NAME

zfs-program - execute ZFS channel programs

SYNOPSIS

zfs program [**-jn**] [**-t** *instruction-limit*] [**-m** *memory-limit*] *pool script* [*script arguments*]

DESCRIPTION

The ZFS channel program interface allows ZFS administrative operations to be run programmatically as a Lua script. The entire script is executed atomically, with no other administrative operations taking effect concurrently. A library of ZFS calls is made available to channel program scripts. Channel programs may only be run with root privileges.

A modified version of the Lua 5.2 interpreter is used to run channel program scripts. The Lua 5.2 manual can be found at **http://www.lua.org/manual/5.2/**

The channel program given by *script* will be run on *pool*, and any attempts to access or modify other pools will cause an error.

OPTIONS

-j Display channel program output in JSON format. When this flag is specified and standard output is empty - channel program encountered an error. The details of such an error will be printed to standard error in plain text.

-n

Executes a read-only channel program, which runs faster. The program cannot change on-disk state by calling functions from the zfs.sync submodule. The program can be used to gather information such as properties and determining if changes would succeed (zfs.check.*). Without this flag, all pending changes must be synced to disk before a channel program can complete.

-t *instruction-limit*

Limit the number of Lua instructions to execute. If a channel program executes more than the specified number of instructions, it will be stopped and an error will be returned. The default limit is 10 million instructions, and it can be set to a maximum of 100 million instructions.

-m *memory-limit*

Memory limit, in bytes. If a channel program attempts to allocate more memory than the given limit, it will be stopped and an error returned. The default memory limit is 10 MiB, and can be set to a maximum of 100 MiB.

All remaining argument strings will be passed directly to the Lua script as described in the *LUA*

INTERFACE section below.

LUA INTERFACE

A channel program can be invoked either from the command line, or via a library call to **lzc_channel_program**().

Arguments

Arguments passed to the channel program are converted to a Lua table. If invoked from the command line, extra arguments to the Lua script will be accessible as an array stored in the argument table with the key ’argv’:

```
args = ...
argv = args["argv"]
-- argv == {1="arg1", 2="arg2", ...}
```

If invoked from the libzfs interface, an arbitrary argument list can be passed to the channel program, which is accessible via the same "..." syntax in Lua:

```
args = ...
-- args == {"foo"="bar", "baz"={...}, ...}
```

Note that because Lua arrays are 1-indexed, arrays passed to Lua from the libzfs interface will have their indices incremented by 1. That is, the element in *arr[0]* in a C array passed to a channel program will be stored in *arr[1]* when accessed from Lua.

Return Values

Lua return statements take the form:

```
return ret0, ret1, ret2, ...
```

Return statements returning multiple values are permitted internally in a channel program script, but attempting to return more than one value from the top level of the channel program is not permitted and will throw an error. However, tables containing multiple values can still be returned. If invoked from the command line, a return statement:

```
a = {foo="bar", baz=2}
return a
```

Will be output formatted as:

```
Channel program fully executed with return value:
    return:
        baz: 2
        foo: ’bar’
```

Fatal Errors

If the channel program encounters a fatal error while running, a non-zero exit status will be returned. If more information about the error is available, a singleton list will be returned detailing the error:

```
      error: "error string, including Lua stack trace"
```

If a fatal error is returned, the channel program may have not executed at all, may have partially executed, or may have fully executed but failed to pass a return value back to userland.

If the channel program exhausts an instruction or memory limit, a fatal error will be generated and the program will be stopped, leaving the program partially executed. No attempt is made to reverse or undo any operations already performed. Note that because both the instruction count and amount of memory used by a channel program are deterministic when run against the same inputs and filesystem state, as long as a channel program has run successfully once, you can guarantee that it will finish successfully against a similar size system.

If a channel program attempts to return too large a value, the program will fully execute but exit with a nonzero status code and no return value.

Note: ZFS API functions do not generate Fatal Errors when correctly invoked, they return an error code and the channel program continues executing. See the *ZFS API* section below for function-specific details on error return codes.

Lua to C Value Conversion

When invoking a channel program via the libzfs interface, it is necessary to translate arguments and return values from Lua values to their C equivalents, and vice-versa.

There is a correspondence between nvlist values in C and Lua tables. A Lua table which is returned from the channel program will be recursively converted to an nvlist, with table values converted to their natural equivalents:

string	->string
number	->int64
boolean	->boolean_value
nil	->boolean (no value)
table	->nvlist

Likewise, table keys are replaced by string equivalents as follows:

string	->no change
number	->signed decimal string ("%lld")
boolean	->"true" |

"false"

Any collision of table key strings (for example, the string "true" and a true boolean value) will cause a fatal error.

Lua numbers are represented internally as signed 64-bit integers.

LUA STANDARD LIBRARY

The following Lua built-in base library functions are available:

assert	rawlen	collectgarbage	rawget
error	rawset	getmetatable	select
ipairs	setmetatable	next	tonumber
pairs	tostring	rawequal	type

All functions in the *coroutine*, *string*, and *table* built-in submodules are also available. A complete list and documentation of these modules is available in the Lua manual.

The following functions base library functions have been disabled and are not available for use in channel programs:

dofileloadfileloadpcallprintxpcall

ZFS API

Function Arguments

Each API function takes a fixed set of required positional arguments and optional keyword arguments. For example, the destroy function takes a single positional string argument (the name of the dataset to destroy) and an optional "defer" keyword boolean argument. When using parentheses to specify the arguments to a Lua function, only positional arguments can be used:

zfs.sync.destroy("rpool@snap")

To use keyword arguments, functions must be called with a single argument that is a Lua table containing entries mapping integers to positional arguments and strings to keyword arguments:

zfs.sync.destroy({1="rpool@snap", defer=true})

The Lua language allows curly braces to be used in place of parenthesis as syntactic sugar for this calling convention:

zfs.sync.snapshot{"rpool@snap", defer=true}

Function Return Values

If an API function succeeds, it returns 0. If it fails, it returns an error code and the channel program continues executing. API functions do not generate Fatal Errors except in the case of an unrecoverable internal file system error.

In addition to returning an error code, some functions also return extra details describing what caused the error. This extra description is given as a second return value, and will always be a Lua table, or Nil if no error details were returned. Different keys will exist in the error details table depending on the function and error case. Any such function may be called expecting a single return value:

```
errno = zfs.sync.promote(dataset)
```

Or, the error details can be retrieved:

```
errno, details = zfs.sync.promote(dataset)
if (errno == EEXIST) then
    assert(details ~= Nil)
    list_of_conflicting_snapshots = details
end
```

The following global aliases for API function error return codes are defined for use in channel programs:

```
EPERM     ECHILD  ENODEV   ENOSPC  ENOENT  EAGAIN  ENOTDIR
ESPIPE    ESRCH   ENOMEM   EISDIR  EROFS   EINTR   EACCES
EINVAL    EMLINK  EIO      EFAULT  ENFILE  EPIPE   ENXIO
ENOTBLK   EMFILE  EDOM     E2BIG   EBUSY   ENOTTY  ERANGE
ENOEXEC   EEXIST  ETXTBSY  EDQUOT  EBADF   EXDEV   EFBIG
```

API Functions

For detailed descriptions of the exact behavior of any ZFS administrative operations, see the main zfs(8) manual page.

zfs.debug(*msg*)

Record a debug message in the zfs_dbgmsg log. A log of these messages can be printed via mdb's "::zfs_dbgmsg" command, or can be monitored live by running

```
dtrace -n 'zfs-dbgmsg{trace(stringof(arg0))}'
```

msg (string) Debug message to be printed.

zfs.exists(*dataset*)

Returns true if the given dataset exists, or false if it doesn't. A fatal error will be thrown if the dataset is not in the target pool. That is, in a channel program running on rpool, **zfs.exists**("rpool/nonexistent_fs") returns false, but **zfs.exists**("somepool/fs_that_may_exist") will error.

dataset (string) Dataset to check for existence. Must be in the target pool.

zfs.get_prop(*dataset, property*)

Returns two values. First, a string, number or table containing the property value for the given dataset. Second, a string containing the source of the property (i.e. the name of the dataset in which it was set or nil if it is readonly). Throws a Lua error if the dataset is invalid or the property doesn’t exist. Note that Lua only supports int64 number types whereas ZFS number properties are uint64. This means very large values (like GUIDs) may wrap around and appear negative.

dataset (string)	Filesystem or snapshot path to retrieve properties from.
property (string)	Name of property to retrieve. All filesystem, snapshot and volume properties are supported except for **mounted** and **iscsioptions**. Also supports the **written@***snap* and **written#***bookmark* properties and the <**user**|**group**><**quota**|**used**>**@***id* properties, though the id must be in numeric form.

zfs.sync submodule

The sync submodule contains functions that modify the on-disk state. They are executed in "syncing context".

The available sync submodule functions are as follows:

zfs.sync.destroy(*dataset*, [*defer*=**true**|**false**])

Destroy the given dataset. Returns 0 on successful destroy, or a nonzero error code if the dataset could not be destroyed (for example, if the dataset has any active children or clones).

dataset (string)	Filesystem or snapshot to be destroyed.
[*defer* (boolean)]	Valid only for destroying snapshots. If set to true, and the snapshot has holds or clones, allows the snapshot to be marked for deferred deletion rather than failing.

zfs.sync.inherit(*dataset, property*)

Clears the specified property in the given dataset, causing it to be inherited from an ancestor, or restored to the default if no ancestor property is set. The **zfs inherit -S** option has not been implemented. Returns 0 on success, or a nonzero error code if the property could not be cleared.

dataset (string)	Filesystem or snapshot containing the property to clear.
property (string)	The property to clear. Allowed properties are the same as those for the **zfs inherit** command.

zfs.sync.promote(*dataset*)

Promote the given clone to a filesystem. Returns 0 on successful promotion, or a nonzero error code otherwise. If EEXIST is returned, the second return value will be an array of the clone’s snapshots whose names collide with snapshots of the parent filesystem.

dataset (string)	Clone to be promoted.

zfs.sync.rollback(*filesystem*)

Rollback to the previous snapshot for a dataset. Returns 0 on successful rollback, or a nonzero error code otherwise. Rollbacks can be performed on filesystems or zvols, but not on snapshots or mounted datasets. EBUSY is returned in the case where the filesystem is mounted.

filesystem (string)	Filesystem to rollback.

zfs.sync.set_prop(*dataset*, *property*, *value*)

Sets the given property on a dataset. Currently only user properties are supported. Returns 0 if the property was set, or a nonzero error code otherwise.

dataset (string)	The dataset where the property will be set.
property (string)	The property to set.
value (string)	The value of the property to be set.

zfs.sync.snapshot(*dataset*)

Create a snapshot of a filesystem. Returns 0 if the snapshot was successfully created, and a nonzero error code otherwise.

Note: Taking a snapshot will fail on any pool older than legacy version 27. To enable taking snapshots from ZCP scripts, the pool must be upgraded.

dataset (string)	Name of snapshot to create.

zfs.sync.rename_snapshot(*dataset*, *oldsnapname*, *newsnapname*)

Rename a snapshot of a filesystem or a volume. Returns 0 if the snapshot was successfully renamed, and a nonzero error code otherwise.

dataset (string)	Name of the snapshot's parent dataset.
oldsnapname (string)	Original name of the snapshot.
newsnapname (string)	New name of the snapshot.

zfs.sync.bookmark(*source*, *newbookmark*)

Create a bookmark of an existing source snapshot or bookmark. Returns 0 if the new bookmark was successfully created, and a nonzero error code otherwise.

Note: Bookmarking requires the corresponding pool feature to be enabled.

source (string)	Full name of the existing snapshot or bookmark.
newbookmark (string)	Full name of the new bookmark.

zfs.check submodule

For each function in the **zfs.sync** submodule, there is a corresponding **zfs.check** function which performs a "dry run" of the same operation. Each takes the same arguments as its **zfs.sync** counterpart and returns 0 if the operation would succeed, or a non-zero error code if it would fail, along with any other error details. That is, each has the same behavior as the corresponding sync function except for actually executing the requested change. For example, **zfs.check.destroy**(*"fs"*) returns 0 if **zfs.sync.destroy**(*"fs"*) would successfully destroy the dataset.

The available **zfs.check** functions are:
zfs.check.destroy(*dataset*, [*defer*=**true**|**false**])
zfs.check.promote(*dataset*)
zfs.check.rollback(*filesystem*)
zfs.check.set_property(*dataset*, *property*, *value*)
zfs.check.snapshot(*dataset*)

zfs.list submodule

The zfs.list submodule provides functions for iterating over datasets and properties. Rather than returning tables, these functions act as Lua iterators, and are generally used as follows:

```
for child in zfs.list.children("rpool") do
  ...
end
```

The available **zfs.list** functions are:

zfs.list.clones(*snapshot*)

Iterate through all clones of the given snapshot.

snapshot (string)	Must be a valid snapshot path in the current pool.

zfs.list.snapshots(*dataset*)

Iterate through all snapshots of the given dataset. Each snapshot is returned as a string containing the full dataset name, e.g. "pool/fs@snap".

dataset (string)	Must be a valid filesystem or volume.

zfs.list.children(*dataset*)

Iterate through all direct children of the given dataset. Each child is returned as a string containing the full dataset name, e.g. "pool/fs/child".

dataset (string) Must be a valid filesystem or volume.

zfs.list.bookmarks(*dataset*)

Iterate through all bookmarks of the given dataset. Each bookmark is returned as a string containing the full dataset name, e.g. "pool/fs#bookmark".

dataset (string) Must be a valid filesystem or volume.

zfs.list.holds(*snapshot*)

Iterate through all user holds on the given snapshot. Each hold is returned as a pair of the hold’s tag and the timestamp (in seconds since the epoch) at which it was created.

snapshot (string) Must be a valid snapshot.

zfs.list.properties(*dataset*)

An alias for zfs.list.user_properties (see relevant entry).

dataset (string) Must be a valid filesystem, snapshot, or volume.

zfs.list.user_properties(*dataset*)

Iterate through all user properties for the given dataset. For each step of the iteration, output the property name, its value, and its source. Throws a Lua error if the dataset is invalid.

dataset (string) Must be a valid filesystem, snapshot, or volume.

zfs.list.system_properties(*dataset*)

Returns an array of strings, the names of the valid system (non-user defined) properties for the given dataset. Throws a Lua error if the dataset is invalid.

dataset (string) Must be a valid filesystem, snapshot or volume.

EXAMPLES

Example 1

The following channel program recursively destroys a filesystem and all its snapshots and children in a naive manner. Note that this does not involve any error handling or reporting.

```
function destroy_recursive(root)
    for child in zfs.list.children(root) do
        destroy_recursive(child)
    end
    for snap in zfs.list.snapshots(root) do
        zfs.sync.destroy(snap)
    end
    zfs.sync.destroy(root)
end
destroy_recursive("pool/somefs")
```

Example 2

A more verbose and robust version of the same channel program, which properly detects and reports errors, and also takes the dataset to destroy as a command line argument, would be as follows:

```
succeeded = {}
failed = {}

function destroy_recursive(root)
    for child in zfs.list.children(root) do
        destroy_recursive(child)
    end
    for snap in zfs.list.snapshots(root) do
        err = zfs.sync.destroy(snap)
        if (err ~= 0) then
            failed[snap] = err
        else
            succeeded[snap] = err
        end
    end
    err = zfs.sync.destroy(root)
    if (err ~= 0) then
        failed[root] = err
    else
        succeeded[root] = err
    end
end

args = ...
argv = args["argv"]
```

```
destroy_recursive(argv[1])

results = {}
results["succeeded"] = succeeded
results["failed"] = failed
return results
```

Example 3

The following function performs a forced promote operation by attempting to promote the given clone and destroying any conflicting snapshots.

```
function force_promote(ds)
  errno, details = zfs.check.promote(ds)
  if (errno == EEXIST) then
      assert(details ~= Nil)
      for i, snap in ipairs(details) do
          zfs.sync.destroy(ds .. "@" .. snap)
      end
  elseif (errno ~= 0) then
      return errno
  end
  return zfs.sync.promote(ds)
end
```