To: Distribution

From: Steve Webber

Subject: Handling System Conditions

Date: 4/23/75

Proposed Changes to Handling System Conditions

There are several system events that occur today in a Multics process that do not fit well into the PL/I condition mechanism but were implemented as part of it for lack of a better approach. Typical examples are:

1. events that are always handled (by default) by the same procedure and are independent of stack history, etc.,
2. events that are frequent and do not want to incur the overhead of the signalling mechanism, and
3. events that are to be signalled normally but that are to be monitored regardless of the value of the "continue" parameter in a call to a handler.

Examples in the first class are:

1. alrm
2. cput
3. mme2
4. a small set handled by default_error_handler_.

Examples in the second class are:

1. linkage faults (when removed from ring 0)
2. lot faults, isot faults (for prelinking)
3. copy-on-write faults

Examples in the third class are more vague but might arise in an attempt to monitor the action of a process.

There have been many attempts to solve these problems using techniques often called "static handlers". The proposals in the past, however, have been very general and attempted to solve all problems associated with such a scheme. The proposal presented here is, on the contrary, quite simple and hence does not provide a general solution. It does, however, provide an efficient, workable solution which, when its limitations are understood and handled, I think, can be useful. The scheme would work as follows:

1. A system condition table (SCT) is allocated (when needed) possibly in a combined linkage segment.

2. An index into this table will be saved as part of the machine conditions. (This index will be the same index used today to select the condition name to be signalled.)

3. The system will record and know about 50-75 events which can be signalled from ring 0. These events are precisely the ones signalled today with a few additions to accommodate new features within the system.

4. The standard signal_ program will call a program to examine the entry in the SCT for the condition being signalled (if machine conditions are provided). If the SCT entry is nonnull, the handler specified therein is invoked. If the SCT entry is null, signal_ proceeds to scan the stack for handlers as today.

5. Entries will be provided to get and set the value of an SCT entry. The value of an SCT entry is a pointer to a handler and may be null. There is no need for the SCT entry to contain an "entry" value because a property of static handlers is that they cannot depend on any automatic storage of a parent block.

6. The calling sequence for all SCT handlers will be simply:

   call handler (mc_ptr, condition_name, continue);

   If the handler sets continue to "0"b, signal_ will not scan the stack. Otherwise, signal_ will scan the stack as is done today.

If a program wants to set up a static handler in this way it should probably first get the previous value of the SCT entry for the condition of interest, and then set up its own value in the SCT. The new handler has the option of calling the previous handler but this, of course, can not be guaranteed to work beacause. of problems such as the old handler being deleted (terminated, etc) unbeknownst to the current handler. Users of this mechanism must have complete knowledge of the execution environment under consideration.

With this brief overview, then, the following subroutines are proposed:

sct_manager_$get_handler (<u>handler</u>, index)

sct_manager_$set_handler (handler, index)

sct_manager_$call_handler (mc_ptr, condition_name, <u>continue</u>)

where:

handler	is of type entry and is the entry to call when the condition occurs.
index	is fixed bin and specifies which static handler is being set or returned.
mc_ptr	is a pointer to the machine conditions for the fault (event) being signalled.
condition_name	is the name of the event being "signalled"
continue	is set to "1"b if the stack should also be searched for a handler and to "0"b if no further processing should be done.

The entry point sct_manager_$call_handler is called by signal_ and is little more than a call forwarder if the SCT entry is nonnull). The program sct_manager_ alone would know the location and format of the SCT.

It is useful to list features and qualities of a condition or event which would cause that event to be unacceptible to be handled by a static handler. The following are such cases:

1. if the necessary handler for the event requires automatic information from an ancestor block in the stack history (overflow fault, for example), or

2. if more than one program wants to know of the event, i.e., if any user program will likely have a handler for the event (such as "quit").

In contrast, handlers that require no previous stack history, require no new storage other than automatic, and have complete and sole interest in an event are possible candidates for static handlers. The following events, currently signalled, are thus likely candidates for static handlers:

alrm
cput
mme2

New events that are good candidates for static handling are:

linkage_fault (when the linker is out of ring 0)
lot_fault
isot_fault
copy-on-write fault (no_write_permission)

These events are indeed static in nature. Were the events handled in ring 0, as linkage faults are today, the handler would be so static that it would be directly called by the fault intercept module. The fact that the handler can now be removed to the user ring does not change the event in such a way that the full condition mechanism is required. The events are still system events and hence the handlers are system programs. There is no need to search the stack for a user handler.

As mentioned earlier, however, the proposed implementation would allow (hopefully knowledgeable) users to provide their own static handlers that might, for example, turn the continue bit ON so that the stack will be searched. The user-ring implementation, thus, provides more freedom -- the defaults have the effect of today.