Using BINS for Inter-Process Communication

Peter C.J. Graham
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada
R3T 2N2

ABSTRACT

A bin is essentially a message drop, used to implement inter-process communications in the **FRANK** programming language. The use of bins provides three distinct advantages over traditional message passing systems.

1) relative anonymity,
2) multiple servers,
3) enhanced compile time checking.
   (relative to systems like Thoth[3]).

This paper discusses the rationale behind bins and the possible implementation strategies under examination. Finally, it gives two examples using bins which will illustrate the concept.

DESCRIPTION

FRANK[1] is an experimental system implementation language under development at the University of Manitoba. It features good abstraction and modularization as well as concurrency and exception handling. The area of interest to this discussion is the means of communications between processes. The facility proposed, bins, arose out of dissatisfaction with existing systems.

After examining the use of common messaging primitives in a variety of applications, they were found to lack not only generality but also consistency. The basic complaints with the most common of these, the SEND/RECEIVE strategy, were the following:

[1] Designers: Glen J. Ditchfield, Peter C.J. Graham, Peter R. King, Michael J. Rogers, Allan J. Stephens & Frank J. Wiebe.

1) lack of generality.
   - There are desired capabilities which are not present[2]. For example, multiple servers are not usually supported.
2) programmer controlled synchronization.
   - the programmer must control synchronization of access to shared data. (Having the programmer code SEND_WAIT rather than SEND provides an oppurtunity for error. If instead, the detection of required synchronization is performed by the machine, the chance of error is lessened.)
   - this intertwining of message passing and synchronization is necessary but should not be left to the programmer.[2]
3) lack of anonymity.
   - the programmer must explicitly manage process identification information to accomplish the aforementioned synchronization.

Thus, the goal for **FRANK** was to increase generality while still hiding as many of the implementation details as possible from the programmer.

A bin is simply a place, with an associated type, where messages may be left. The type is used to define what may or may not be placed in the bin. **FRANK's** strong typing rules allows compile time checking of bins. It was felt that having distinct bins for distinct functions was superior to allowing a bin to serve more than one purpose. In keeping with this idea, a bin is capable of accomodating only a single type of message. Thus, strong typing is enforced in order to avoid the possibility of misplacing a message.

Consider the following **FRANK** declarations:

```
mtype : TYPE = [ a:int ; b:real ]
msg   : mtype
drop  : BIN OF mtype
```

The first statement declares a "constructed" (record) type consisting of an integer field 'a' and a real field 'b'. The second statement declares a variable 'msg' of that type and the third a bin which may accept such messages.

There are two fundamental operations which may be performed by processes on such a bin:

'PLACE' and 'TAKE'

The PLACE operation puts a message into the specified bin while the TAKE operation removes a message from it. Thus, in its simplest form, a bin may be used as an information pipe between processes. In other forms, much more may be accomplished.

[2] An interesting comment on this may be found in : "Messages vs. Remote Procedures is a False Dichotomy" by M.L. Scott, SIGPLAN Notices, Vol.18, No.5.

More than one process may place data in or take data out of a given bin. This allows a very general system with many producers and consumers accessing the same bin. This ability is subject to certain scoping rules. Clearly, a shared bin must be visible to all processes using it. This means that it must be declared at a logically higher level in the scope hierarchy of the program. Thus, in most instances a bin will be accessed globally. Alternately, a process could be passed a bin as a parameter. Since the called process would be a child of the caller (to whom the bin is visible), the bin must still have been declared at a higher level. Finally, a bin might be exported by a package. If this is the case, the bin is considered to be global to the importing module.

Such a general system remedies the problem of lack of functionality in SEND/RECEIVE systems. It does however leave two questions unanswered:

"How is synchronization accomplished?"

"How can a priorized message be implemented?"

The answer to the question of synchronization is very simple indeed. All synchronization required is performed by **FRANK** without any specific programmer activity. This is accomplished by examining the message type for each bin. In order to make this clear, an example is in order.

```
btype : TYPE = [ a:INT ; b: ALTER REAL ]
abin  : BIN OF btype
```

In this example, one of the fields in each message in 'abin' has the "ALTER" attribute indicating that it may be altered by whoever processes the message. (ALTER is also used to indicate a "copy-out" parameter on subroutine invocation). When one or more of the fields in the message has the ALTER attribute, **FRANK** will automatically suspend the "placing" process as appropriate. The use of ALTER does not cause blockage until the altered variable is referenced. Thus, it should be noted that the use of ALTER is not equivalent to the use of SEND-WAIT rather than SEND.

By making the messages placed in a BIN similar to a parameter list, a significant advantage is gained. All programmers are familiar with the concept of parameters being either alterable or fixed. This means that there is less chance of an error being made because a programmer is unfamiliar with a multi-process environment. Remembering to ensure correct synchronization is no longer necessary.

Besides providing a means by which **FRANK** can control synchronization, the ALTER attribute also provides a "return" capability. This allows a server to return data or status information through the ALTER fields. The classic example of this is a file server which accepts a read request containing an empty ALTER field into which it will place the retrieved record.

The answer to the question of priorization lies in the syntax of the PLACE and TAKE statements. If the programmer so desires, he may associate a "code" (of an enumerable type) with each message during the PLACE, thereby indicating its priority.

```
          coded_bin:BIN OF some_type CODETYPE value_type

          PLACE msg IN bin CODE value
```

On the "TAKE" statement, a facility exists for selecting messages with appropriate codes. "TAKE" has the syntax:

```
          TAKE msg FROM bin
            <takebody>
            ENDTAKE
```

The "<takebody>" may consist of a number of statements which will be executed regardless of code value or it may consist of a "CODE" construct. The "CODE" construct is essentially a case statement based on the message's code. The order in which the cases occur defines the relative priority of each code and its associated messages. Consider the following:

```
          TAKE msg FROM bin
            CODE 1,2 DO
              high priority.
            CODE 3 DO
              next priority.
                   -
                   -
                   -
            OTHERWISE
              lowest priority.
          ENDTAKE
```

When ALTER fields are contained in the message, the placer is suspended only until the taker is finished with the message. (i.e. until the take body has been executed.) After the ENDTAKE, the taking process must not access a message if it contains ALTER fields or the ALTER fields themselves. This may be checked at compile time as may the restriction that no non-ALTER message field may appear on the left hand side of an assignment. Note that while the message variable ('msg' in the above example) is not declared locally to the takebody, it may be thought of as being local. References to a message variable outside of a takebody do not make sense and will be considered semantically illegal in **FRANK.**

This syntax also allows the association of code values with priorities to be different for each taker. Thus, one server may treat a message with code '1' as being more important than one with code '2' while another server may do the opposite. This is particularily useful in writing schedulers where the various scheduling processes may wish to select jobs with different execution requirements.

Note that the OTHERWISE clause is optional on a TAKE. If it is not included, and if no code clause matches the code value, then an exception is raised. If no exception handler exists, the process which placed the message with the invalid code will be terminated. Should this happen, the taking process will simply return to the beginning of the TAKE statement in order to attempt to take another message from the bin.

A PLACE operation may always be performed on a bin. There is no conceptual limit on the number of messages which may be queued in a bin. A TAKE from an empty bin will result in the taking process being suspended until a message is available in the bin. This is not a busy-wait.

It should now be clear that **FRANK** is an anonymous system. No process needs to know the process-id of any process it is communicating with.[4] It is also possible to receive process-id information explicitly (as a field in the msg) should it be so desired. This anonymity is a desirable feature in any encapsulated system such as **FRANK**.

IMPLEMENTATION

Each bin must consist of at least two parts:

1) a synchronization mechanism,
2) a reference to a list of messages.

Since more than one process may be simultaneously attempting to access a given bin, a facility for mutual exclusion is required. The simplest approach to solving this problem is to make use of an access semaphore. Implementation of such a semaphore is exceedingly simple on most machines. Even many microprocessors provide indivisible operations such as TAS (Test And Set) on the MOTOROLA 68000 which are suitable for this purpose.

Once a process has acquired access to the bin, via a sucessful "P" semaphore operation, a new message may be added or an existing one removed as appropriate. Although **FRANK** does not define a processing order for messages in a bin, a simple FIFO organization seems appropriate.

Thus, the physical structure of a bin is the following:

sema	@first	@last

Each message within the chain delimited by the pointers "@first" and "@last" will contain at least three fields in addition to the message text. They are:

1) sender's process-id,
2) wait flag,
3) link field.

[4] This anonymity applies only to inter-process communication as a process-id is required for process creation/deletion purposes.

Additionally, a code field will be required for coded messages and a semaphore must be present to allow suspension if the message contains alter fields.

The sender's process-id is used for reactivation purposes. Reactivation occurs upon exit from the take body if the wait flag is set. At this point, the modification of all ALTER fields is guaranteed to be complete and the "placing" process may resume execution.

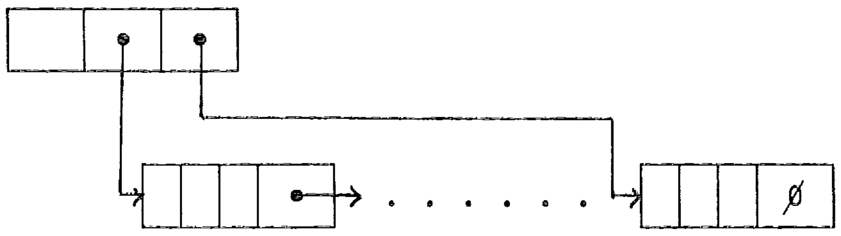
The format of the text portion of the message is determined by the type of the message. Whether the data is actually copied or whether an address is passed is unimportant. What is important is that ALTER fields should behave in a "copy-in/copy-out" fashion while non-ALTER fields will behave as "copy-in" only.

This implementation strategy is subject to inherent inefficiencies when dealing with coded messages. If a small enumeration or subrange is used as the code type then these inefficiencies may be removed by implementing a unique queue for each code. If for instance, a sparse subset of the integers is used, then this is impractical. Furthermore, all members of the subset which will actually be used are not determinable at compile time. In such a situation one must either accept the inefficiency of a linear search within the queue or use a more elaborate data structure (Eg. a dynamic index, heap or multiple queues) with increased overhead.

If we assume the simple implementation of bins, then we will have a basic message which will look like:

sender-pid	wait flag	message text	link

and a bin structure which will look like:



EXAMPLES

The first example is an interesting solution to the "protected variable" problem. This example is courtesy of Glen Ditchfield and is taken from the **FRANK** Reference Manual[4].

```
VARBIN: TYPE= BIN OF VARTYPE CODE REQUESTS;  # define a bin type.
REQUESTS: TYPE= < READ,WRITE >;              # an enumeration.
REQBIN, DATABIN: VARBIN;                     # two bins are used.
```

```
DUMMY,VAL: VARTYPE;                                # variables for bins.

VARSERVER: TYPE= PROC[ REQBIN,DATABIN:VARBIN ]; # declare proc. type
SERVER: PROCESS FOR VARSERVER;                     # and process.
SERVERPROC: VARSERVER = BODY                       # declare server.
   DECLARE
      MSGV,PROTV: VARTYPE;
   BEGIN
      TAKE PROTV FROM REQBIN                     # start up.
         CODE WRITE DO
            NOTHING
      ENDTAKE

      LOOP                                 # do forever.
         TAKE MSGV FROM REQBIN             # accept a request.
            CODE WRITE DO                  # is it a write request?
               PROTV <- MSGV;
            CODE READ DO                   # is it a read request?
               PLACE PROTV IN DATABIN;
         ENDTAKE
      ENDLOOP
   ENDBODY;

FORK SERVER USING SERVERPROC [ REQBIN,DATABIN ];  # create a process.

# to write do;
PLACE VAL IN REQBIN CODE WRITE;

# to read do;
PLACE DUMMY IN REQBIN CODE READ;
TAKE VAL FROM DATABIN;
```

The following is a more straightforward example of the use of bins. The code shown handles applications where several processes must be able to concurrently read and write the same file. In such an application, any number of reading processes may access the file simultaneously but a writing process must have exclusive access. This example also demonstrates some of the support for packages provided in **FRANK**.

```
PACKAGE MULTI_ACCESS;        # this is a package.

EXPORT STREAD;STWRITE;ENDREAD;ENDWRITE    # export the synchronization
                                          #  functions.
DECLARE
   CNTBIN:TYPE=BIN OF INTEGER CODETYPE INTEGER;

   RWBIN:CNTBIN;                # the read/write bin.

   NO_READING:INTEGER<-0;       # compile-time assignment.

   STREAD:PROC = BODY           # routine to start reading.
      BEGIN
      TAKE NO_READING FROM RWBIN; # retrieve reader count.
         NO_READING<-NO_READING+1 # increment count.
      ENDTAKE
      PLACE NO_READING IN RWBIN CODE NO_READING;  # restore.
      ENDBODY

   ENDREAD:PROC = BODY          # routine to end reading.
      BEGIN
      TAKE NO_READING FROM RWBIN; # retrieve reader count.
         NO_READING<-NO_READING-1 # decrement count.
      ENDTAKE
      PLACE NO_READING IN RWBIN CODE NO_READING;  # restore.
      ENDBODY

   STWRITE:PROC = BODY          # routine to start writing.
      BEGIN
      TAKE NO_READING FROM RWBIN; # retrieve reader count.
      CODE 0 DO                   # no one reading.
         NOTHING
      ENDTAKE
      ENDBODY

   ENDWRITE:PROC = BODY         # routine to end writing.
      BEGIN
      PLACE NO_READING IN RWBIN CODE 0;     # release write access.
      ENDBODY

   BODY
   BEGIN
   PLACE NO_READING IN RWBIN CODE 0;    # initialize the reader bin.
   ENDBODY

ENDPACKAGE
IMPORT MULTI_ACCESS;

#
# reading processes do:
#
STREAD;
# use file freely for reading.
ENDREAD;

#
# writing processes do:
#
STWRITE;
# use file freely for writing.
ENDWRITE;
```

CONCLUSIONS

In comparing bins to a variety of different inter-process communication strategies, the following observations may be made. Bins provide a high degree of anonymity that is lacking in Ada rendezvous, and in remote procedure call. They also permit selective processing of "messages" (via codes) more directly than monitors do. Bins additonally offer a non 1:1 mapping between sender and receiver which is uncommon in SEND/RECEIVE systems.

This is not to say that bins are free from problems. There is no easy way of selecting a message, determining that it either should not, or cannot, currently be serviced and return it to the bin for later processing. Simply to place the message back in the bin is insufficient for two reasons. As currently defined, the placing process waits only until a taker has processed the message. If the removal and replacement of a message counts as a processing, then the placer may resume prematurely. This can clearly be handled as a special case in code generation and therefore does not present a serious problem. More serious is the fact that there is no facility to ensure that a replaced message is not immediately re-taken. This could perhaps be solved via a special REPLACE statement or by modification of the existing PLACE statement.

Another potential problem (or at least a point of contention among the designers) is the question of the limitations of codes. This manifests itself in two complaints. First, codes may be only ordinal types and second the CODE clause of the TAKE statement accepts only a constant rather than an expression as is permitted in a PL/I SELECT. For most applications, these problems are insignificant but for others they make programming an inconvenience. Whether or not the existing facility is modified remains to be seen.

Nevertheless, bins provide an extremely flexible yet simple way of implementing inter-process communication. Deciding to provide a lower-level facility, than for instance Ada, provides a distinct advantage in supporting the implementation of high level facilities. The use of the message drop approach and of ALTER guarantees a minimum of programmer-visible implementation details. Compile time type checking is also enhanced using bins. Finally, bins appear to be straightforward to implement and should prove to be no less efficient than current message passing systems.

An effort will soon be made to embed bins within an existing language (likely C with its preprocessor) and the practical experience gained from this may influence future work on bins.

REFERENCES

[1] Brinch Hansen, Per.. The Programming Language Concurrent Pascal "IEEE Transactions on Software Engineering", June 1975, pp.199-207.

[2] Cashin, Peter M.. Inter Process Communication Internal B.N.R. document, Reference: 8005014, June 1980.

[3] Cheriton, David R.. The Thoth System: Multi-Process Structuring & Portability "Computer Science Library", pp.55-74.

[4] Ditchfield, Glen, etal. Preliminary FRANK Reference

Manual. (to be published as a University of Manitoba Scientific Report)

[5] Hoare, C.A.R.. Communicating Sequential Processes "Communications of the ACM", August 1978, pp.666-667.

[6] Hoare, C.A.R.. Monitors: An Operating System Structuring Concept. "Communications of the ACM", October 1974, pp. 549-557.

[7] Mitchell, James G. ,etal. Mesa Language Manual Version 5.0 XEROX Palo Alto Research Center report #CSL-79-3.

[8] Nelson, Bruce J.. Remote Procedure Call XEROX Palo Alto Research Center report #CSL-81-9.