Published: 12/10/68
(Supersedes: BQ.6.04, 07/25/67,
BQ.6.06, 07/27/67)

Identification

The User-ring IPC
Michael J. Spier

Purpose

The User-ring IPC (henceforward referred to as "IPC") is a collection of procedures in segment <ipc>; it is available in all non-hardcore rings and is responsible for the management of the process' Event Channel Tables (ECT).

Introduction

The IPC consists of three major collections of procedures,

1. Procedures to create, destroy and maintain event channels; these procedures are collectively referred to in this writeup as the "event channel manager".

2. The process' wait coordinator (WC); these procedures keep track of all the event signals received by this process. They file incoming event messages into the appropriate event channels and retrieve event messages (or call block to suspend the process' execution until some event message arrives) whenever the process' flow of control demands it.

3. Procedures to set control variables associated with the wait coordinator, collectively referred to as WC-controller. The WC-controller gives the user a handle with which to control the internal logic of the wait coordinator.

There is an ECT per non-hardcore ring (see BJ.10.02). The IPC manipulates the ECT of its current ring as well as those of all outer rings (the IPC, executing in ring n can manipulate the ECTs of rings n->63). To this end it maintains in segment <process_info>, which is readable in all rings, a table of pointers to all (potential) 63 ECTs of this process.

An event channel name uniquely identifies an event channel entry in a specific event channel table; it consists of 72 bits of information structured as follows:

```
dcl     1 event_channel_name based(p),
          2 ring bit(6),              /* the ring number of the
                                         ECT in which this channel
                                         is allocated */
          2 entry_pointer bit(14),    /* a relative pointer within
                                         the ECT to this channel's
                                         entry */
          2 key bit(52);              /* a 52-bit clock reading to
                                         uniquely identify this
                                         event channel */
```

The ipc is able to directly access any given event channel by merely "dissecting" its event channel name and reconstructing a pointer to it.

Following is a detailed description of the ipc procedures. Calling sequences are given, however the arguments' PL/1 declarations are not. They can be looked up in the IPC reference manual, section BJ.10.01. For the sake of clarity, return arguments in the calling sequences are underlined.

The event channel manager

The IPC can manipulate the ECT of either its current, or outer rings. When reaching for the ECT of an outer ring, it attempts to use the corresponding ect-pointer stored in <process_info>; if that pointer is null then that specific ECT is considered to be non-existing and an error return is made.

The ECT is a table which is allocated in the ipc's internal static storage. When IPC reaches for the ECT in its current ring, it is able to determine whether or not that ECT exists and in the latter case call entry point ipc$init which initializes the current ring's ECT, creates this ring's channel-1 and channel-2 (see BJ.10.02) and calls the Hardcore-IPC entry point hcs$ipc_init (ect_pointer) (see BJ.10.04) to entry "ect_pointer" into the corresponding slot in <process_info>.

To create an event channel in the caller's validation ring associated ECT (caution: to create an event channel in one's current ring, it is advisable to explicitly set one's validation level before making this call),

```
call ipc$create_ev_chn(chname, code);
```

this call creates an event channel in the ECT associated with the current validation ring. The newly created channel has by default the event-wait type. The call returns the event channel name of the new channel; it may return an error code (code¬=0) if reference was made to a nonexistent outer-ring ECT.

Analogous to ipc$create_ev_chn, the following two calls are provided to return the names of channel-1 and channel-2 of a given ring (validation level). As mentioned above, those two channels are automatically created by ipc$init. They are just like any normally created channel except for their name which is not unique, but rather universal within a given ring. By knowing my ring's channel-1 name I automatically know my fellow-process' channel-1 name in the same ring and can communicate with him over his channel-1. This is extremely useful in "broadcast"-type event signalling where the sending process has had no previous understanding (basic interprocess communication) with the target process. The two calls are,

```
call ipc$chn_1(chname, code);

call ipc$chn_2(chname, code);
```

A newly-created channel (including channel-1 and channel-2) has by default the event-wait type. To convert this channel into an event-call channel,

```
call ipc$decl_ev_call_chn(chname, procptr, dataptr,
                          prior, code);
```

where chname is the name of the affected event channel, procptr is a pointer to a procedure entry point to be associated with the channel, dataptr is a pointer to some data to be associated with this channel, prior is a priority number assigned to this channel by the caller and code is a variable in which ipc stores status information.

An event call channel (see BJ.10.02) is a channel which is automatically interrogated by the wait coordinator; upon reception of an event message over such a channel, the wait coordinator automatically invokes an associated procedure (pointed to by procptr) and passes to it as arguments dataptr and the received event message.

ipc$decl_ev_call_chn goes through the following steps:

1. Retrieve the channel. If unsuccessful make an error return.

2. If the channel already is an event-call channel, call ipc$decl_ev_wait_chn to reset it to the event-wait type.

3. Lookup the ECT's associated-procedure list for an entry corresponding to procptr. (Note: more than one event call channel can be associated with a single procedure). If successful go to step 5.

4. Create an associated procedure entry for procptr, thread it into the associated procedure list.

5. Increment the associated procedure entry's user-count by one. This count shows the number of channels currently associated with this procedure.

6. Create an event call trailer entry, store dataptr and prior in it and link it to the associated procedure entry and to the event channel entry.

7. Follow down the event call channel list, thread this channel into the relative location indicated by its priority number. The lower the value of prior, the closer the channel will be to the head of the list.

8. Set the channel's event_call flag to "on". Return.

An event call channel can be reset to the event wait type by calling,

```
call ipc$decl_ev_wait_chn(chname, code);
```

which goes through the following steps:

1. Retrieve the channel. If unsuccessful make an error return.

2. If the channel already is an event-wait channel, return.

3. Find associated procedure entry. If it is inhibited (procedure called by wait-coordinator but has not yet returned) this is clearly a logical error (like trying to cut off the branch on which one sits). Make an error return.

4. Decrement the associated procedure's user count by one. If count has reached zero, remove entry from associated procedure list, then delete entry.

5. Delete the channel's event call trailer entry.

6. Unthread the channel from the event call channel list.

7. Reset the channel's event call flag to "off". Return.

The following three calls are associated with the reception of event messages.

```
call ipc$drain_chn(chname, code);
```

deletes all the event message entries which may be appended to the channel's event queue. This call resets the channel to an "empty" state, discarding all received event messages.

```
call ipc$cutoff(chname, code);
```

makes the channel appear to be "empty" without actually deleting the event messages. This call sets a flag which is interrogated by the wait coordinator before it attempts to read a channel. When this flag is "on", the wait coordinator simply ignores the channel. As described above, event call channels are set up once whereupon they are automatically interrogated whenever the process executes in the wait coordinator. ipc$cutoff is very useful in temporarily disabling such channels.

```
call ipc$reconnect(chname, code);
```

resets the flag which was set by ipc$cutoff.

When an event channel is no longer useful,

```
call ipc$delete_ev_chn(chname, code);
```

which causes the channel to be destroyed. It goes through the following steps,

1. Call ipc$drain_chn to delete possible pending event messages.

2. Call ipc$decl_ev_wait_chn to delete possible event call trailers and associated procedure entries.

3. Delete the channel entry.

The Wait Coordinator

When a process reaches a point in its execution where it cannot proceed until some event (or one of several different events) occurs, it calls the wait coordinator.

The wait coordinator manages the traffic of incoming event messages by "filing" them into the appropriate event channels (by appending the message to the event channel's queue). When it is called because the process cannot continue its execution, the wait coordinator interrogates all the (not "cutoff") event channels of interest. If all of them are found to be empty, it calls the Hardcore_IPC (see BJ.10.04) at entry point hcs_$block knowing that a return will be made only if some event message was received (but not necessarily in a channel of interest).

We now define the expression "channel of interest". Some procedure has called the wait coordinator with a list of one or more event wait channel names, asking the wait coordinator not to return to its caller before it received an event message over one of these channels. However, in addition to this explicitly-stated wait list, the wait coordinator also disposes of an event-call list which implicitly states event channels of continuous interest. The wait coordinator concatenates both lists which then form the list of channels of current interest. The order of concatenation is determined by the ECT's call_wait flag; when "on", the event-call list becomes the head of the list of current interest, else it is the wait-list. The list of current interest is scanned sequentially; the order of event-wait channels is arranged by the wait coordinator's caller, the order of event-call channels is determined by their priority number.

Note that the wait list may comprise channels residing in different ECTs, whereas the wait coordinator always interrogates the call list of its actual-ring ECT only, for obvious reasons of protection.

The wait coordinator goes through the following steps (CLIST stands for the list of channels of current interest);

1. Make CLIST. This operation requires the following sub-steps:

    a. If event-calls are masked in this ring (explained below) set call-list=null.

b. Get this ring's call_wait flag to determine the order of concatenation.

c. Make CLIST by concatenating wait-list (argument of call) and call-list (of current ring's ECT) in order specified by call_wait.

2. Scan CLIST sequentially, interrogating every channel. Upon finding the first non-"empty" channel, go through the following sub-steps:

a. If this channel is an event wait channel, write the event message into argument provided by caller. Return.

b. Else, this is an event call channel. Set its associated procedure's inhibit flag to "on", then invoke the associated procedure passing to it as arguments the event message and the channel's associated data pointer.

c. Upon return from the associated procedure reset its inhibit flag; go to step 2.

3. If we arrive here, we know that all the event channels on CLIST are "empty". We call the Hardcore_IPC entry point hcs_$block. We return only if at least one event message was received for this (or an outer) ring (but not necessarily for a channel on CLIST).

4. Invoke internal procedure copy_itt_messages. The Hardcore_IPC (see BJ.10.04) has put ITT messages into the ITT transcription area of this ring's ECT (see BJ.10.02) as well as those of outer rings. These event messages belong in event channels; copy_itt_messages looks up all the event channel tables in ring-brackets (current_ring)->63, looks into the ITT transcription areas of those ECTs, and wherever it finds an ITT message it copies it into the ECTs entry-table and appends it to the appropriate channel's queue.

5. Now that all the event messages received by this process have been "filed" away into their corresponding event channels, the wait coordinator can try again to interrogate CLIST, in the hope that at least one of the received event messages was meant for a CLIST channel. Go to step 2.

Note: because, in the actual implementation, the wait and call lists have different formats and cannot physically be concatenated into a single list, CLIST is only a conceptual entity introduced here for the sake of clarity. Its actual implementation is more complex than one is led to believe from the description above.

The wait coordinator is invoked as follows,

```
call ipc$block(argptr, msgptr, code);
```

where argptr points to the wait list, msgptr points to a structure into which the wait coordinator puts the received event message, and code is a variable into which the wait coordinator stores its return status. The wait list pointed to by argptr has the following structure:

```
dcl     1 wait_list based(argptr),
        2 n_chn fixed,                  /* size of wait list */
        2 list(argptr->wait_list.n_chn) fixed bin(71);
```

The message structure has the following format:

```
dcl     1 message based(msgptr),
        2 ev_chn fixed bin(71),         /* name of channel over
                                           which message was
                                           received */
        2 message fixed bin(71),        /* a 72-bit message
                                           associated with this
                                           event */
        2 sender bit(36),               /* sending process ID */
        2 origin,                       /* origin of event
                                           message */
          3 dev_signal bit(18),         /* 0=user event,
                                           1=device signal */
          3 ring bit(18),               /* ring number of
                                           signalling procedure */
        2 wait_list_index fixed;        /* index in wait-list
                                           corresponding to
                                           "ev_chn" */
```

The wait coordinator invokes an event call associated procedure as follows:

```
call [associated_procedure] (msgptr)
```

where msgptr points to an event message similar to the one described above except that instead of item "wait_list_index" it contains the channel's associated data pointer,

```
dcl     1 associated_proc_argument based(arg_ptr),
        2 ev_chn fixed bin(71),
        2 message fixed bin(71),
        2 sender bit(36),
        2 origin,
          3 dev_signal bit(18),
          3 ring bit(18),
        2 associated_data_ptr pointer;
```

In addition to entry point ipc$block, the wait coordinator also has an entry point ipc$read_ev_chn which allows the user to interrogate one event channel without blocking the process in case the channel is "empty".

```
call ipc$read_ev_chn(chname, readmark, msgptr, code);
```

where chname is the channel to be interrogated, readmark is a flag which is set "on" if the interrogation was successful, msgptr is a pointer to an event message structure similar to ipc$block's in which a message is returned if readmark="on", code is a returned error status.

ipc$read_ev_chn goes through the following steps:

1. Retrieve the channel. If unsuccessful make an error return.

2. Examine the channel's queue to see whether or not it contains event messages. If yes, go to step 5.

3. The channel is "empty", however there may be event messages waiting for this process in the ITT. Call the hardcore_ipc entry point hcs_$read_events which retrieves the process' ITT messages and copies them into the appropriate ECTs without, however, calling the Traffic Controller's entry point block.

4. Upon return from hcs_$read_events, invoke copy_itt_messages to retrieve and file the (possibly) received messages.

5. Attempt to read the event channel. If successful, set readmark="on" and copy the event message into the caller-specified structure. If unsuccessful, set readmark to "off". Return.

The WC-controller

The WC-controller has four entry points which set the wait coordinator's control flags "call_wait" and "mask_calls".

```
call ipc$set_wait_prior(code);
```

sets call_wait to "wait" (the default setting).

```
call ipc$set_call_prior(code);
```

sets call_wait to "call".

As explained above, the setting of call_wait determines which list (wait or call) is interrogated first by the wait coordinator.

```
call ipc$mask_ev_calls(code);
```

sets the mask_calls flag to "on",

```
call ipc$unmask_ev_calls(code);
```

resets the flag.

As explained above, if event calls are masked in a given ring (mask_calls="on") then the wait coordinator assumes the call list to be null. These two calls do to a whole list what ipc$cutoff and ipc$reconnect do to a single channel.