Published: 9/28/66

Identification

Event-Watchers for Interactive Debugging Aids
D. B. Wagner

Purpose

The command breaker, described in BX.10.03, accepts requests to interrupt program execution upon the occurrence of certain events. These events are such things as a certain amount of real time elapsed or a certain kind of access (execute, read, or write) to a segment or set of locations in a segment. Each of the routines described here handles all of the arrangements for a particular kind of event, and is the only part of the debugger which "knows" how that event is handled. Naturally some of these routines will have uses outside of debugging.

Usage

Every event-watcher is called using the form:

```
     call routine (id, callback, specific arguments relevant
                   to event);
```

The first two arguments are declared

```
     dcl  id  fixed,

          callback entry (fixed,...) bit(1);
```

where the ellipsis indicates declarations peculiar to the routine. The particular watcher stores up, in a personal data base, id, callback, and any other information that may be necessary. It makes arrangements with the system for a trap upon occurrence of the event, and then returns. When this trap occurs the routine regains control of the process and executes a call to callback with the identification number id as the first argument (this is the only use ever made of id) and perhaps other arguments giving precise details of the event.

"Watching" for events of this kind associated with this id is suspended until the return from callback. Callback should return "1"b if such watching is to be resumed, and "0"b if it is to be abandoned.

Particular Watchers

Two watchers are available in the initial implementation: the core-cycle watcher and a very primitive form of "execution-access" watcher (similar to the "break" mechanism in FAPDBG).

To watch core-cycles the call is:

```
        call cycle_watch (id, callback, cycles)
```

with declarations

```
        dcl  id  fixed,

             callback entry (fixed) bit (1),

             cycles fixed binary (63);
```

Cycle-watch watches for the event "cycles more core cycles used by this process".

To watch for control passing to a particular location in a program segment, the call is

```
        call location_watch (id, callback, seg, loc);
```

with declarations

```
        dcl  id  fixed,

             callback entry (fixed) bit (1),

             (seg, loc) bit (18);
```

If necessary location_watch notifies the file-system that the segment is not to be considered pure any more, then plants a special instruction (probably an illegal instruction which will be trapped by the System) into location loc in segment number seg (seg is normally obtained through a call to Segment Management). When this instruction is executed and the trap occurs, the original instruction is reinstated and the call to callback is made. If callback returns "0"b then control merely passes to the (now-reinstated) instruction at the location where the trap occurred. If callback returns "1"b then the instruction at that location is executed interpretively, the special instruction is put back into loc, and control passes to the next instruction. (At this point it does not matter if the next instruction is being trapped too.)

Depending upon how System fault-handling is arranged, it may not be necessary to execute the replaced instruction interpretively: a clever use of the RCU instruction should do the trick.

Naturally location_watch should not be used on any location which the user's program modifies or reads as data, but this is not likely to be a problem in Multics, since pure procedures should be the normal case.