# MESSAGE CLASSES:

# AN APPROACH TO PROCESS SYNCHRONIZATION*

Gregory R. Andrews

TR 76-275

April 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

---

* This research was supported in part by the National Science Foundation under Grant GJ-42512.

# MESSAGE CLASSES:
# AN APPROACH TO PROCESS SYNCHRONIZATION*.

Gregory R. Andrews

Abstract

In multiprogramming systems, parallel processes compete for access to shared resources and cooperate by exchanging information. Semaphores are a useful means for controlling competition and synchronizing execution and inter-process messages are useful for communication. Neither semaphores nor inter-process messages, however, are natural for solving both problems. This paper introduces a new approach, message classes, which combines and extends features of both semaphores and message passing. Using message classes, numerous mutal exclusion, producer/consumer, process communication, and resource allocation problems can be readily solved.

* This research was supported in part by the National Science Foundation under Grant GJ-42512.

## 1. INTRODUCTION

The two fundamental process coordination problems in multiprogramming systems are mutual exclusion and communication. When two or more processes compete for a shared resource which can only be used by one at a time, each must exclude the others when accessing it. This requires that each process reserve the resource, use it, and then release it. Communication on the other hand is the means by which two or more processes cooperate in performing a task; one process produces information which another consumes.

For many years, semaphores [2], or variants thereof, have been the most commonly used means for coordinating processes. A semaphore is a special type of integer operated on by only two operations: P (Wait) and V (Signal). Although semaphores are a simple and natural means for mutually excluding processes, they cannot be used directly for communication. Using semaphores, one can implement inter-process message buffers, but the message facility is then distributed among processes and each must call buffer manipulation procedures.

The communication problem can be directly solved using inter-process messages such as those described by Brinch Hansen [1]. Inter-process messages are sent to a named process and added to a queue, or mailbox, associated with the process. While one process can receive from many senders, he can only send to one receiver at a time. Using message passing alone, the mutual exclusion problem cannot be directly solved; it requires having one process which mediates requests for access or performs the access itself.

In this paper, we describe a single coordination mechanism, message classes, which combines properties of semaphores and message passing and can directly solve both exclusion and communication problems. A message class is a group of "resources" of the same type; for example, a set of empty buffers, or IO messages, or channels. It is operated on by two operations, Send and Receive which respectively release (communicate) and acquire members of th class[1]. Like a semaphore, a message class is a shared variable; it is not tied to a process. It is different from a semaphore, however, because an item in a message class can contain information such as a buffer address, a channel address, or an IO command. In addition to combining aspects of semaphores and inter-process messages, a message class can also be used to solve problems where a combination of exclusion and communication are present. In a spooling system, for example, there might well be a fixed number of buffers used by reader and writer processes. Each buffer is accessed by only one process at a time but after being filled (or emptied) it is passed on to another process; it becomes a message stored in a pool.

In the remainder of this paper, we define message classes

---

[1] Message classes are a major simplification, with some changes, of a more general process synchronization facility described by Shaw [4,5].

and give examples to illustrate their application. Their implementation is also considered. They have been successfully used in numerous student operating systems at Cornell, and have been the basis for the design of a real-time executive for military applications.

## 2. DEFINITION OF MESSAGE CLASSES

A message class is a set of "messages", each containing the same type of information, accessed by Send and Receive operations. For the moment, assume that we can declare a message class as follows:

<name>: message class(type,length);

The name refers to a descriptor describing the contents and status of the class; type and length will be described shortly. The operation Send(class name, message) adds a message to the class and awakens a waiting process, if possible; the operation Receive(class name, message) returns a message, delaying the receiver, if necessary, until one becomes available. Both are considered indivisible (un-interruptible) operations with respect to the class.

A message class descriptor has the following fields, as shown in Figure 1:

(1) The class type, either directed or undirected.

(2) An available list of messages sent but not yet received.

Figure 1

Message Class Descriptor

| Field | Meaning |
| --- | --- |
| Type | "directed" or "undirected" |
| Avail_list | Header of a linked list of available messages. |
| Length | Length of each message on the available list. |
| Wait_list | Header of a list of processes waiting for messages. |

(3) A length field indicating how long each message is.

(4) A waiting list of processes who have executed Receives which are not yet satisfied.

A directed type of class is used for process to process communication; each message is sent to a specific process identified by the first word in the message. A directed class acts like a large switch, connecting senders and receivers; it may be used by a single sender and receiver or by many. An undirected class, on the other hand, contains a pool of messages which can be produced (sent) or consumed (received) by <u>any</u> user of the class. It is used for exclusion, for resource allocation, or for communication when a process does not care who receives his message. The purpose of the length parameter is to indicate how much information is to flow from senders to receivers. As will be shown, this may vary from zero (to simulate semaphores) to many words, depending on the function of the class.

The operation Send(class, message) adds a message to the named class; the message parameter is the address of (pointer to) the message contents. Directed classes require that the first component of the message name another process. If that process is waiting, then a <u>copy</u> of the message is given to it and it is awakened (by some unspecified means). If the intended receiver is not waiting, then a copy of the message is stored on the available list. In either case, the process executing Send can continue execution. The only difference for undirected classes is that a Send awakens <u>any</u> waiting process and gives it a copy of the message. These actions are outlined in a pseudo-Algol style in Figure 2.

Figure 2

Send Operation

```
Send: procedure (class, message);
    /* Add a message to the named class.
       Give it to a waiting receiver, if any. */
    if Type(class) = "directed" then
       begin /* first word of message identifies
                the intended receiver, IR */
             IR := first word of message;
             if IR is on Wait_list(class)
                then begin
                     Remove(IR) from Wait_list(class);
                     Copy message contents into
                            IR's storage; change first
                            word of message to name of
                            sender
                     Wake_up(IR)  end
                else store copy of message on
                     Avail_list(class), storing
                     Length(class) words and
                     name of sender
       end
    else begin /* type is undirected -
            any process can receive message */
             if Wait_list(class) not empty
                then begin
                    Receiver := first process on
                              Wait_list(class)
                    Copy message contents into
                         Receiver's storage
                    Wake_up(Receiver)   end
                else begin
                      Store copy of message on
                      Avail_list(class), storing
                      Length(class) words  end
         end
    end Send;
```

The Receive(class, message) operation takes a message from the Avail_list of the named class; the message parameter is the address of (pointer to) the locations where a copy of the message is to be stored. It is outlined in Figure 3. A Receive of a directed class selects the first available message sent to the receiving process, if one is available, and returns a copy of the message, changing its first component (which named the receiver) to the name of the sender. If no message is available, the calling process is blocked until one is sent; some scheduling mechanism then selects another process for execution unless a busy wait is employed. A Receive of an undirected class selects the first available message without concern for who sent it. In this case, the message is copied in its entirety into the receiver's address space; the caller is blocked only if no messages are available. A zero length message, of course, results in no copying. It merely serves as a signal.

As stated, the difference between directed and undirected classes is that the former pass messages from senders to specified receivers whereas the latter just queue them up without regard for the receiver. Therefore, in an undirected class, at least one of the Avail_list or Wait_list is empty. In a directed class there might be entries on both lists but no process is kept waiting if a message for him is available. In either case, the Send or Receive operations have very few statements to execute. We now look at some examples to clarify these concepts.

Figure 3

Receive Operation

```
Receive: procedure (class,message);
        /* Take a message from the named class, if there
           is one, and store it in the locations referred
           to by message.  Otherwise block the caller. */
        if Type(class) = "directed" then
           begin /* Find a message intended for calling process.
                  Let CP denote current(calling) process */
               if a message for CP is on Avail_list(class)
                  then begin
                         Remove first such from Avail_list(class)
                         Copy contents into locations
                              pointed to by message parameter
                              (store Length(class) words)
                         Set first word of message to name of sender
                         end
                  else begin /* no message available */
                         Store CP name and message
                              parameter on Wait_list(class)
                         Block(CP)  end
           end
        else begin /* Type is undirected - any message
              can be received.  Let CP denote current process */
               if Avail_list(class) not empty
                  then begin
                            Remove first message from Avail_list(class)
                            Copy contents into locations pointed
                                 to by message parameter (store
                                 Length(class) words)
                            end
                  else begin /* no message available */
                            Store CP name and message parameter
                                 on Wait_list(class);
                            Block(CP);  end
             end
        end Receive;
```

## 3. APPLICATIONS

In this section we illustrate the use of message classes to simulate semaphores, handle IO and interrupt communication, and manage a pool of buffers in a spooling system.

### 3.1 Mutual Exclusion and Semaphores

The critical section problem is to insure that at most one process at a time can access a shared variable. The usual means for solving the problem is to use semaphores. Let V be a shared variable (or set of variables). Then each process accessing V does so via statements contained in a critical section, implemented as follows. Let $S_V$ be a semaphore, with initial value 1, which is used to synchronize access to V. Then the critical section becomes:

```
Wait(S_V)
  <statements accessing V>
Signal(S_V)
```

To use a message class instead of a semaphore, we need to declare a class, initialize it and then use it with Send and Receive. Since semaphores are not attached to processes, and Wait and Signal do not name processes, an undirected class is appropriate. Also, since semaphores do not contain information directly accessible to processes, all the Avail_list need indicate is present or absent. Therefore our class to simulate $S_V$ is declared as:

$C_V$: message class ("undirected",0);

Since $S_V$ was initialized to 1, $C_V$ must initially contain one "message", generated by Send($C_V$,0); a zero second argument is used since there is no data in the message. Finally, a critical section becomes:

```
Receive(C_V,0);
 <statements accessing V>
Send(C_V,0);
```

Receive serves the same purpose as Wait, namely it sees if anything is available and takes it if possible. Similarly, Send and Signal serve the same purpose: they both increment a value and awaken a waiting process if there is one.

This relation between semaphores and message classes is more generally true. Whenever a semaphore is initialized to some value C, a message class is initialized by C Sends; this is a little awkward but need only be done once per class. Whenever a semaphore is signalled, a message (of length 0) is sent. Whenever a semaphore is awaited, a message (of length 0) is received. Message classes can therefore do anything that a semaphore can at the slight cost of extra initialization code and one extra parameter per Send and Receive. It should be noted that any implementation of semaphores which uses a waiting list (as opposed to a busy wait) executes the same basic actions that are taken by Send and Receive (see Figures 2 and 3).

### 3.2 IO and Interrupt Processing

An efficient means for performing IO on peripheral devices is to have a distinct driver process for each device. IO is then initiated by sending a message to a device driver and the result is returned by a completion message. Coordination between a device driver and the hardware IO interrupt mechanism can also be handled by message passing. After starting IO, a driver waits for an interrupt message. When the interrupt occurs, the interrupt handler sends a message to the appropriate driver process. This IO processing technique is depicted in Figure 4. To effect the coordination, three directed message classes are needed, DOIO, COMP, and INT. We assume that a DOIO message has the format:

1. driver process name
2. operation
3. address on device (e.g. sector and track)
4. buffer address
5. byte count

A completion message has the format:

1. user process name
2. completion status

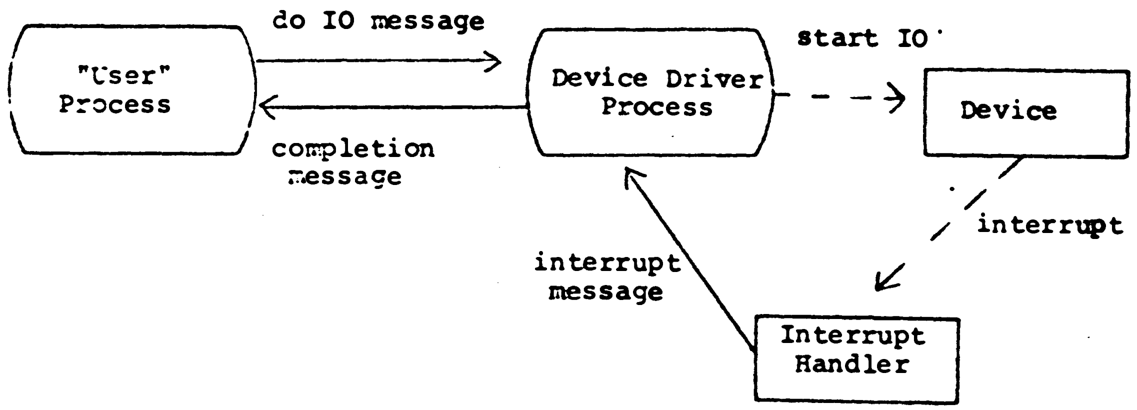
Finally, an interrupt message has the format:

1. driver process name
2. device status

DOIO, COMP, and INT messages therefore have lengths 5, 2, and 2, respectively.

Figure 4

IO and Interrupt Processing



With these assumptions, we can now declare the message classes:

```
DOIO: message class ("directed",5);
COMP: message class ("directed",2);
INT:  message class ("directed",2);
```

To activate a driver process, a user does a Send of a DOIO message describing the operation to be performed. Completion of the operation is awaited by executing a Receive of a COMP message. With this approach, a user can do other work before awaiting device completion. For example, a process logging a user onto a terminal could send an acknowledgement to the terminal and a message to the operator's console, or system log, before waiting for either completion. In fact, the order of completion does not matter, since messages are stored by type, not by who sends or receives them.

An IO driver process in this scheme has the basic form:

```
do forever;
    Receive(DOIO, IO_message)
    build channel program
    Start IO
    Receive(INT, status)
    Send(COMP, status)
    end;
```

When a DOIO message is received, recall that the sender's name is returned as the first parameter. Therefore, the IO driver knows who to send the completion message to. Notice also that because messages are grouped by type, the driver can first get a DOIO message and then an INT message. In a system such as the RC4000 [1], a process has only one queue of message buffers and consequently needs to look through the queue for the correct message if it can be sent more than one type. It is quite conceivable that DOIO messages and INT messages are not perfectly inter-leaved, one after the other. This causes no problems with message classes; it does when a process only has one message queue. Grouping messages by type, not by process, gives message classes a major advantage relative to other communication techniques.

To complete our IO handling scheme, the interrupt handler takes the following actions when entered via an interrupt:

```
Save state of executing process
Determine identity and status of device
Format interrupt message to device driver
Send(INT, status)
Restore state
```

This scheme for IO and interrupt processing has been used successfully in experimental systems at the University of Washington [4,5] and in student projects at Cornell. In both cases, the overhead has been quite tolerable and any slight inefficiencies have been more than offset, we feel, by a clearly structured approach. Because DOIO and COMP act as large switches with "users" on one side and device drivers on the other, it is very easy to add or delete device drivers or users without any change to the coordination scheme. And the use of distinct drivers for each device makes it easy to concentrate on the special channel program structure or other constraints imposed by the device. Although we have defined two large classes, DOIO and COMP, for user-driver communication, there is no reason why separate message classes for each type of device couldn't be used. This might be useful if IO message formats differ from device to device[2]. It would also shorten the lengths of the Avail_list and Wait_list within the class descriptor thus decreasing the execution time of Send and Receive.

Before leaving this example, we would like to point out one further use of message classes. For serial devices, such as card readers or line printers, it is necessary for one user at a time to reserve the device, do his IO, and then release his reservation. In the previous example, it was shown how to use message classes to implement mutual exclusion. For device reservation, however, a user needs to know what device to use. For this, we can use a message class to implement a pool of driver process names. A user then receives a name from the pool, does IO by communicating with that driver, and then sends the name back to the pool. The implementation goes as follows. Let a message class DEVICE be declared by:

---

[2] Separate classes per device would also be useful if other than FIFO allocation of messages were possible. Then more efficient scheduling of devices such as drums could be achieved. This point is considered further in Section 5.

DEVICE: message class ("undirected",1);

and initialized by as many

Send(DEVICE, driver name message)

operations as there are devices in the pool. A user then does IO within a critical section preceded by a Receive(DEVICE, driver name) and followed by a Send(DEVICE, driver name).

This same technique can also be used to control the allocation of other classes of serially reusable resources; for example, memory blocks or channels. But it must be used with caution, because a message class uses lists of resources which can be inefficient for keeping track of such things as free pages or drum records where a bit map may be more appropriate. Message classes can still be used, however, to synchronize the allocation and thereby preserve the integrity of the data structures used.

3.3 Buffer Passing in a Spooling System

The previous examples illustrated the use of message classes for mutual exclusion, process communication and reusable resource allocation. In a spooling system, buffers are typically used to hold card or line images. These buffers are effectively reusable resources requiring exclusive access, but they are usually passed between processes as spooling proceeds. Suppose we have an input spooling system with two main processes, one to read cards and the other to write card images to the disk. The card reader process cyclically acquires an empty buffer, fills it, and passes it on to the disk writer. The disk writer in turn acquires full buffers, transfers them to the disk and then releases an empty buffer. This coordination is depicted in Figure 5. An output spooling system would perform analagous operations.

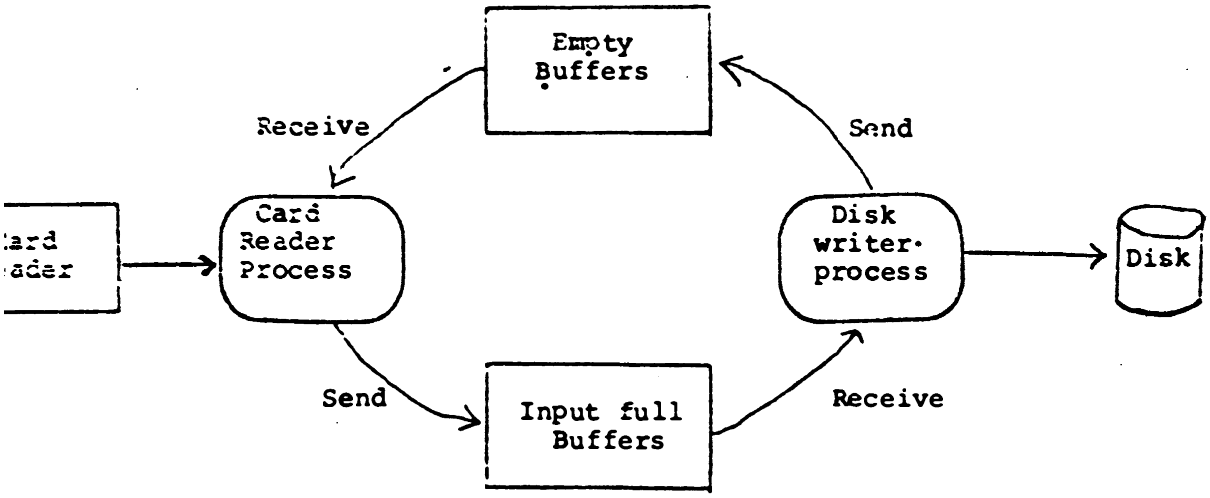
To synchronize the execution of the reader and writer processes, we can use message classes to form pools of empty and input-full buffers. Each buffer is represented by a "message" with one item of data: a buffer address. The classes are declared as follows:

```
EMPTY: message class ("undirected",1);
INPUT_FULL: message class ("undirected",1);
```

If there are initially N empty buffers, and no input full ones, the EMPTY class is initialized by N calls of Send(EMPTY, buffer address message), each message containing a different address. The two processes then use the classes by executing code such as that shown in Figure 5. At all times there are exactly N buffers in use or in classes. Each changes type as spooling proceeds but the total number is conserved. The <u>same</u> sychronization scheme using a single buffer pool can also be used if there are more readers and/or writers. <u>All</u> control needed to allocate buffers and synchronize their use is taken care of by message classes. A multiprogramming system having semaphores and/or inter process messages for synchronization would need to use both <u>and</u> have extra buffer management procedures in order to synchronize multiple readers and writers.

Figure 5

Buffer Passing in an Input Spooler




**Reader Process**

```
do forever;
   Receive(EMPTY, answer);
   read card into buffer;
   process it;
   Send(INPUT_FULL, answer);
   end;
```

**Writer Process**

```
do forever;
   Receive(INPUT_FULL, answer);
   write to disk;
   Send(EMPTY, answer);
   end;
```

## 4. IMPLEMENTATION

In order to implement message classes, it is necessary to code the Send and Receive procedures and manage storage space for descriptors and list entries. In addition, there needs to be some means for defining each class. In our implementation, we have proceeded as outlined below. Within an operating system nucleus which also implements processes, Send and Receive exist as user accessible primitive operations executed with interrupts inhibited. In addition, a primitive Create_message_class exists to build class descriptors. It is called by:

Create_message_class(name, type, length)

and dynamically implements the static message class declaration which we have been using. Create has an area of descriptor storage from which it selects a block of space whenever called. In practice, Create_message_class is called during system initialization; if users can define classes it could also be called during user execution. Its function is to remember the class name and initialize the descriptor (see Figure 1). Initially both the available and waiting lists are empty.

Implementing Send and Receive requires a few support procedures to look up the class name and find its descriptor, manage the Avail_list and Wait_list, and block or wakeup (i.e. schedule) processes. When Send, Receive and Create are each indivisible primitives, the most efficient storage management for available list entries is to have a pool of free storage, implemented by linked lists. When a message needs to be saved by Send, one node is removed from the free list, filled with the message, and inserted on the appropriate available list. Conversely, when a message is allocated by Receive, the message node can be returned to the free list.

One of the attributes of a message class is the length of its messages. Since this can vary from class to class, it has to be taken into account in managing free storage. One approach is to have one size of message node for all classes and just fill as much as needed for each type of message. Another approach would be to have pools of different size message nodes, managed by using the buddy system perhaps [3], and then have Send select a node of the appropriate size. The first approach is simple but might waste some storage; the second requires a little more overhead but achieves much better storage utilization if message lengths vary. Conceivably, one could instead have a separate pool of free space for each class but there appears to be no advantage to this approach. With a common pool an over-abundance of messages can be queued in one class as long as other classes have short Avail_lists. Fluctuations in Avail_list size cannot be handled by a pool per class without wasting a good deal of storage.

Regardless of the technique used to implement free storage for messages, it is possible that free space will be exhausted.

As long as enough space is allocated for all expected messages, this should only happen if one (or more) processes have erroneously or maliciously generated many messages. A suitable recovery action in such an event would be to destroy the offending process. More generally, the exhaustion of free space can be prevented if each message class or each process has a maximal claim associated with it and the sum of the claims is never allowed to exceed the available storage.

In order to implement the Wait_list, it is natural to link wait list entries through process descriptors. A process then is always on a Wait_list or a ready list [4,5]. The same linkage space in the process descriptor can be used for both purposes.

The components and relationships of a representative implementation of message classes are depicted in Figure 6. It is our experience that a Send or Receive operation results in the execution of about 250 machine instructions on an IBM S/360 or S/370. This includes all list manipulations and process scheduling (using priorities). Therefore, the time during which interrupts are inhibited never exceeds a half a millisecond. Interrupt handling, when done as outlined in Section 3, takes about 300 instructions, 50 or so for state saving and acknowledgement of the interrupt followed by 250 to Send the interrupt message. The approach of using message classes therefore appears feasible for all but possibly the most time sensitive applications.

## 5. EXTENSIONS AND CONCLUSION

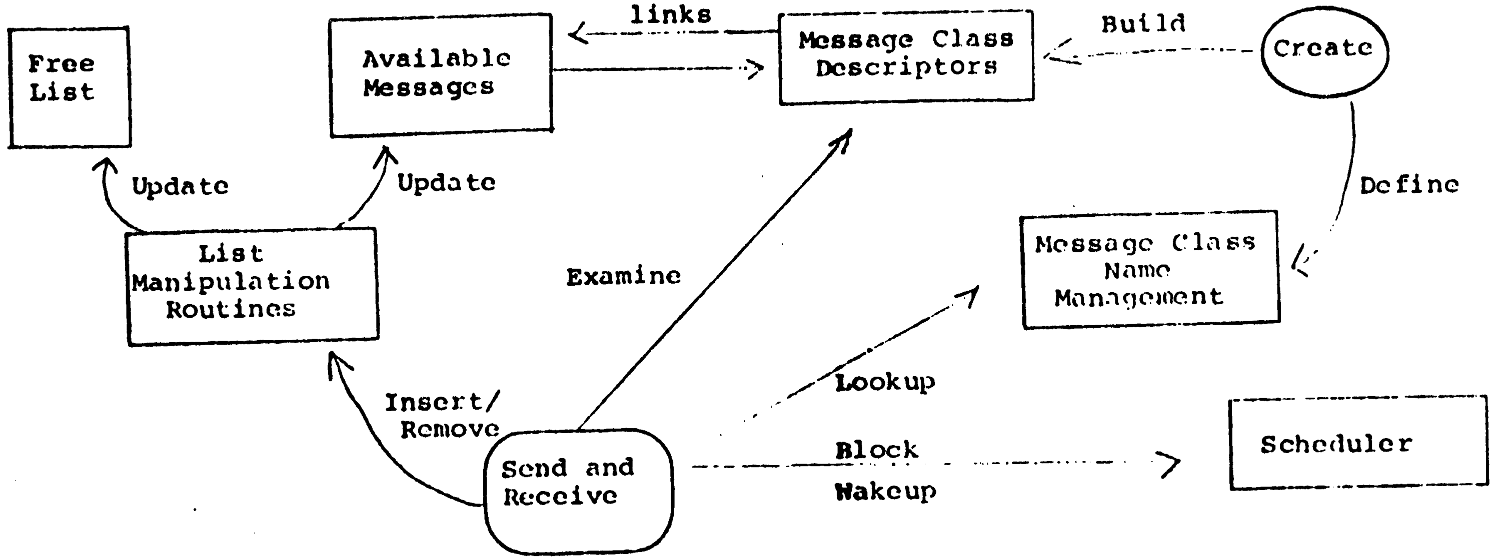
In this paper, we have described an approach to process synchronization, called message classes, which can be used to solve mutual exclusion, message passing and resource allocation problems. As such, it is more powerful than either semaphores or inter-process messages by themselves. In addition, it can be implemented as cheaply as inter-process messages. The only extra cost with respect to semaphores is the use of list manipulation to decrease or increase the value of the "semaphore". Our students have also found message classes easy to understand, implement, and use.

Three possible extensions to message classes as they have been defined suggest themselves. First, for many problems it is desirable to order available messages. Some tasks might be more urgent than others, for example. And in scheduling activity on a drum or disk, efficiency improves greatly if accesses are ordered to minimize rotational delay or movement of heads. In order to attach a priority order to available messages, it is necessary to do something more than a queue (FIFO) insert on the Avail_list. If one field of a message is designated as the key, an insertion routine could then insert the message into the appropriate place. This, of course, takes longer in general than insertion at the end of the list but may well be worth the overhead. A "library" of list insertion routines could be implemented

Figure 6

Implementation of Message Classes:

Components and Connections



and the appropriate one selected when a class is created (via an extra parameter).

The second extension is to allow a choice of list structures for storing the available list. We previously mentioned the use of message classes to control memory page or drum sector allocation. If bit maps or some other data structure can be selected when a class is created, more problems could be efficiently solved with message classes. The price is implementing extra insert and remove routines.

The final extension is to protect access to the classes. It may be desiralbe to limit the use of a specific message class to just a few processes thus preventing others from executing Send or Receive on the class. In addition, it might be useful to distinguish between producers who only Send a certain type of message and consumers who only Receive it. Both situations can be controlled by either storing capabilities with each process, identifying the classes and operations it can access, or by storing access lists of process names with each message class. If message classes cannot be created dynamically by users, however, this type of protection is probably not worth the expense unless it is necessary to dynamically change the set of resources a process can access.

ACKNOWLEDGEMENTS

I am most grateful to Alan Shaw who inspired this approach to process synchronization, and to Susan Owicki and James McGraw who provided helpful comments on a draft of this paper.

**BIBLIOGRAPHY**


1. Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13,4 (April 1970), 238-250.

2. [illegible] The structure of the "THE" multiprogramming system. Comm. ACM [illegible] 341-346.

3. Knuth, D.E. [illegible] Programming [illegible] Vol. 1. [illegible] 1968, pp. [illegible]

4. Sh[illegible] Operating Systems, [illegible] 1974, pp. 166-191.

5. Shaw, A.C. et al. A [illegible] with dynamic [illegible] Software Practice and Experience 5 (1975), 24[illegible]-26[illegible].