Inter-Office Memorandum

To	Bill Lynch	Date	July 15, 1977
From	Hugh C. Lauer	Location	Palo Alto
Subject	A Simple Message-Passing Facility for Pilot	Organization	SDD/SD

XEROX

Filed on: <Lauer>SendMessage.memo

At your request, I have prepared this memo outlining a streamlined version of a process and message mechanism based on the philosophy of Chapter 2 of the current Pilot Functional Specifications. The scheme described here should only be considered if the recommendations of the Process Working Group are rejected, and then only after it has been subjected to a careful technical scrutiny itself. The ground rule which I have followed is that the scheme is to be implementable within the existing specifications of the Mesa Language and D0 Principles of Operations -- i.e., no changes to the compiler or microcode are to be contemplated. For purposes of this presentation, I have deliberatedly changed all terminology, both to be more consistent with generally accepted terminology of computer science and to avoid confusing concepts described here with similar but not identical ones in previous documents about Pilot.

The mechanism and interfaces described here represent a very basic set of process communication facilities. It is hard to tell at this time whether or not a careful implementation of these will impose an excessive amount of overhead on Pilot or its applications (although that overhead will clearly be orders of magnitude greater than if similar facilities were implemented in Mesa and/or microcode). If it is felt that these facilities are not sufficient and that additional mechanisms must be added to cater for various special cases, then the whole scheme is likely to get out of hand very quickly, requiring both excessive space and excessive time to execute its basic functions. Thus, I have resisted the temptation to add extra functionality and virtual flexibility and have instead concentrated on identifying the style of system design for which this mechanism is most appropriate.

The Basic Mechanism

This scheme is a message-passing facility. All communication and synchronization among processes in the system is done by passing messages. The mechanism assumes the existance of a *Ready Queue* of processes (or its equivalent) ordered by priority. The processor is always allocated to the process at the top of this queue. The message-handling operations manipulate this queue as a side effect, thus providing the scheduling function at the microscopic level. I do not expect that scheduling at the macroscopic level will be required for any of our applications; but if it is, it would be provided by a process which could manipulate the Ready Queue explicitly. The implementation is expected to provide buffers in which messages are constructed, stored, and queued; this decouples the management of messages from the management of memory and the swapping of the code and data of processes.

Data Types

The format of a message is a fixed number of words (say, four) of unspecified type. All message will be constructed from and interpreted as other Mesa data types by means of loopholes.

```
Message: TYPE = RECORD[UNSPECIFIED, UNSPECIFIED, . . .];
```

The mechanism provides a means of sending a message to a destination and optionally, waiting for a reply to that particular message. In order to be able to identify messages for the purpose of sorting out their replies, the following type is introduced. Objects of this type are used by processes which originate messages.

```
MessageId: TYPE = PRIVATE . . . ;
```

By symmetry, the mechanism provides a means for a process to receive arbitrary messages. Such a process will operate upon the contents of a message and may choose to reply. In order to permit that process to reply to the originator of the message without imposing on it the burden of keeping track of who or where that originator is, the following type is introduced.

```
MessageHandle: TYPE = PRIVATE RECORD[sender: ProcessId, msg: MessageId,
        . . .];      -- or something equivalent
```

The type **ProcessId** is strictly internal to the Pilot message-handling facility and will not be available to any process. Instead, processes communicate over *channels*, which identify not only the destination process but a *port* which is associated with the 'kind' of message or service being requested. Thus, the following types are defined.

```
ProcessId: PRIVATE TYPE = . . . ;

PortNumber: TYPE = [0 .. MaxPortNumber];
                   -- 15 seems reasonable for MaxPortNumber

Channel: TYPE = PRIVATE RECORD[server: ProcessId, port: PortNumber];
```

Message-Handling Operations

There are four basic message-handling operations provided by this scheme: **SendMessage**, **WaitForMessage**, **SendReply**, and **AwaitReply**. The system will maintain for each process a queue of messages sent to it but not yet read or received by it. These queues are maintained in FIFO order. The operations cause messages to be added to or deleted from the the appropriate queues; they have the added side effect of relinquishing the processor if necessary. In order to prevent a run-away process from clogging the system resource used for implementing message buffers, each process is assigned a limit of the number of outstanding messages which can be attributed to it. A message is attributed to a process if either it was originated by that process via **SendMessage** or it is a reply to a message previously attributed to it. A process may send at most one reply to any message it receives.

```
SendMessage: PROCEDURE[msg: Message, destination: Channel]
        RETURNS[MessageId];
```

This procedure causes the message represented by the first parameter to be queued on the destination process and port identified by the second parameter. If the destination process is waiting for such a message, it is inserted in the Ready Queue; if it is of higher priority the originating process, the processor is relinquished. The value returned from this operation is used only as a parameter to **AwaitReply**.

> **WaitForMessage: PROCEDURE[SET OF PortNumber] RETURNS[h: MessageHandle, p: PortNumber, m: Message];**

This operation is the means by which a process waits for an arbitrary message on any of the ports specified by the parameter. If no messages are queued, the processor is relinquished to the next process on the Ready Queue. If such a message is queued, the first one is returned along with its PortNumber and a handle through which a reply can be sent.

> **SendReply: PROCEDURE[h: MessageHandle, reply: Message];**

This procedure is the means of replying to a particular message. The destination of the reply is the process which sent the original message identified by the first parameter. If that process is waiting for this particular reply, it is inserted in the Ready Queue; if it is of higher priority, the processor is relinquished to it. Otherwise the reply is queued.

> **AwaitReply: PROCEDURE[MessageId] RETURNS[reply: Message];**

This operation is the means by which a process waits for a reply to a particular message. If the reply has already been queued, then it is returned. Otherwise the processor is relinquished to the next process of the Ready Queue.

These operations can generate several possible signals or errors in response to bad parameters; these will not be listed in this memo. However, two other signals are particularly relevant:

> **TooManyMessages: SIGNAL;**

> **ReplyNotPermitted: ERROR;**

The first of these is generated by **SendMessage** if the process has too many messages already attributed to it. The second is generated by **SendReply** if a reply has already been issued to the particular message. A little more control can be imposed and the implementation can be simplified if **Channel** is geven another attribute, namely a Boolean indicating whether or not a reply is required to any message transmitted over that channel. In this case, a message received by a process (via **WaitForMessage**) but not yet answered would be attributed to that process. The signal **TooManyMessages** could then also be generated if a process tried to wait for more messages without first replying to some in hand. Similarly, **ReplyNotPermitted** would be generated by **AwaitReply** or **SendReply** if either were applied to a **MessageId** for which no reply was required.

Creating Processes

In order to create a new process, several things must be provided. First, a configuration file which contains the Mesa Object code is required. It is proposed that this be the BCD file output by the new binder. Each process will require its own file, and binding of processes together in advance is not feasible. Second, an MDS must be provided for the process to execute in. We will assume that this is a *Space* in the sense of my memo on a streamlined memory management system (<Lauer>memory.memo), however the exact nature of this is not important to this discussion. Third, the new process must be provided with a means of communication in order to get started. Since we are working within the existing constraints of Mesa, no binding facilities are available to associate channels with ports at compile or binding time (see <Lauer>MessageSystem.memo) and thus this must be done at run-time in long-hand by each process using this communication means. Finally, we also need to specify a priority and a limit on the number of messages which can be attributed to this process; both of these numbers will remain static throughout the life of the process. The following procedure creates a new process.

CreateProcess: PROCEDURE[configuration: FileCapability, MDS: SpaceHandle, interface: Channel, priority: CARDINAL, messageLimit: CARDINAL];

There are no constraints on the relation between the priority of the creating process and the created one. The created process is immediately entered into the Ready Queue, and if it is of higher priority, the processor is relinquished to it. If the MDS parameter is defaulted, the new process is added to the mds of the creating process. The **interface** parameter is a channel which is *intended* to be used for exporting to the outside world the channels implemented by this process and importing from it the channels needed by this process.

channel: PROCEDURE[PortNumber] RETURNS[Channel];

This procedure is the means by which a process generates the channels which it exports.

Stylistic Considerations

The message-passing scheme of this memo is best suited to a restricted kind of system, namely one in which processes are few in number and relatively static. Thus **CreateProcess** can afford to be fairly slow and cumbersome and do both the loading of Mesa object code as well as setting up the process in the Ready Queue; it should be called primarily (or only) at system configuration time. There is no operation to delete a process because this rarely or never needs to be done. (These two operations in the message-passing system are analogous to the Mesa NEW and DeleteGlobalFrame operations; neither is expected to be called very often in the production environment, if at all. See my memos on <Lauer>Duality.ears and <Lauer>MessageSystem.ears.) The typical system organization for our model has a process associated (statically) with each system resource, a few others for special 'background' functions and a small fixed number of processes to implement the applications. The total number of processes in the system is of the order of a dozen or two. Priorities are assigned to processes at the time of creation to reflect the timing constraints of the functions they support, and these remain fixed.

Messages, by contrast, are lightweight objects; they can be created often, rapidly, and with ease. They are used in lieu of the lightweight processes of procedure-oriented systems to encapsulate asynchronous activity, to represent trainsient situations, and even to encode the state of application transactions. Processes stay in static environments, while messages fly around the system from process to process, requesting service of the various resources.

In general, a process is set up so that all messages on a particular port mean the same thing or represent the same kind of request. For example, a file server process may have one port for creating files, another for reading them. If we were in the strongly-typed environment of Mesa, we would use procedures for this purpose and have full type-checking on parameter lists. Unfortunately, our ground rule prevents us from modifying Mesa to take advantage of type-checking on messages. Thus, the typical use of the message-handling facilities should be encapsulated in modules which present a procedural interface and use loopholes to transform the structured, typed, parameter lists into untyped messages and replies. It is unwise to design systems or applications programs with direct calls upon **SendMessage**, for example, scattered about the code.

This style of system design does not readily support data shared among processes. It is much easier to organize things so that a 'shared' data resource is 'owned' by a particular process and is updated and accessed only as a result of messages to that process. For data objects which do need to be accessed by more than one process (such as application data structures in certain cases), it is better to arrange it so that only one process at a time has knowledge of each object. This requires a very strict discipline controlling the naming and accessing of such objects. For example, one successful method is to store pointers to these things only in messages, with the constraint that a pointer to a given object may not appear on more than one message at a time. All references to an object are then indirect through messages. This automatically guarantees that only one process at a time knows about an object and that it 'forgets' about it as soon as the message is answered or deleted.

The facilities to wait for messages and replies thus provide all the synchronization required for designing the operating system, common software, and applications. Nothing resembling monitors, semaphores, or locks is needed or contemplated. In particular, the hardware instructions for blocking and waking up processes and for manipulating the wakeup count should *not* be used anywhere except within the message-handling procedures themselves (and also the procedures which queue IOCB's onto Controller Status Blocks -- these are effectively operations which 'send messages' to I/O devices to control their activity).

Since processes in this model are fairly static, so should the interfaces between them be static. Each process defines its own ports -- i.e., the classifications of messages it is willing to accept -- then 'exports' these in the form **Channels** to other processes. Similarly, it must 'import' **Channels** from other process in order to access the resources and services it needs. These exports and inports should be established at the time the process is created during system configuration. The mechanism is not really rich enough to support dynamic channels usefully without introducing a lot of complexity in the structure of the system or applications and a lot of overhead a run-time. (In particular, the common practice in the communications area of setting up sockets dynamically for each activity on the network is really too expensive. The kind of connections we need to establish resemble more closely the connections made by the BIND operation in Mesa -- i.e., something we do not contemplate doing very often in Peoria.)

We can summarize the style of system design with our mechanism supports best by providing a paradigm for its processes. This can be expressen be the following outline of Mesa code.

```
process: PROGRAM[interface: Channel] =
      BEGIN  port1: PortNumber = . . . ;
             port2: PortNumber = . . . ;
             : . .
             exports: RECORD[channel1, channel2, . . . : Channel] ←
                   [channel[port1], channel[port2], . . . ];
                         -- use symbolic names for ports and channels
             imports: RECORD[extChannel1, extChannel2, . . . : Channel];
             mask: SET OF PortNumber ← . . . ;
             h: MessageHandle; p: PortNumber; m: Message;
                   . . .
                         -- other declarations for the process
             imports ← AwaitReply[SendMessage[exports], interface];
                         -- Export our channels and Import the external ones
                         -- Loopholes are not shown
                         -- Include other process initialization here
             DO
                   [h, p, m] ← WaitForMessage[mask];
                   SELECT p FROM
                         port1 => . . . ;
                         port2 => . . . ;
                               . . .
                   ENDCASE;
             ENDLOOP;
      END.
```

This program takes one parameter, namely the interface channel passed to it by **CreateProcess**. It declares the ports which it will implement and a variable to contain the channels implemented by other processes which it will need. It then sends a message over the interface channel to export channels which access its ports and it waits for a reply containing the external channels it needs to import. (Obviously, if the size of these records is too large for a message, then it must arrange to send a pointer to a record containing the logical contents of the message. We have not shown any of these details here.) After any other initialization, the process enters an infinite loop which waits for messages on any of the ports named by the variable *mask* and performs whatever action is required on the message received. The variable *mask* provides a way of controlling which kind of messages the process is willing to receive and corresponds roughly to the condition variables of monitors.

The effect of this model of process is to implement a resource server which acts on each request serially. For may purposes, this is quite sufficient. But for some situations, the serialization imposed by this discipline is intolerable. In these cases, several more elaborate techniques are normally used. One trick is to set up several processes with some private channels for communication among them. One process servers as the interface to the outside world, receives requests, and multiplexes these in messages to its cohorts. Another technique is for the process itself to simulate internally a number of subprocesses, each of which keeps track of one request. Neither of these techniques is as well-developed as that of object-style programming in Mesa and other procedure-oriented environments, but they are usually satisfactory for most purposes.

Implementation

The implementation of the mechanism described in this memo is straightforward and needs little comment. It should be pointed out that the Process State Blocks will be fairly large and cumbersome and will need to reside in non-swapped memory. However, there are not expected to be too many of them, so this is not much of a problem. Message buffers will be very small but all of them will also have to reside in non-swapped memory. The total number of message buffers will have to be fairly carefully calculated to avoid a system disaster resulting from exhaustng the supply. We might take a hint from the GEC4080 and implement the free message pool as a FIFO queue in order that the freed message buffers could serve as a useful diagnostic tool during development. Finally, we would implement the message passing operations so that the distinctions between hard and soft processes is invisible to everyone save the microcode; this, of course, makes the implementation a little more complicated but it is worth it.