To: Distribution

From: Benson I. Margulies

Date: 01/22/82

Subject: Improvements to the Search Paths

1 ABSTRACT

The search facility has an important limitation: there is no way to validate that the objects put into the lists are acceptable to the programs that use them. This MTB proposes a flexible mechanism for solving this problem.

Comments should be sent to the author:

via Multics Mail:
Margulies.Multics on either MIT Multics or System M.

via US Mail:
Benson I. Margulies
Honeywell Information Systems, inc.
575 Tech Square
Cambridge, Massachusetts 02139

via telephone:
(HVN) 261-9391, or
492-9391

2 INTRODUCTION

Ever since the introduction of the search facility, there have been complaints from the users that the error detection available on the search path manipulation commands is inadequate. There are two major problems.

2.1 Error checking is too little and too late

While there are often strict rules for what sorts of objects may be put in a given list, the add_search_path and set_search_path commands cannot enforce them. Thus users can add directories, or segments whose suffices do not end in ".dict," to the dictionary search path, and not be told that anything is wrong until much later.

2.2 Comparisons are clumsy

To the search path commands, all paths are just character strings. To delete a pathname from a search list the user must give the identical character string that was given when it was added. Short names cannot be used. This is unlike anything else in the system, and is a major limitation. This also prevents the reliable detection of duplicates.

3 PER-LIST VERIFICATION PROCEDURES AND PATH UID'S SOLVE THE PROBLEM

For validation, an easy solution is to allow the system or a user to supply a per-list verification procedure. For comparisons, the definition of an existing pad field in the search list structure as a UID, supplied by the verification procedure, would allow reliable detection.

3.1 Verification Procedures

Verification procedures will be named LISTNAME_sl_. This will impose a limit of 28 characters on the length of names of search paths that have these procedures. The first name defined for the search list in the search segment will be used regardless of the name specified by the user on the command line. The LISTNAME_sl_ program may have the entrypoints:

* LISTNAME_sl_$validate,
* LISTNAME_sl_$compare
* LISTNAME_sl_$duplicates_ok
* LISTNAME_sl_$find
* LISTNAME_sl_$find_ptr

See the MPM pages for the exact calling sequences.

3.1.1 HOW THE PROCEDURES ARE CALLED

For additions to a search path, LISTNAME_sl_$validate will be called on the new path. This program examines the path structure and returns approval or disapproval. If it approves, it may optionally return a UID. Then the check for duplicates is made. If a LISTNAME_sl_$duplicates_ok entrypoint exists, it is called to find out whether duplicate paths are acceptable for this list. This entrypoint may specify that duplicates are to be accepted, rejected, or accepted with a warning. If no LISTNAME_sl_$duplicates_ok can be found, the default is to the user but accept the duplicates. If UID's are available, the duplicate check is made via them. If not, then multiple calls to LISTNAME_sl_$compare are used. If there is no LISTNAME_sl_$compare entrypoint, then string comparisons are used.

For deletions, LISTNAME_sl_$validate is again called for syntax verification. Again, if UID's are available, they are used to search for the path to be deleted. If not, the LISTNAME_sl_$compare entry is again used. If it is not defined, character string comparisons are used.

LISTNAME_sl_$find is used to extend the search_paths_$find_dir and find_all entrypoints. If this entrypoint is defined for a list, then those corresponding entries in search_paths_ will make use of them to find things. For example, a search list of value segments could have a LISTNAME_sl_$find that called value_.

LISTNAME_sl_$find_ptr is a performance enhancement for objects that can be in the address space. Since searching for an object frequently involves initiating the segment that contains it, this saves an initiation when the procedure calling search_paths_$find wants a pointer. For some things, like value segments, the pointer may want to be a pointer to the base of the containing segment rather than to the particular object.

3.2 UID's: their definition and management

For most objects, the standard file system UID will serve as a UID. As of now, there is no entry to the hardcore that returns the UID of a non-initiated segment other than status_long. However, search path changes are not frequent, and need not be especially cheap. A better interface for fetching UID's would be a great improvement.

For search paths that are not file system objects, some other source of UID's is needed. One solution would be to give up, and make use of the compare procedures each time. Another would be to make use of the large number of past-time UID's that will never be assigned to an object. Since the control argument paths do not need UID's at all, the only problem is non-entry objects. Since the ID's only have to be unique within a search path, the use of small numbers (beginning with UID "000000000001"b3) is a reasonable solution.

4 THE INTERFACE TO THE PROCEDURES

To get good error messages out of the verification procedures, they are specified to call sub_err_ to report invalid paths. The search path commands will handle sub_error_, note errors signalled by the verification procedure, and extract the message from the info structure. Since there are no subroutine interfaces for manipulating the paths except modification or replacement of the entire sl_info structure, any programs that wish to modify lists and make use of validation and comparison will have to make_entry and call the procedures themselves.

5 IMPLEMENTATION COST

The modifications to the search list commands, and even verification procedure for all the installed search lists, could be coded in a matter of several working days.

Name: LISTNAME_sl_

LISTNAME_sl_ is a generic name for a procedure associated with a search list that provides validation facilities for paths in the search list. For example, a validation procedure for the dictionary search path would be named "dictionary_sl_". Not all search paths have validation procedures, and not all validation procedures provide all the entrypoints. The documentation for the individual entrypoints specify the correct default action to take if the entrypoint does not exist.

Entry: LISTNAME_sl_$validate

checks a search path for correctness in a given search list. If there is no validate entrypoint for a list, all paths that consist of the standard control arguments, all absolute or relative pathnames, and all such pathnames including active strings are to be considered valid. If the search path is valid, the procedure will set the UID in the path to the correct UID, if any, and return. If the search path is invalid, the procedure will call sub_err_ with a "name" argument of "LISTNAME_sl_$validate," and other arguments sufficient to produce an appropriate error message.

Usage

```
     dcl LISTNAME_sl_$validate entry (character (*), pointer);

          call LISTNAME_sl_$validate (search_list_name, search_path_ptr);
```

where:

search_list_name is the primary name of the search list. (Input)

search_path_ptr is a pointer to a search_path structure, as declared in the include file sl_info.incl.pl1: (Input)

```
               dcl 1 search_path aligned based,
                     2 type fixed binary,
                     2 code fixed bin (35),
                     2 UID bit (36) aligned,
                     2 pathname character (168) unaligned;
```

type may be chosen from the type values defined in sl_info.incl.pl1. (Input)

code will always be zero. (Input)

UID should be set to the UID of the object, or ""b if there is no UID
defined. (Output)

Entry: LISTNAME_sl_$duplicates_ok

returns information about duplicate paths in a search list. If this entrypoint is not defined, then the default is to warn of duplicated.

Usage

```
     dcl LISTNAME_sl_$duplicates_ok entry (char (*)) returns (fixed bin);
          dcl Value fixed bin;

          Value = LISTNAME_sl_$duplicates_ok (search_list_name);
```

where:

search_list_name is the primary name of the search list. (Input)

Value defines the correct treatment of duplicate paths for this list.
(Output) It may be one of:

```
          dcl DUPLICATES_ALLOWED init (1) fixed bin;
          dcl WARN_DUPLICATES init (0) fixed bin;
          dcl PROHIBIT_DUPLICATES init (2) fixed bin;
```

Entry: LISTNAME_sl_$compare

compares two search paths. This entry is only called when one or both of the paths to be compared both has no UID, and is not of a type that can be compared without UID. All of the control argument paths (-working_dir, etc) can be compared without UID. If this entrypoint is not defined, character string comparison is appropriate.

Usage

```
     dcl LISTNAME_sl_$compare entry (char (*), pointer, pointer)
                                          returns (bit (1) aligned);

           Equal = LISTNAME_sl_$compare (search_list_name,
          search_path_ptr_1,
                                                        search_path_ptr_2);

          dcl Equal bit (1) aligned;
```

where:

search_list_name is the primary name of the search list. (Input)

search_path_ptr_1 is a pointer to the first path to be compared. (Input)

search_path_ptr_2 is a pointer to the second path to be compared. (Input)

Equal is "1"b if the paths are equal, and "0"b otherwise.

Entry: LISTNAME_sl_$find

given a search path and an object of interest, indicates whether the object exists in the path.

Usage

```
     dcl LISTNAME_sl_$find entry (pointer, char (*), fixed bin (35))
          returns (bit (1) aligned);
          dcl Found bit (1) aligned;

           Found = LISTNAME_sl_$find (search_path_ptr, search_string,
          code);
```

where:

search_path_ptr is a pointer to a search path structure to be searched for the object. Note that the current referencing path will be available in the pathname portion of the structure for the -referencing_dir path. (Input)

search_string is the object that the caller is looking for. (Input)

code is a standard system status code. It will be nonzero if the search path is invalid, or if the search_string is malformed. Note that the sub_error_ condition may be signalled with an informative string in the information. (Output)

Found will be equal to "1"b if the object was found, and "0"b otherwise.

Entry: LISTNAME_sl_$find_ptr

given a search path an and object of interest, returns a pointer to the object if it exists in the path.

Usage

```
     dcl LISTNAME_sl_$find_ptr entry (pointer, char (*), fixed bin (35))
          returns (pointer);
          dcl Thing_ptr pointer;

          Thing_ptr = LISTNAME_sl_$find_ptr (search_path_ptr,
          search_string, code);
```

where:

search_path_ptr is a pointer to a search path structure to be searched for the object. (Input)

search_string is the name of the object that the caller is looking for. (Input)

code is a standard system status code. It will be nonzero if the search path is invalid, or if the search_string is malformed. Note that the sub_error_ condition may be signalled with an informative string in the information. (Output)

Thing_ptr will be a pointer to the object found, or null. The exact semantics of this pointer are defined on a search list by search list basis.