Name: CoFork

Maintainer: Sturgis

Date: August 7, 1981

Purpose: Provides for the convenient start up and shut down of Co-Routines which use Mesa PORTS.

Filed as: CoFork.Bravo

Files: There are 3 files:

CoFork.mesa, .bcd -- a definitions file
CoForkImpl.mesa, .bcd -- the implementation
CoForkDemo.mesa -- some examples of use

How to use:

You should think of your main program as a Consumer which "consumes" items "produced" by a Producer. The consumer from time to time requests items by calling getItem, and the producer from time to time produces items by calling putItem. getItem and putItem are pointers to PORTS with complementary type.

Begin by defining an ItemType and a RequestType, followed by two pointer to port types and one procedure type:

```
ItemSource: TYPE = POINTER TO PORT[RequestType] RETURNS[ItemType];
ItemSink: TYPE = POINTER TO PORT[ItemType] RETURNS[RequestType];
ItemStopper: TYPE = PROCEDURE[RequestType];
```

Notice that the two port types are complementary, and that the item stopper takes a request type as a parameter. Also, ItemType and RequestType need not be single types, they can be any sequence of types which could appear as a sequence of parameter types. Frequently the request type is simply a Boolean that indicates another item should be delivered.

WARNING: Both ItemType and RequestType must be "short" parameter types. This is a maximum of 5 words in the Mesa 6 Alto implementation.

The item producer will make port calls on ItemSink to deliver items, while the consumer will make port calls on ItemSource to obtain items. The producer is a procedure which terminates by returning, while the consumer signals termination by calling ItemStopper. These two termination events must both occur, but can occur in either order.

Special items and requests should be defined which are used to signal termination.

```
NullItem: ItemType;

NullRequest: RequestType;
```

(Of course, if ItemType or Request type are actually sequences, these specific declarations could not occur)

Now write the producer; this is a procedure which accepts some initialization parameters, an initial request, and an ItemSink. The producer calls the ItemSink from time to time to deliver an item, and to receive the next request. This Port call can occur nested any number of procedure calls deep. The producer should examine the new request to see if it suggests termination. The producer can terminate

before seeing a NullRequest, or any time after seeing a NullRequest. The producer can even continue to produce items after seeing a NullRequest (they will be absorbed, and a NullRequest will be returned).

```
Producer: PROCEDURE[param1: Initial1, ... paramN: InitialN,
                                                    initialrequest:
RequestType,
                                                    putItem: ItemSink]
=
                                        BEGIN
                                        . .
                                        newRequest _ putItem[item];
                                        . .
                                        END;
```

Having written a Producer, the user next writes the following 6 line procedure:

```
MakeProducer: PROCEDURE[param1: Initial1, ... paramN: InitialN]
                                                    RETURNS[getItem: ItemSource,
stopItems: ItemStopper] =
                                BEGIN
                                putItem: ItemSink _ CoForkMe[];
                                Producer[param1, . . , paramN, putItem[nullItem], putItem];
                                END;
```

The form of this procedure is very rigid. It peforms several functions:

1) It starts the producer coRoutine (via CoForkMe) (CoForkMe is imported from CoFork and is implemented in CoForkImpl).

2) It defines the nullItem, (through the first call to putItem), so that CoFork can deliver this item to the consumer in the event that the Producer terminates before the Consumer calls stopItems.

3) It obtains the first request from the Consumer and delivers it to the Producer.

note: frequently a request is simply a boolean requesting one more item. In this case the MakeProducer procedure can examine the first request to decide whether to call the Producer, and if so the producer need not take a first request as a parameter. e.g. If a RequestType is such a BOOLEAN, then the second code line of the MakeProducer procedure could be:

```
IF putItem[nullItem] THEN Producer[param1, . . paramN, putItem];
```

and the Producer would not have an "initialRequest" parameter.

4) The consumer obtains control at the time that MakeProducer makes the first call on putItem. The consumer obtains control as if its call to MakeProducer had returned with the expected two parameters. This return is faked by CoForkMe, which intercepts the first call on putItem.

5) Since CoForkMe fakes the return to the consumer, the compiler does not type check the return from MakeProducer to see that exactly the correct two parameters are returned. Thus the user must be sure that MakeProducer returns exactly the two parameters in the correct order. In addition, the return parameters from MakeProducer should be named so that the compiler does not object when the code for MakeProducer does not explicitly return the two parameters.

6) MakeProducer regains control the first time the consumer calls getItem. From then on control alternates between the consumer and the producer.

Finally write the Consumer with the following general form:

```
BEGIN
initial: InitialParamsType;
getItem: ItemSource;
stopItems: ItemStopper;
item: ItemType;
request: RequestType;

   --
   --

[getItem, stopItems] _ MakeProducer[initial];

   --
   --

item _ getItem[request]; -- this call may occur as often as desired and nested to any procedure
call depth.

   --
   --

stopItems[NullRequest]; -- it is important that this call contain NullRequest, and it is essential that
getItem and stopItems not be called after this call.
   --
   --

END;
```

RESTRICTION: Once stopItems has been called, do not call either stopItems or getItem, DISASTER will result.

Review:

1) Define ItemType and RequestType.
2) Define two pointer to Port types, ItemSource and ItemSink.
3) Define ItemStopper type.
4) Define NullRequest and NullItem.
5) Write the Producer procedure.
6) Write MakeProducer.
7) Write the Consumer.

A Simple Example:

ItemType will be a pair, <n: CARDINAL, "nIsNull: BOOLEAN> (Not a RECORD).
RequestType will be <moreWanted: BOOLEAN>.
nullItem is <anything, TRUE>.
nullRequest is FALSE.

```
NumberSource: TYPE = POINTER TO PORT[moreWanted: BOOLEAN]
            RETURNS[n: CARDINAL, nIsNull: BOOLEAN];
```

```
NumberSink: TYPE = POINTER TO PORT[n: CARDINAL, nIsNull: BOOLEAN]
        RETURNS[moreWanted: BOOLEAN];

NumberStopper: TYPE = PROCEDURE[moreWanted: BOOLEAN];

NumberProducer: PROCEDURE[first, last: CARDINAL, putNumber: NumberSink] =
        BEGIN
        FOR I: CARDINAL IN [first..last] DO
                IF NOT putNumber[I] THEN EXIT;
                ENDLOOP;
        END;

MakeNumberProducer: PROCEDURE[first, last: CARDINAL]
                RETURNS[getNumber: NumberSource, stopNumbers: NumberStopper] =
        BEGIN
        putNumber: NumberSink _ CoForkMe[];
        IF putNumber[0, TRUE] THEN NumberProducer[first, last, putNumber];
        END;

MainCode (Consumer)
        BEGIN
        getNumber: NumberSource;
        stopNumbers: NumberStopper;

        [getNumber, stopNumbers] _ MakeNumberProducer[5, 25];
        FOR I: CARDINAL IN [0..100) DO Print[getNumber[TRUE].n] ENDLOOP;
        stopNumbers[FALSE];
        END;
```

More Examples:

See CoForkDemo.mesa for more examples.

Execution

We describe here the sequence of events which occur during execution of a producer and a consumer.

The Consumer calls MakeProducer

MakeProducer calls CoForkMe

CoForkMe constructs a co-routine with two procedure frames. The base frame is CoForkMe, and CoForkMe appears to have called MakeProducer.

The local frame of CoForkMe contains the two ports.

While retaining its frame in existence, CoForkMe tfakes a return to MakeProducer, returning the sink port.

MakeProducer then calls the sink port (passing nullItem).

CoForkMe captures this call, and fakes a return from MakeProducer to the Consumer. This faked return returns the source port and the stopper procedure to the Consumer. The CoRoutine continues in existence with the two frames as described above.

The Consumer calls the source port (with the first request).

This is a port call resulting in a port return to MakeProducer (from its call on the sink port),

delivering the first request.

MakeProducer now calls the Producer.

The Producer calls the sink port delivering an item.

This results in a port return to the Consumer, delivering the first item.

Etc.

There are two ways for this sequence to terminate:

A) The Producer returns to MakeProducer.

(At this time the consumer is waiting for an item.)

MakeProducer returns (to CoForkMe)

CoForkMe makes a port call on the sink port, delivering the null item (which it received from MakeProducer during the start up sequence). This port call results in a port return to the consumer.

The consumer can make more port calls on the source port.

CoForkMe receives each one, and recalls the sink port with NullItem.

Finally the consumer calls the stopper.

CoForkMe receives this call, shuts down the co-routine, and returns to the consumer.

B) The Consumer calls the stopper.

(At this time the producer is waiting for the next request.)

CoForkMe receives this call, and makes a portCall on the source port delivering the nullrequest, which it just received via the call by the consumer on stopper. This port call results in a port return to the producer.

The producer can make more port calls on the sink port.

CoForkMe receives each one, and recalls the source port with the null request.

The producer returns to MakeProducer.

MakeProducer returns (to CoForkMe).

CoForkMe returns to the Consumer following the call to the stopper.