NAME

mkRegistered, mkLink, reName, rmLink1, rmLink2 – creating and destroying registered Shore objects

SYNOPSIS

```
VASResult       shore_vas::mkRegistered(
    const Path          name,
    mode_t              mode,
    const lrid_t        &typeobj,
    const vec_t         &core,   // initial core value
    const vec_t         &heap,   // initial heap value
    ObjectOffset        tstart, // where TEXT starts
    lrid_t              *result
);

VASResult       shore_vas::mkRegistered(
    const Path          name,
    mode_t              mode,
    const lrid_t        &typeobj,
    ObjectSize          csize,    // size of uninitialized core
    ObjectSize          hsize,    // size of uninitialized heap
    ObjectOffset        tstart, // where TEXT starts
    lrid_t              *result
);

VASResult       shore_vas::mkLink(
    const Path          oldname,
    const Path          newname
);

VASResult       shore_vas::reName(
    const Path          oldname,
    const Path          newname
);

// The following two methods must be used in sequence:
VASResult       shore_vas::rmLink1(
    const Path          name,
    lrid_t              *obj,
    bool          *must_remove
);

VASResult       shore_vas::rmLink2(
    const lrid_t        &obj
);
```

DESCRIPTION

These methods are used to create, destroy, and rename registered objects other than directories.

NB: The SVAS does not check for type integrity. That is the job of the type system. So, for example, the SVAS would not notice if **mkRegistered** were called with

```
typeobj == UnixFile, tstart == NoText,
```

and

```
core.size() > 0.
```

Similarly, an object of an SDL-defined type can be removed by EFSD, and no integrity maintenance is done. **This will be added. (TODO-integrity maintenance in the server).**

The **mkRegistered** methods create objects with initial data (the first form) or with uninitialized data (the second form). The caller is responsible for the integrity of the object's type. The Shore Value-Added server does not interpret the type object and does not verify the legitimacy of the data written.

The method **mkLink** creates a new entry in a directory. The new entry refers to an existing registered object. The object need not be in the same directory as the new entry, but it must be on the same file system (volume) as the new entry's directory. The target of a cannot be a directory (unless the caller is the super-user) or a symbolic link (if a symbolic link is named, it is followed), and the target must be a registered object.

The method **reName** effectively changes the name of an object from *oldname* to *newname.* If *newname* already exists, it is removed. If *newname* is a directory, it must be empty, and its prefix must not include the object named by *oldname.* The caller must have write access to the last directories in *newname* and *oldname.* If *oldname* is a directory, the caller must also have write permission it. If the final component of *oldname* is a symbolic link, the symbolic link is renamed. If *newname* already *exists,* both objects must reside on the same file system, and both objects must be of the same kind (directories or non-directories).

To remove a registered object takes one or two steps, depending on the object's type and its reference (link) count. The first step is to call **rmLink1,** which decrements the object's reference count, and destroys the object if it can. If it cannot, and the caller must destroy the object by calling **rmLink2, rmLink1** returns **TRUE** in ∗*must_remove.* There are three cases:

1 The object's reference count does not reach zero; the object is not destroyed, regardless of its type. **RmLink1** returns **FALSE** in ∗**must_remove.**

2 The object is a file system type (other than a directory -- directories are removed with **rmDir)** and the reference count reaches zero; **rmLink1** destroys **the** object. **RmLink1** returns **FALSE** in ∗**must_remove.**

3 The reference count reaches zero and the object is not a file system type. The caller has to perform some work to maintain type integrity before the object can be destroyed, so **rmLink1** returns TRUE in ∗must_remove.

If the caller attempts to commit the transaction without calling **rmLink2** after **rmLink1** returned **TRUE** in ∗*must_remove,* or the caller tries to call **rmLink2** on an object without first calling **rmLink1,** the Shore Value-Added Server aborts the transaction and returns in error.

ARGUMENTS

Name is the name of the object of interest. It can be a full pathname, a relative pathname, or simply the file name. If it is a file name, the current working directory is assumed to be the directory in which the object is to reside (or resides, in the case of removal). *Mode* is bit mask that specifies the **permissions** for the object. *Typeobj* is the logical object identifier of the object that describes the type of the object to be created. The type object must already exist and be frozen.

In the first form of *mkRegistered*, *core* and *heap* are vectors containing initial values of the core and heap. In the second form, *csize* and *hsize* indicate the sizes of the core and heap, but not their values. *Tstart* is the offset, from the beginning of the object, at which the TEXT field starts. If the object has no TEXT, the value **NoText** should be given.

If the object is created without problems, the resulting object identifier is returned in ∗*result.* (There is no way to assign a pre-allocated object identifier to a registered object.)

Removing objects takes two steps. In the first step, the object is identified by its pathname. The object's identifier is returned in **obj* if obj is non-null when the function **rmLink1** is called. The link count for the object is decremented, and if it reaches zero, the value TRUE is placed in **must_remove.* (*Must_remove* must not be null.) If TRUE is returned, the caller must complete the removal of the object by calling **rmLink2** after performing all the integrity maintenance required by the type system.

ENVIRONMENT

These methods are available to both client and server processes. They can be used only in a transaction.

ERRORS

Deadlocks can occur while locks are being acquired. See **transaction(svas)** for information about deadlocks.

A complete list of errors is in **errors(svas).**

BUGS

Integrity maintenance isn't done on the server.

Hard links to type objects aren't maintained on the server.

VERSION

This manual page applies to Version 1.1 of the Shore software.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

SEE ALSO

errors(svas), transaction(svas), and **text(svas).**