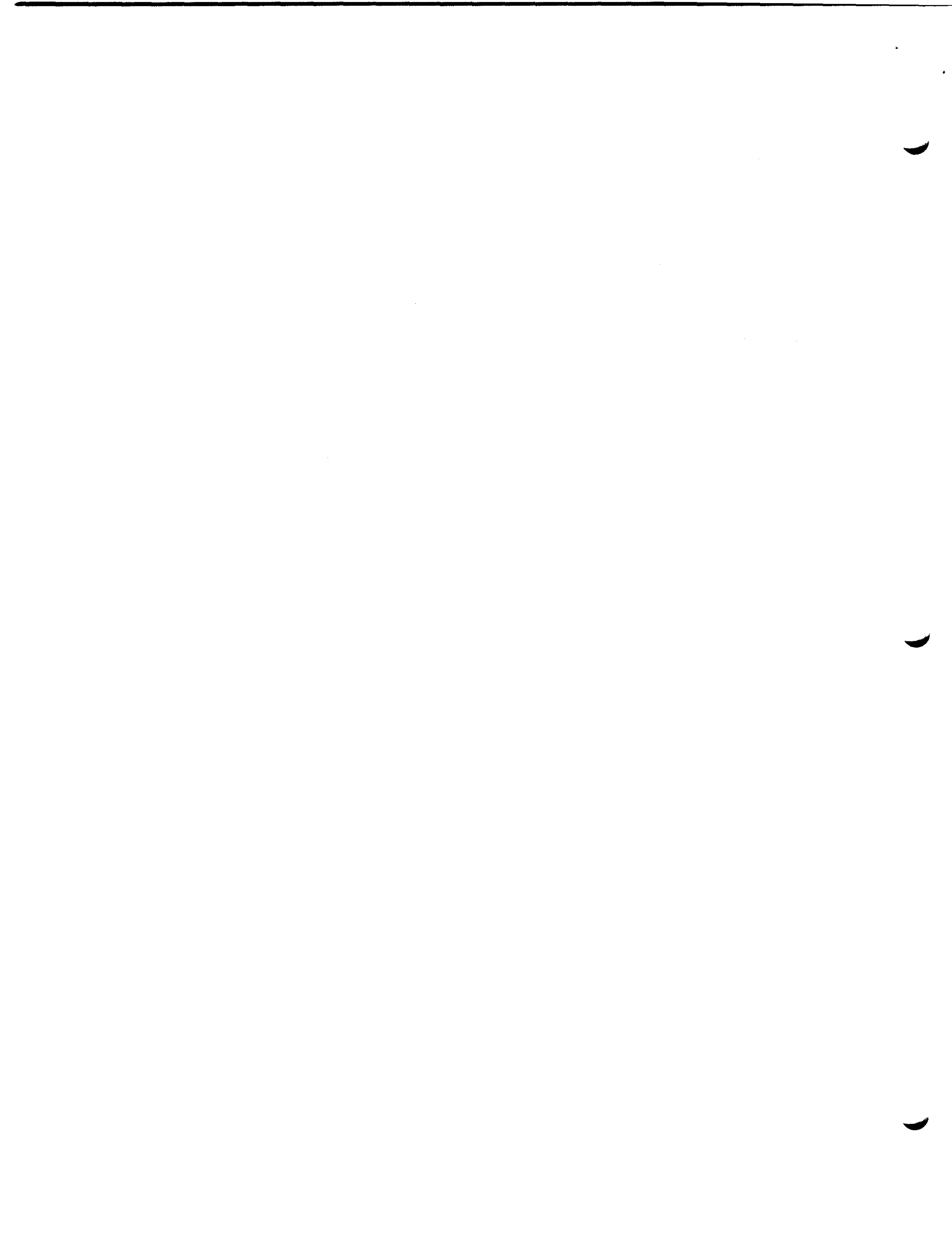
processor, the process is not guaranteed to keep that processor for any predetermined length of time. Consequently, the time gap ("grain of time") between the execution of any two consecutive machine instructions is unpredictable.

Central to Multics is a module called the Traffic Controller[3,4] under whose control independent computations may compete fairly for (i.e., share) the computer resources of the system. But, more importantly for consideration here, the Traffic Controller also provides block/wakeup services. Block and wakeup are general functions that govern the transmission of control communication which, for example, permit two or more processes to cooperate or synchronize their activities[8]. The block function is invoked by a process to force its virtual processor to pause. The wakeup function is invoked by some other process in order to restore to the (possibly blocked) first process its execution capabilities.

The job of the Traffic Controller is to make the multiplexed system look like a dedicated processor system by creating one virtual processor for each process. The virtual processor may be thought of as "always executing"[4]; from its point of view the effect of the block function is to insert an unpredictably large "grain of time" between two events.

Although a process is always "conscious" of giving up the processor when (and because) it invokes the block function, the mechanics of processor switching, apart from real-time "clock jumps", are essentially invisible to the affected process. This means that a process can completely ignore the multiplexing being performed by the Traffic Controller. In the same

spirit of modularity, the Traffic Controller of Multics, unlike its counterpart in the forerunner, CTSS[10], is itself unconcerned with the origin of the requests for its block/wakeup services, e.g., inter-console messages, input-output completion signals, alarm clock services, etc.



3. Fundamentals of IPC

A limited number of basic (implementation-independent) and, for the most part, self-evident considerations form the foundation of IPC. We survey them here.

Communication among processes that coexist within a private universe can only be achieved by an exchange of messages in a commonly accessible mailbox (shared data base) whose identity is known to each by common convention. Moreover, in order to exchange meaningful messages, an added convention is necessary by which communicating processes may interpret the state of the mailbox.

The mechanism by which, within some universe U, some process A may send a message S_a to some other process B is summarized in the following two stages:

ipc-setup (stage 1)

a) By convention, both processes gain knowledge of a commonly accessible mailbox M; for example, by its distinct file system name.

b) By convention, both processes agree to interpret a specific state S_o of M to imply "mailbox is empty". The mailbox should be initialized accordingly.

basic-mechanism (stage 2)

a) Process A sets the state of M to some state S_a which is different from S_o.

b) Process B interprets any non-S_o state of M to imply "message has arrived". Depending upon its convention with A, process B may or may not then consider state S_a to be meaningful.

We observe that in order for the processes to be able to communicate at all, there is a need for some "previous communication", or convention, which we name ipc-setup. In a multi-processing (computer) "universe", the ipc-setup normally consists of conventions that are made at program-coding time among the authors of interactive programs (often, the convention is established by the single author of such programs), to the effect that an unambiguous (symbolic) address is used to reference mailbox M. Also, a value is pre-assigned at coding time to state S_o or perhaps to one or more of the non-S_o states. Once an initial ipc-setup has been achieved, additional ipc-setup's may be achieved by possibly re-using the same mailbox M and by transmitting in it (in either direction) the names of additional mailboxes. This is one application in which the specific values of non-S_o states becomes meaningful.

As a rule, the ipc-setup must occur in a universe that is "external" to (and perhaps "containing", or "responsible for") universe U in which the communication is to take place. Admittedly, the term "universe" is a little vague, and the interrelationships mentioned (external, containing, responsible) are even less meaningful. We do, however, wish to convey the idea that an ipc-setup cannot occur within the same framework as its associated communication; rather, that every IPC communication depends upon an ipc-setup that has happened previously, and that the "universe" in which the ipc-setup was made is in some difficult-to-define way the "ancestor" of universe U.

The actual interprocess communication is achieved by exchanging messages in mailbox M. The basic mechanism shown above is the most primitive and elementary way for processes to communicate; it allows only for a single, one-way message transmission. As will be seen, the basic mechanism may be

expanded and made more useful by the introduction of additional conventions among the communicating processes. Thus, the design of a general-purpose IPC facility has largely to do with the establishment of useful conventions.

A familiar example of a very primitive IPC mechanism (though slightly more sophisticated than the basic mechanism in that it allows the repetitive transmission of one-way messages), is that by which an I/O channel sends signals to interrupt a processor (CPU) when serving notice of an I/O completion or an I/O error. The channel and the CPU both share a mailbox (a special, but commonly accessible memory cell) whose location in this case is predesignated in the hardware (ipc-setup). (Also predesignated is another cell for holding a transfer instruction for sending control to an appropriate interrupt handler.) The channel sets one or more bits in the mailbox (the message) to indicate the nature of the event being signalled, and concurrently sets a flip flop which is interrogated by the CPU at the end of each instruction cycle, which is the "grain of time" in this case. If the CPU finds the flip flop has been set (non-S_0 state), the interpretation is, "I have been interrupted by the I/O channel."

The basic IPC mechanism places no a priori restrictions either upon the size of the mailbox or on the amount of information that may be contained in it. An actual IPC facility, such as the one in Multics, may be designed as an <u>event oriented</u> mechanism for the communication of control items, rather than as a general-purpose clearing house for bulk data exchange. Thus, the Multics IPC has restricted data transmission capabilities, i.e., it handles small, fixed-length messages, adequate for items of control information. These items

are, however, large enough to serve as effective pointers to larger messages.

We have in the introduction defined any change of state as an "event." However, for the purposes of IPC, we are interested only in a small subset of all events, namely those which are known to be of interest to some process other than the one whose state has changed. The subset is limited by the actual coding of the programs which these processes execute. For the purpose of this paper, we use the term "event" in the context of "subset"-event.

Although every event is necessarily a unique occurrence in time, it is often the case that the significance of an event is indistinguishable from that of other events in a given category. Typically, we would then associate all such related events with a common event type and may think of the occurrence of an event as being the cyclic reoccurrence of its associated event type.

In the Multics IPC, it is the event type, rather than the individual event, which may be put into correspondence with a mailbox. The ipc-setup includes, therefore, an association (by agreement) of a particular event type with a particular mailbox.

By convention, the ipc-setup is made between a single potentially interested process, which we name Recipient, and one or more potentially observing processes which we name Senders.

4. Design Objectives and Decisions

In designing the Multics IPC facility, two types of objectives are defined, a) primary objectives which are of a general nature, and b) secondary objectives which are related to the Multics environment. Decisions are made in order to achieve an implementation that effectively and efficiently meets the design objectives. Remember too, that the facility strived for here is supplementary to, and not a replacement for, the basic IPC mechanism. Programmers are free to ignore an IPC facility entirely, if they choose to establish their own mailboxes and the conventions for their use among the processes that share them.

4.1 Primary objectives

Our primary objective is to design an IPC facility which offers the programmer an opportunity to design and code complex multi-process computations without a disproportionate attendant growth in complexity. At the same time we wish to add to, or at least conserve, the useful properties inherent to the multiplexed computer system, i.e., 1) virtual parallel-processing, and 2) efficient hardware processor resource management. An additional primary objective is modular implementation and simplicity of application. We shall now show, by expanding from the basic IPC mechanism, one way to achieve these primary objectives.

First, we examine the behavior of the basic IPC mechanism in a multiplexed system by applying this mechanism in a generalized example and by trying to assure ourselves that its behavior characteristics do not, possibly, have the effect of "incapacitating" the system.

In our example, suppose that Sender A is a cyclic process which, at a certain point in its loop wishes to send a message S_a to Recipient B by using an agreed upon mailbox M. Suppose recipient B is also a cyclic process:

in every loop transit process B reaches a wait point where it cannot proceed unless it finds a message in mailbox M. By convention, state S_0 of M is interpreted as "mailbox is empty".

In Figures 1-4 we shall make use of a "free style" PL/I, inspired by Dijkstra[1]. We shall apply functions lock and unlock to insure mutual exclusive access to mailbox M[1,8]. These primitives have been implemented elsewhere in Multics (using a hardware test and set instruction[8]), so we freely use them here.

Figure 1 shows coding for an ipc-setup and for the basic IPC mechanism. To follow the coding in this and in subsequent figures note that we have found it convenient to define a new type of variable to type "mailbox" with the following attributes:

1) A mailbox variable is a shared data base. It is associated with a single Recipient.

2) A mailbox M of a recipient process is, more precisely, a structural variable (we shall here nickname it an "M-structure"), consisting of four components: M.ℓ, M.i, M.p, and M.m.

   where: M.ℓ is a "lock-word" operated on by functions lock and unlock.

   M.i is a binary indicator which may assume states "empty" or "not-empty" (the empty state corresponds to the above-mentioned state S_0). The Boolean function test (M.i) is "true" for m.i="empty".

   M.p is the unique identifier of the associated Recipient process.

   M.m is a single interprocess message.

It is also convenient to employ the notion of a variable of type "message" to indicate an interprocess message "attribute." We also note that the function initialize (M) resets mailbox M to its "empty" state. Lastly, observe the use of the coding brackets "PARBEGIN" and "PAREND" within which we define processes A and B to be timing-independent and executing in parallel.

The IPC mechanism shown in Figure 1 is functionally deficient; it can not normally be doing any meaningful work because if A's speed of execution exceeds that of B then A may overwrite (and hence lose) its own messages. If, on the other hand, B is faster than A, it might "read" the same message several times. We are clearly confronted with a synchronization problem.

One improvement that comes to mind is to synchronize the two processes by adding to A's computation a wait loop (similar to B's) to prevent it from sending a message if M.i="empty" and by having process B reset M to the "empty" state after copying the message into its local memory space. This solution however deprives us of the opportunity to have virtual parallel processing, a primary design objective, and is therefore unacceptable. For this reason we choose an alternative solution which is to provide an infinite buffering capability, so that the Sender may always be able to "unload" his message regardless of the Recipient's degree of interaction.

We can achieve this effect by re-defining the mailbox's fourth component, as suggested in Figure 2. Hereafter we shall assume that M.m is in fact a FIFO (First In, First Out) queue of interprocess messages. Thus, we use notations initialize(M), M.m=append(S_i,M) and S_i=remove(M) to convey the ideas of "initialize-", "append to-" and "remove from-" mailbox.

The coding in Figure 2 also removes another weakness of the Figure 1 mechanism, namely, a failure to meet the primary design objective of efficient hardware processor management. Note that in Figure 1 process B's wait loop is wasteful. We would prefer instead a way for process B to willingly abandon its processor. Moreover, we know that B has no need for a processor until A has actually put a message in M. We therefore make use of the Traffic Controller's block/wakeup function, inserting a call to block into B's wait loop and adding to A's message-sending logic the responsibility to wakeup process B, which is potentially blocked, after having appended a message to M. By definition, a mailbox may be associated with a single Recipient only, so use of the notation wakeup (M.p) means "wakeup the process that is known to be associated with M." Our improved mechanism can now be seen in Figure 2.

In order to meet the final design objective, that of modularity and simplicity, it behooves us now to isolate in our IPC mechanism that part which is common to all communicating processes, and achieve a simplifying generalization. This can be done by establishing a set of two IPC primitives which we shall here name _wait_ and _notify_. These new primitives are event-oriented extensions of block and wakeup, respectively and are defined as follows:

notify(M,S) causes message S to be appended to mailbox M, and a wakeup to be signalled to the process associated with M.

_S_ℓ=wait(M)_ assigned to S_ℓ the value of a message S which was removed off the top of M. The wait function has the property that it is associated with a "grain of time" of unpredictable dimension.

Figure 3 shows the wait and notify functions and Figure 4 shows our example communication using the generalized IPC functions.

4.2 Secondary Objectives

Our secondary objectives are to implement functions wait and notify in Multics*, and if necessary adapt them to the Multics environment.

The implementation of wait and notify calls for the availability of a number of difficult-to-provide capabilities, namely a) the ability to attain a shared data base through the use of its unambiguous file system name, 2) lock and unlock primitives, 3) block/wakeup services, and 4) the ability to implement the "mailbox"-type variable.

Fortunately the very "cornerstone" of the Multics design embraces the first three capabilities, so the only significant implementation problem that remains is that of providing an efficient and inexpensive "mailbox" queueing mechanism. The ideal mailbox is an infinite FIFO queue of variable-length messages. As already mentioned, a design decision has been made to restrict the size of an interprocess message to the minimum necessary in order to be able to communicate control information. Two design problems reamins, 1) to set a limit to the mailbox's capacity, and 2) to prevent the saturation of the system's storage media by a multitude of independent and space consuming mailboxes.

The solution adopted ts that of allocating the queue-containing M-structures (one per mailbox) in a single table of generous proportions. By dynamically allocating space in that table as needed, for each element of the queue in an M-structure, it is possible to allow for considerable size fluctuations in the individual queues, within the limits of the table's dimensions.

---

* Terminology used in this paper is not necessarily keyed to that used in internal documents of the Multics project. Thus the wait and notify functions actually have, for historical reasons, other names in "actual" Multics.

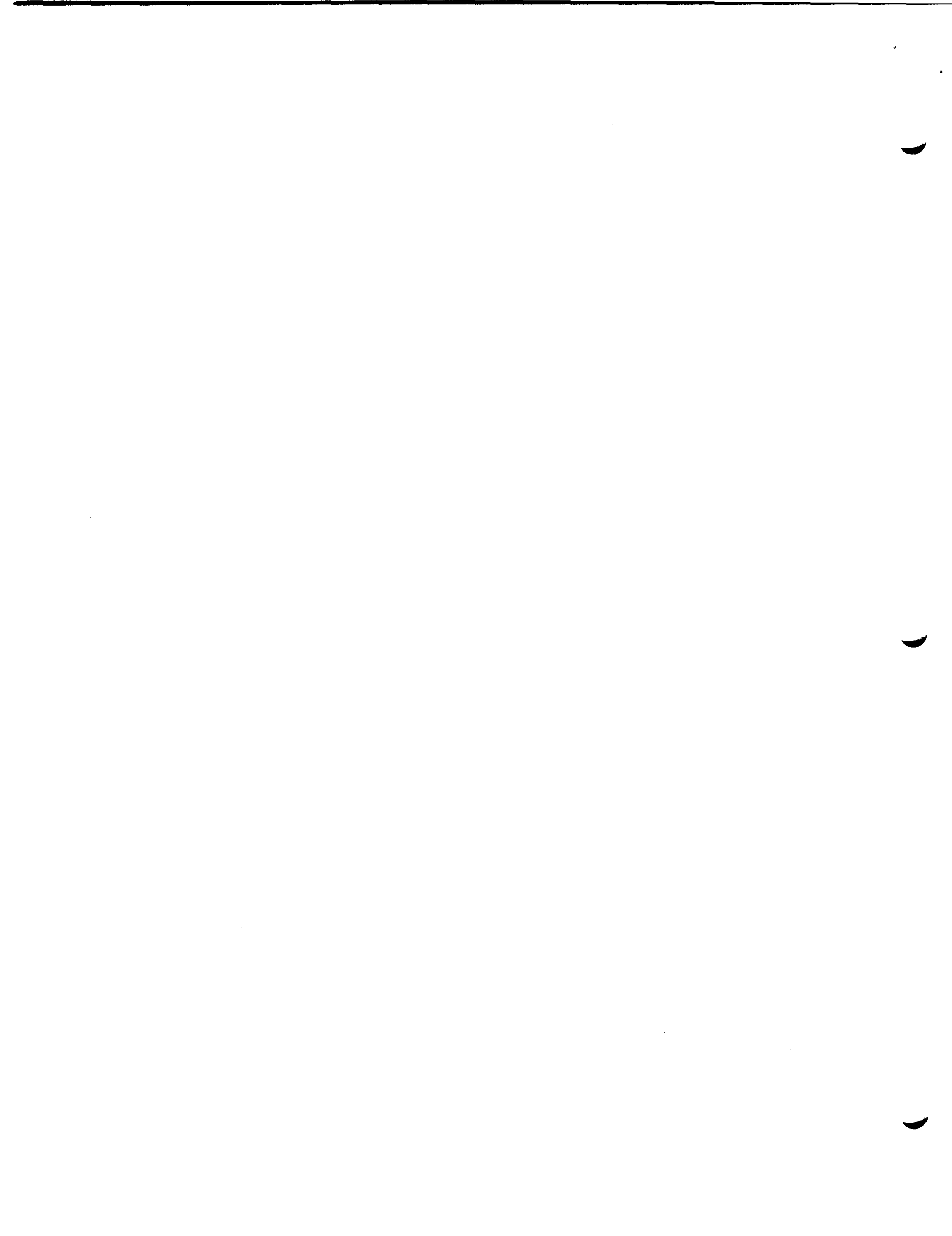
This implementation approach permits us to physically separate the mailbox (which by definition has an agreed-upon "address") from the actual M-structure which is managed ay the system and whose address is a priori unknown. In Multics, we name the M-struction "event channel" and associate it with a system-provided, i.e., system-generated, event channel name which is an (internal and unambiguous) address that is meaningful to the IPC facility (only).

It is the IPC facility's responsibility to correctly manage and manipulate the event channels. The only interface which the programmer needs is provided in the three function references, initialize(M), notify(M.S) and S=wait(M).

The implementation-dependent function initialize(M) which creates an "empty" event channel, associates it with a Recipient process' identification, and with an unambiguous (internal) event channel name, and finally places the event channel name into mailbox M. Any subsequent invokation of wait or notify that uses M as an argument then allows the IPC facility to retrieve the event channel name that is stored in M and with it locate the appropriate event channel (M-structure).

Physical separation of the mailbox from the M-structure was achieved after it was recognized that the mailbox M need only be large enough to contain an event channel name. Thus the mailbox serves only to hold a "pointer" to the place (event channel) that actually holds (or is to hold) the interprocess message. Moreover, it was also recognized that the system-provided event channel name happens to be more compact and convenient to use than the mailbox's file system name. Hence it was in practice found useful to modify the arguments for wait and notify, so that the caller directly specifies the event channel name rather than the file system name of the mailbox that holds the event channel name. It is the event channel name, therefore, that is the "key" to the Multics IPC facility.

The interprocess message in the M-structure's queue has a format such that it may contain an event channel name. This adds an additional sophistication to the IPC facility in that a mailbox requiring a file-system name is needed only for the establishment of the very first ipc-setup. Subsequent ipc-setups may use the initialized event channel(s) to provide further "mailbox services".



5. The Actual Implementation

The implemented wait and notify functions have been slightly expanded to allow for additional sophistication as discussed below.

(a) The notify function returns some control information feedback concerning the Recipient process so that the Sender process may learn, if this is indeed of interest to him, whether or not the Recipient process is still in working order. Thus a Sender may discover, if he examines the return information, that his would-be Recipient no longer exists as an active process.

(b) The IPC appends, as part of each message the identification of the Sender process so that a Recipient process may, if it chooses, inspect the origin of the message and act accordingly. For example, a system process in Multics, called System Control, maintains an event channel in which to receive messages that request termination of any of the processes now being served. Inspecting the origin of the received message helps determine the reason for the termination request, e.g., hangup, logout, system shutdown, out of funds condition, etc., and the appropriate action to be taken in each case.

(c) The wait function has been expanded so that a process may wait for a compound event to happen, e.g., a process may specify that it is waiting for $e_1|e_2|e_3|....|e_n$ (where "|" is the inclusive or and e_i are messages in event channels of type i). This additional function is made possible by expanding the wait function into a module named the wait coordinator which acts as a broker for event messages.

(d) One further (and perhaps the most interesting) extension of the wait function allows a programmer to incorporate event-driven multi-programming

of separate tasks within a single process. This inherent capability which is part of the IPC's wait coordinator has found application in certain key system processes, in particular in the system's System Control process. An appreciation of this feature may well lead to insight for future research and to the identification of new system objectives.

Because a Multics process is a single virtual computer, i.e., consists of a single address space (virtual memory) and a single execution point (virtual processor), it is natural to think of it as having a single purpose. There are two occasions when a single-purpose process would call upon the wait coordinator's services:

1) The process has reached a wait point at an interim stage in its execution path and cannot proceed until an event message has been received, hence wishes to be put into the blocked state.

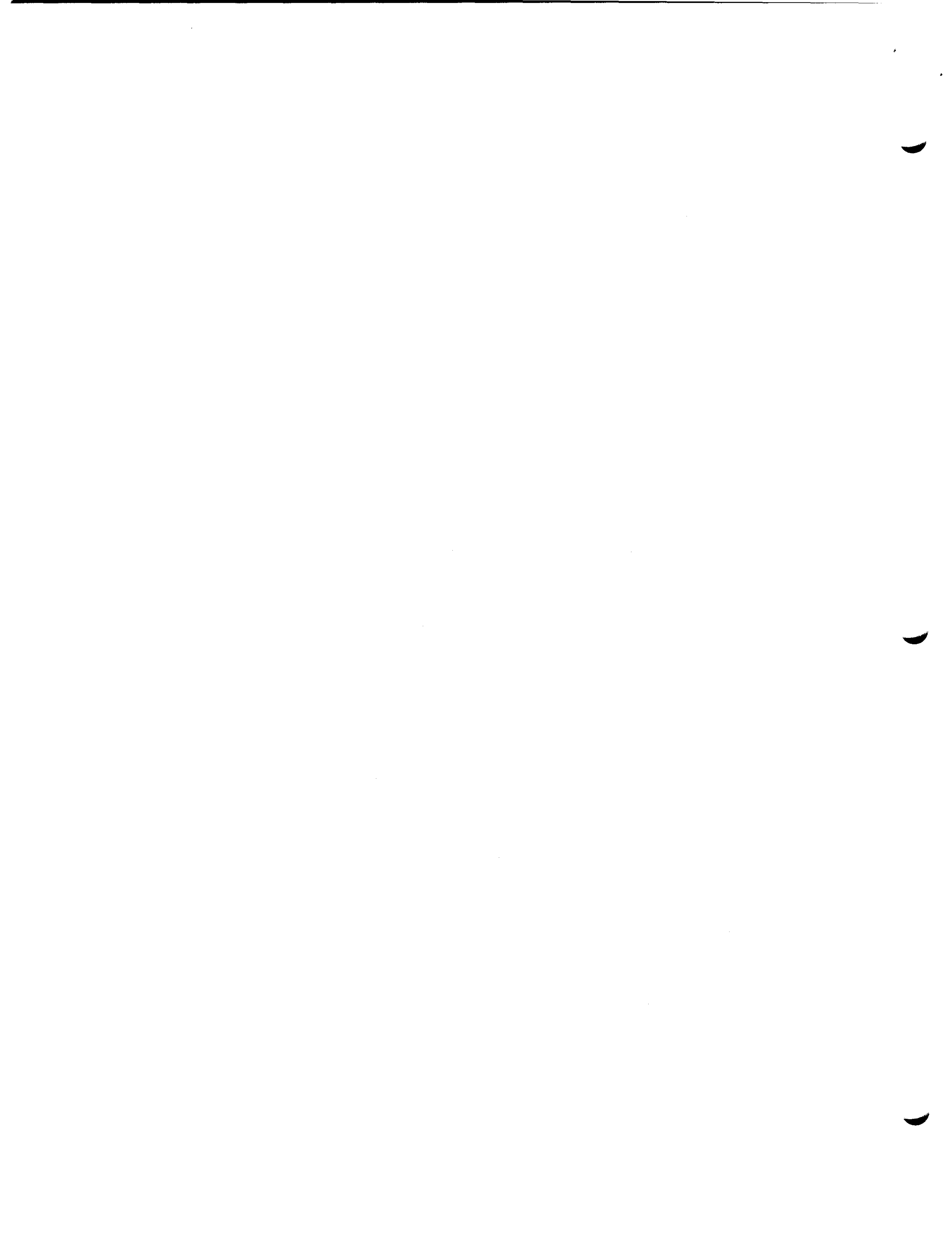
2) The process has reached an end point, i.e., it has executed to the completion of its single purpose and it wishes to be put into the blocked state, either to be terminated by its creator or to await the arrival of an event message indicating that the purpose (of the process) is to be repeated.

Suppose that for reasons of economy (mainly conservation of the work required to maintain address spaces), one wished to coalesce (and condense) into a single process (or rather into a single virtual computer) the individual purposes of a number (n) of separately conceived, though possibly related, processes. Each of the separate purposes may be conceptually regarded as a program to be executed following receipt of a "triggering" event message. We now have a virtual computer that makes the wait coordinator

its homebase; upon receipt of a "triggering" event message the wait coordinator invokes the associated program (purpose) which, upon reaching its end point, returns to the wait coordinator.

It may be useful to name such a process a "Multi-purpose process" and observe that the multi-purpose process blocks itself only if none of its n purposes (or member programs) is capable of running. An additional requirement imposed on our multi-purpose process is that whenever a currently-executing program reaches a wait point it should be able to recursively invoke the wait coordinator and relinquish control in favor of a member program without forcing all other member programs to depend upon the arrival of the first program's peculiar event message.

The Multics implementation of the wait function in fact obeys all of the principles and requirements of the model just described. Hence, we observe that for multi-purpose processes the wait coordinator can serve as an event-driven controller that "multi-programs" the separate tasks (programs) within the process. On the other hand it is conceded that there are limitations in the use of this device. Thus, the response to the receipt of any one event message will tend to deteriorate as the number of separate tasks within a process becomes excessive. Also, with the current implementation of the Multics virtual computer, it is possible for some task within a process to reach an intermediate wait point and thereby indirectly degrade the response of some other task within the same process.



6. Some Applications for Systems Designers

The following discussion may provide the reader some insight as to how an IPC facility like Multics' may be used by system programmers and subsystem designers in the processing of I/O interrupts.

We have, in the introduction, discussed the design decision to convert device signals into IPC messages. In this way there need be no a priori recipient associated with a device signal. A process may "attach" an I/O device by entering its identifier and an event channel name into a table entry which is associated with that specific device (ipc-setup). An incoming interrupt is intercepted by a dedicated interrupt handler procedure which then notifies the intended process over the appropriate event channel.

For certain devices, in particular the user consoles, the interrupt handler actually notifies different processes of different types of interrupts. For example, a console's interrupts are grouped into three types, i.e., "power on/off", "attention", and "end of transmission". The end of transmission interrupts are signalled to the process that is currently "using" the console, however the power on/off and (perhaps) the attention interrupts are signalled to special system processes (typically the above-mentioned system control process) which have the privilege to intervene and perhaps destroy a user process whose console "went dead".

One of the advantages of using an IPC interface at interrupt time is that the system programmer who codes the interrupt handler may concentrate on the intricacies of correctly interpreting the I/O channel's status information without having to bother with the additional problems of processor management and process synchronization. An additional advantage is that

any process may, if given the privilege to do so, simulate an I/O channel. This is very useful in substituting a process for a temporarily disabled I/O device without affecting the corresponding user processes. Also, it is possible to simulate and study some experimental I/O device and use the same interface that would be used in conjunction with the actual device.

7. Recapitulation

Mechanisms for communicating among coexisting processes are required in any multi-programmed computer system. In Multics these essentials are achieved largely as a byproduct of other capabilities that lie at the heart of that system's design, i.e., shared data bases by virtue of unambiguous file system names, lock and unlock primitives, and block/wakeup services for processor multiplexing. A judicious selection of additional system wide conventions has led to the installation of an IPC facility that builds on these central capabilities. The resulting extension of the Multics supervisor provides general and functionally modular wait and notify services. These are simple to implement and maintain because their application is independent of the specific nature or purpose of the communication or of the Sending and Receiving processes.

The IPC facility is used extensively by the system itself, for instance in handling I/O interrupts after converting these hardware signals to software calls to the notify function. As additional applications for the IPC facility were recognized the facility has evolved in its sophistication (and may infact evolve further). Thus the wait function in particular has evolved into a (recursively called) wait coordinator which polls for various event messages coming to a process and, if desired, acts as an event driven controller that can multi-program separate tasks within a process. All these services simplify the logical structure of the applications and systems programs that communicate with the shared, central supervisor.

Control over the complexity in various computer applications is one of the important promises our profession has made to those it serves. Wait/notify services like those described in this paper appear to offer promise as tools for limiting the growth in complexity of useful multi-process computations and subsystems which operating systems are designed to support.

```
BEGIN
ipc-setup-a: declare M mailbox;
ipc-setup-b: initialize(M);
PARBEGIN
A: BEGIN
   declare Sa message;
   local initialization of A;
   A1:first part of A's computation;
      lock(M.ℓ);
      basic-mechanism-a: M.m=Sa;
      unlock(M.ℓ);
      second part of A's computation;
      goto A1;
   END;
B: BEGIN
   declare Sb message;
   local initialization of B;
   B1:first part of B's computation;
   B2:lock(M.ℓ);
      basic-mechanism-b:
      if test(M.i) then
      BEGIN
         unlock(M.ℓ);
         goto B2;
      END;
      Sb=M.m;
      unlock(M.ℓ);
      second part of B's computation;
      goto B1;
   END;
PAREND;
END;
```

Figure 1: Application of the basic IPC mechanism to a generalized interprocess communication problem.

```
BEGIN
ipc-setup-a: declare M mailbox;
ipc-setup-b: initialize(M);
PARBEGIN
A: BEGIN
   declare Sa message;
   local initialization of A;
   A1:first part of A's computation;
      lock(M.ℓ);
      basic-mechanism-a: M.m=append(Sa,M);
      wakeup(M.p);
      unlock(M.ℓ);
      second part of A's computation;
      goto A1;
   END;
B: BEGIN
   declare Sb message;
   local initialization of B;
   B1:first part of B's computation;
   B2:lock(M.ℓ);
      basic-mechanism-b:
      if test(M.i) then
      BEGIN
         unlock(M.ℓ);
         block;
         goto B2;
      END;
      Sb=remove(M);
      unlock(M.ℓ);
      second part of B's computation;
      goto B1;
   END;
PAREND;
END;
```

Figure 2: Basic mechanism improved by introduction of block/wakeup functions and by addition of queuing capabilities to the "mailbox" variable.

```
BEGIN    /* the notify function */
notify: procedure(M,S);
declare M mailbox, S message;
   lock(M.ℓ);
   M.m=append(S,M);
   wakeup(M.p);
   unlock(M.ℓ);
END;

BEGIN    /* the wait function */
wait: procedure(M) returns(S);
declare M mailbox, S message;
declare S_t message; /* temporary */
W1:lock(M.ℓ);
   if test(M.i) then
   BEGIN
      unlock(M.ℓ);
      block;
      goto W1;
   END;
   S_t=remove(M);
   unlock(M.ℓ);
   return(S_t);
END;
```

Figure 3: The generalized notify and wait functions.

```
BEGIN
ipc-setup-a: declare M mailbox;
ipc-setup-b: initialize(M);
PARBEGIN

A: BEGIN
   declare Sa message;
   local initialization of A;
   A1:first part of A's computation;
      basic-mechanism-a:notify(M,Sa);
      second part of A's computation;
      goto A1;
   END;

B: BEGIN
   declare Sb message;
   local initialization of B;
   B1:first part of B's computation;
      basic-mechanism-b:
      Sb=wait(M);
      second part of B's computation;
      goto B1;
   END;

PAREND;
END;
```

Figure 4: Application of the generalized IPC mechanism.

References

1. Dijkstra E.W., "Synchronizing Primitives," an appendix to "The Structure of the 'THE'-Multiprogramming System," first ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1967. Published in the Communications of the ACM, May 1968, pp. 345-346.

2. Corbató, F.J. and Vyssotsky, V.A., "Introduction and Overview of the Multics System," AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., pp. 185-196.

3. Saltzer, J.H., "Traffic Control in a Multiplexed Computer System," Sc.D. Thesis, M.I.T. Department of Electrical Engineering, May 1966. Available as M.I.T. Project MAC Technical Report TR-30.

4. Rappaport, R.L., "Implementing Multi-Process Primitives in a Multiplexed Computer System," S.M. Thesis, M.I.T. Department of Electrical Engineering, August 1968. Available as M.I.T. Project MAC Technical Report TR-55.

5. Dennis, J.B., and Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations," ACM Programming Languages and Pragmatics Conf., San Dimas, California, August 1965. Published in the Communications of the ACM, March 1966.

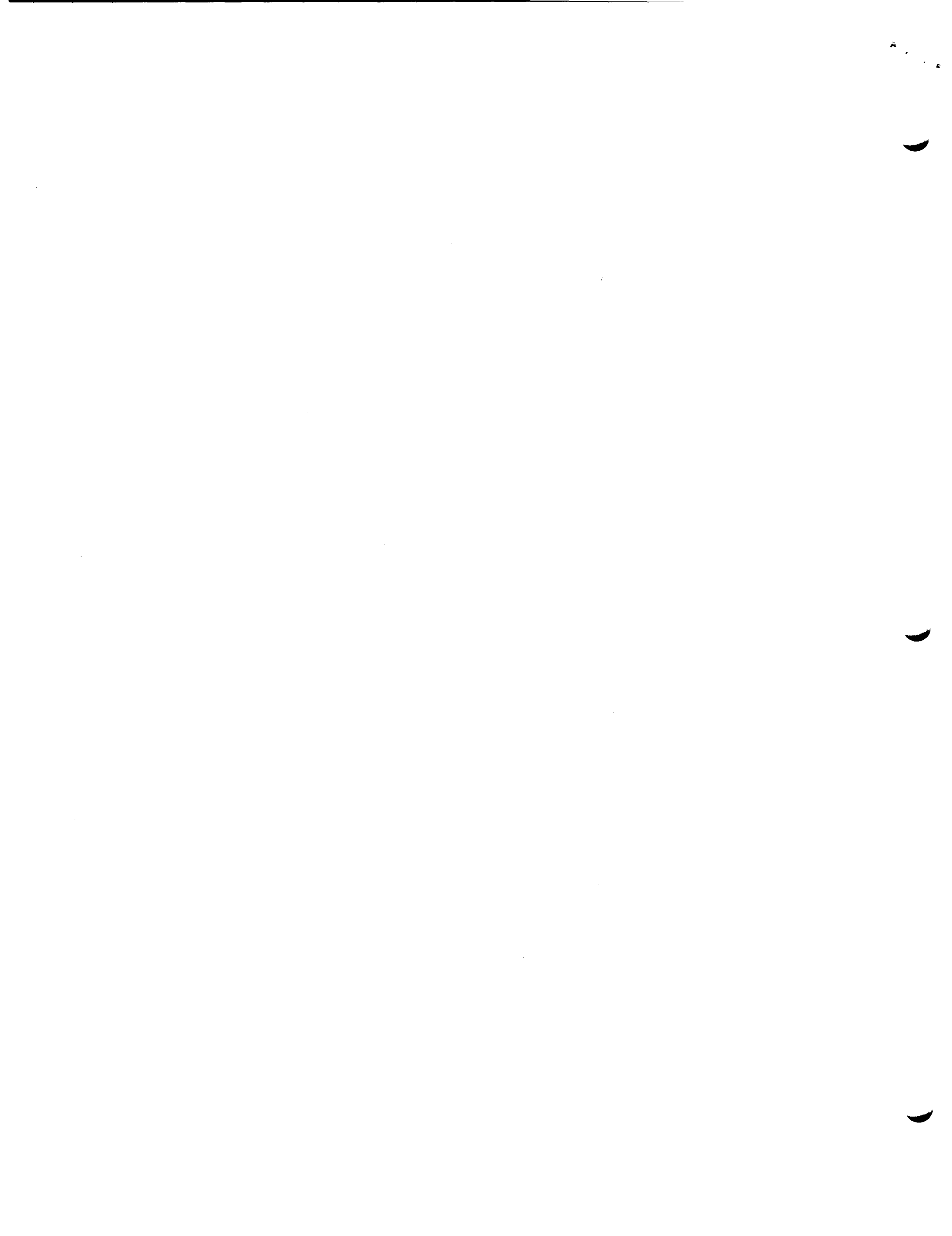
6. Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory," second ACM Symposium on Operating System Principles, Princeton, New Jersey, October 1969.

7. Graham, R.M., "Protection in an Information Processing Utility," first ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1967. Published in the Communications of the ACM, May 1968, pp. 365-369.

8. Lampson, B.W., "A Scheduling Philosophy for Multiprocessing Systems," first ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1967. Published in the Communications of the ACM, May 1968, pp. 347-360.

9. Witt, B.I., "The Functional Structure of OS/360: Part II, Job and Task Management," IBM System Journal, 5, Part 1, (1966), pp. 12-29.

10. Saltzer, J.H., "CTSS Technical Notes", Massachusetts Institute of Technology, Project MAC Technical Report TR-16, March 1965.



M0114
Addendum

The following acknowledgement was inadvertantly omitted and should be added to "The Multics Interprocess Communication Facility" by Michael J. Spier and Elliot I. Organick.

ACKNOWLEDGEMENT

The research reported in this paper is part of the Multics development effort of Project MAC at M.I.T., Bell Telephone Laboratories and the General Electric Company, Inc. Following is the (alphabetically ordered) list of (some of) the persons who have contributed to the work described in this paper:

A. Bensoussan, F. J. Corbató, R. C. Daley, L. Lambert, K. J. Martin, J. F. Ossanna, R. L. Rappaport, J. H. Saltzer, P. Schicker, M. D. Schroeder, B. A. Tague and C. M. Vogt.