NAME

nis_tables, nis_list, nis_add_entry, nis_remove_entry, nis_modify_entry, nis_first_entry, nis_next_entry – NIS+ table functions

SYNOPSIS

cc [*flag* . . .] *file*. . . **–lnsl** [*library*. . .]

#include <rpcsvc/nis.h>

nis_result *nis_list(const nis_name *name*, **const u_long** *flags*, **int (*callback)(const nis_name** *table_name*, **const nis_object** **object*, **const void** **userdata*), **const void** **userdata*);

nis_result *nis_add_entry(const nis_name *table_name*, **const nis_object** **object*, **const u_long** *flags*);

nis_result *nis_remove_entry(const nis_name *name*, **const nis_object** **object*, **const u_long** *flags*);

nis_result *nis_modify_entry(const nis_name *name*, **const nis_object** **object*, **const u_long** *flags*);

nis_result *nis_first_entry(const nis_name *table_name*);

nis_result *nis_next_entry(const nis_name *table_name*, **const netobj** **cookie*);

void nis_freeresult(nis_result **result*);

MT-LEVEL

MT-Safe

DESCRIPTION

These functions are used to search and modify NIS+ tables. **nis_list( )** is used to search a table in the NIS+ namespace. **nis_first_entry( )** and **nis_next_entry( )** are used to enumerate a table one entry at a time. **nis_add_entry( )**, **nis_remove_entry( )**, and **nis_modify_entry( )** are used to change the information stored in a table. **nis_freeresult( )** is used to free the memory associated with the **nis_result** structure.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket '[ ]' characters. Indexed names have the following form:

[*colname*=*value*, **. . .],***tablename*

The list function, **nis_list( )**, takes an indexed name as the value for the *name* parameter. Here, the tablename should be a fully qualified NIS+ name unless the **EXPAND_NAME** flag (described below) is set. The second parameter, *flags*, defines how the function will respond to various conditions. The value for this parameter is created by logically **OR**ing together one or more flags from the following list.

FOLLOW_LINKS
If the table specified in *name* resolves to be a **LINK** type object (see **nis_objects**(3N)), this flag specifies that the client library follow that link and do the search at that object. If this flag is not set and the name resolves to a link, the error **NIS_NOTSEARCHABLE** will be returned.

FOLLOW_PATH This flag specifies that if the entry is not found within this table, the list operation should follow the path specified in the table object. When used in conjunction with the **ALL_RESULTS** flag below, it specifies that the path should be followed regardless of the result of the search. When used in conjunction with the **FOLLOW_LINKS** flag above, named tables in the path that resolve to links will be followed until the table they point to is located. If a table in the path is not reachable because no server that serves it is available, the result of the operation will be either a "soft" success or a "soft" failure to indicate that not all tables in the path could be searched. If a name in the path names is either an invalid or non-existent object then it is silently ignored.

HARD_LOOKUP
This flag specifies that the operation should continue trying to contact a server of the named table until a definitive result is returned (such as NIS_NOTFOUND).

ALL_RESULTS This flag can only be used in conjunction with **FOLLOW_PATH** and a callback function. When specified, it forces all of the tables in the path to be searched. If *name* does not specify a search criteria (imply that all entries are to be returned), then this flag will cause all of the entries in all of the tables in the path to be returned.

NO_CACHE This flag specifies that the client library should bypass any client object caches and get its information directly from either the master server or a replica server for the named table.

MASTER_ONLY This flag is even stronger than **NO_CACHE** in that it specifies that the client library should *only* get its information from the master server for a particular table. This guarantees that the information will be up to date. However, there may be severe performance penalties associated with contacting the master server directly on large networks. When used in conjunction with the **HARD_LOOKUP** flag, this will block the list operation until the master server is up and available.

EXPAND_NAME
When specified, the client library will attempt to expand a partially qualified name by calling **nis_getnames( )** (see **nis_local_names**(3N)) which uses the environment variable **NIS_PATH.**

RETURN_RESULT
This flag is used to specify that a copy of the returning object be returned in the **nis_result** structure if the operation was successful.

The third parameter to **nis_list( )**, *callback*, is an optional pointer to a function that will process the **ENTRY** type objects that are returned from the search. If this pointer is **NULL**, then all entries that match the search criteria are returned in the *nis_result* structure, otherwise this function will be called once for each entry returned. When called, this function should return **0** when additional objects are desired and **1** when it no longer wishes to see any more objects. The fourth parameter, *userdata*, is simply passed to callback function along with the returned entry object. The client can use this pointer to pass state information or other relevant data that the callback function might need to process the entries.

nis_add_entry( ) will add the NIS+ object to the NIS+ *table_name*. The *flags* parameter is used to specify the failure semantics for the add operation. The default (*flags* equal 0) is to fail if the entry being added already exists in the table. The **ADD_OVERWRITE** flag may be used to specify that existing object is to be overwritten if it exists, (a modify operation) or added if it does not exist. With the **ADD_OVERWRITE** flag, this function will fail with the error **NIS_PERMISSION** if the existing object does not allow modify privileges to the client.

If the flag **RETURN_RESULT** has been specified, the server will return a copy of the resulting object if the operation was successful.

nis_remove_entry( ) removes the identified entry from the table or a set of entries identified by *table_name*. If the parameter *object* is non-null, it is presumed to point to a cached copy of the entry. When the removal is attempted, and the object that would be removed is not the same as the cached object pointed to by *object* then the operation will fail with an **NIS_NOTSAMEOBJ** error. If an object is passed with this function, the search criteria in name is optional as it can be constructed from the values within the entry. However, if no object is present, the search criteria must be included in the *name* parameter. If the flags variable is null, and the search criteria does not uniquely identify an entry, the **NIS_NOTUNIQUE** error is returned and the operation is aborted. If the flag parameter **REM_MULTIPLE** is passed, and if remove permission is allowed for each of these objects, then all objects that match the search criteria will be removed. Note that a null search criteria and the **REM_MULTIPLE** flag will remove all entries in a table.

nis_modify_entry( ) modifies an object identified by *name*. The parameter *object* should point to an entry with the **EN_MODIFIED** flag set in each column that contains new information.

The owner, group, and access rights of an entry are modified by placing the modified information into the respective fields of the parameter, *object*: **zo_owner**, **zo_group**, and **zo_access**.

These columns will replace their counterparts in the entry that is stored in the table. The entry passed must have the same number of columns, same type, and valid data in the modified columns for this operation to succeed.

If the flags parameter contains the flag **MOD_SAMEOBJ** then the object pointed to by *object* is assumed to be a cached copy of the original object. If the OID of the object passed is different than the OID of the object the server fetches, then the operation fails with the **NIS_NOTSAMEOBJ** error. This can be used to implement a simple read-modify-write protocol which will fail if the object is modified before the client can write the object back.

If the flag **RETURN_RESULT** has been specified, the server will return a copy of the resulting object if the operation was successful.

nis_first_entry( ) fetches entries from a table one at a time. This mode of operation is extremely inefficient and callbacks should be used instead wherever possible. The table containing the entries of interest is identified by *name*. If a search criteria is present in *name* it is ignored. The value of *cookie* within the 34nis_result structure must be copied by the caller into local storage and passed as an argument to **nis_next_entry( )**.

nis_next_entry( ) retrieves the "next" entry from a table specified by *table_name*. The order in which entries are returned is not guaranteed. Further, should an update occur in the table between client calls to **nis_next_entry( )** there is no guarantee that an entry that is added or modified will be seen by the client. Should an entry be removed from the table that would have been the "next" entry returned, the error **NIS_CHAINBROKEN** is returned instead.

RETURN VALUES

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
        nis_error   status;
        struct {
                    u_int         objects_len;
                    nis_object    *objects_val;
        } objects;
        netobj      cookie;
        u_long      zticks;
        u_long      dticks;
        u_long      aticks;
        u_long      cticks;
};
```

The *status* member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )** (see **nis_error**(3N)).

The **objects** structure contains two members. *objects_val* is an array of *nis_object* structures; *objects_len* is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )** (see **nis_names**(3N)). If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )** (see **nis_server**(3N)).

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations (see **nis_db**(3N)).

zticks The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks The time spent in any ''accelerators'' or caches. This includes the time required to locate the server needed to resolve the request.

cticks The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating a NIS+ request.

Subtracting the value in *dticks* from the value in *zticks* will yield the time spent in the service code itself. Subtracting the sum of the values in *zticks* and *aticks* from the value in *cticks* will yield the time spent in the client library itself. Note: all of the tick times are measured in microseconds.

ERRORS

The client library can return a variety of error returns and diagnostics. The more salient ones are documented below.

NIS_BADATTRIBUTE The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED The entry returned came from an object cache that has *expired*. This means that the time to live value has gone to zero and the entry may have changed. If the flag NO_CACHE was passed to the lookup function then the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ The object pointed to by *object* is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL The attempted modification failed for some reason.

NIS_NAMEEXISTS An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table, does not know about the directory in which the table resides.

NIS_NOSUCHTABLE The named table does not exist.

NIS_NOT_ME A request was made to a server that does not serve the given name. Normally this will not occur, however if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND No entries in the table matched the search criteria. If the search criteria was null (return all entries) then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**.

If the FOLLOW_PATH flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i**) to flush this cache.

NIS_NOTSAMEOBJ An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL This result is similar to **NIS_NOTFOUND** except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND The named entry does not exist in the table, however not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS The request was successful.

NIS_SYSTEMERROR Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

ENVIRONMENT

NIS_PATH When set, this variable is the search path used by **nis_list( )** if the flag **EXPAND_NAME** is set.

SEE ALSO

niscat(1), **niserror**(1), **nismatch**(1), **nis_cachemgr**(1M), **nis_error**(3N), **nis_local_names**(3N), **nis_names**(3N), **nis_objects**(3N), **syslog**(3)

WARNINGS

Use the flag **HARD_LOOKUP** carefully since it can cause the application to block indefinitely during a network partition.

NOTES

The path used when the flag **FOLLOW_PATH** is specified, is the one present in the *first* table searched. The path values in tables that are subsequently searched are ignored.

It is legal to call functions that would access the nameservice from within a list callback. However, calling a function that would itself use a callback, or calling **nis_list( )** with a callback from within a list callback function is not currently supported.

There are currently no known methods for **nis_first_entry( )** and **nis_next_entry( )** to get their answers from only the master server.