lua_iptable reference

git.pdh

lua_iptable.h

Defines

LUA_IPTABLE_ID

Identity for the table_t-userdata.

LUA_IPT_ITR_GC

Identity for the itr_gc_t-userdata.

ipt ERROR numbers

lua_iptable.c

Lua bindings for iptable

Structures

itr_gc_t

The itr_gc_t type has 1 member: table_t *t and serves only to create a userdata to be supplied as an upvalue to each iterator function. That userdata has a metatable with a __gc garbage collector, (see ipt_itr_gc) which will remove radix nodes in t that are (still) flagged for deletion only when it is safe to do so (i.e. t->itr_lock has reached zero).

static const char *_errstr(int errno) { / *if (errno < 0 || errno > LIPTE_ZMAX) return NULL;* / return lipt_errstr[errno]; / *}* /

lualib

`luaopen_iptable`

```
  int luaopen_iptable(lua_State *L);
```

Called by Lua upon `require("iptable")` to initialize the module. Returns `iptable`'s module-table with the module functions and constants. Instance methods go into its metatable.

error handlers

`lipt_err`

```
static int lipt_err(int errno, table_t *t);
```

Safely sets either the library global `lipt_errno` or `t->errno` if t is non-NULL. Used by functions that fail silently by returning 0 to Lua while setting the table or library-global error number.

`lipt_errfail`

```
static void
lipt_errfail(lua_State *L, int errno, const char *fmt, const char *s);
```

Raises an error to Lua using the error description for errno. If fmt is not NULL, adds a space and the formatted string (limited to 1024 characters) to the end of the error message.

static void / lipt_errfail(lua_State L, int errno, const char fmt, ...) / { / #define MAX_ERRBUF 1024 / dbg_stack("inc(.) <–"); / char buf[MAX_ERRBUF]; / va_list args; / lipt_err(errno, NULL); / lua_pushstring(L, lipt_errstr[lipt_errno]); / if (fmt) { / va_start(args, fmt); / buf[0] = ' '; / vsnprintf(buf+1, MAX_ERRBUF-1, fmt, args); / va_end(args); / lua_pushstring(L, buf); / lua_concat(L, 2); / } / luaL_error(L, lua_tostring(L, -1)); / } */

stack functions

- `iptL`-functions get/set/push stack values.
- `iptT`-functions manipulate k,v-pairs of a table on top of L

iptL_gettable

```
static table_t *iptL_gettable(lua_State *L, int index);
```

Checks whether the `L` stack value at `index` contains a userdata of type `LUA_IPTABLE_ID` and returns a `table_t` pointer. Errors out to Lua if the stack value has the wrong type.

iptL_getaf

```
static int iptL_getaf(lua_State *L, int idx, int *af);
```

Checks whether `L[idx]` is a known af_family number or name and sets `*af` accordingly or to `AF_UNSPEC` if an unknown af_family was found. If `L[idx]` is missing, `*af` is not modified. Caller decides if a missing or unknown af_family is an error or not.

iptL_getbinkey

```
static int iptL_getbinkey(lua_State *L, int idx, uint8_t *buf, size_t
*len);
```

Copy a binary key, either an address or a mask, at given 'idx' into given 'buf'. 'buf' size is assumed to be MAX_BINKEY. A binary key is a byte array [LEN | key bytes] and LEN is total length. However, radix masks may have set their LEN-byte equal to the nr of non-zero bytes in the array instead.

iptL_getpfxstr

```
const char * iptL_getpfxstr(lua_State *L, int idx, size_t *len)
```

Checks if `L[idx]` is a string and yields a const char ptr to it or NULL. Sets len to the string length reported by Lua's stack manager. Lua owns the memory pointed to by the return value (no free by caller needed).

iptL_refpcreate

```
static int iptL_refpcreate(lua_State *L);
```

Create a reference id for the lua_State given. This allocates an int pointer, gets a new ref_id, stores it in the allocated space and returns a pointer to it.

iptL_refpdelete

```
static void iptL_refpdelete(void *L, void **r);
```

Delete a value from LUA_REGISTRYINDEX indexed by **r (which is treated as an** int). This function acts as the purge_f_t (see iptable.h) for the table.

iter_fail_f

```
static int iter_fail_f(lua_State *L);
```

An iteration func that terminates any iteration.

module functions

iptable.new

```
 static int ipt_new(lua_State *L);
```

Creates a new userdata, sets its `iptable` metatable and returns it to Lua. It also sets the purge function for the table to `iptL_refpdelete` which frees any memory held by the user's data once a prefix is deleted from the radix tree.

iptable.tobin

Return byte array for binary key, masklength & AF for a given prefix, nil on errors. Note: byte array is [LEN-byte | key-bytes].

iptable.tostr

Return string representation for a binary key, nil on errors

iptable.tolen

Return the number of consecutive msb 1-bits, nil on errors.

- stops at the first zero bit.

iptable.size

Returns the number of hosts in a given prefix, nil on errors

- uses Lua's arithmatic (2^hostbits), since ipv6 can get large.

`iptable.address`

Return host address, masklen and af_family for pfx; nil on errors.

```
addr, mlen, af = iptable.address("10.10.10.10/24")
```

`ipt_error`

Returns nil if there is no error, otherwise returns the description and number of the last error seen; also clears the error.

`iptable.network`

Return network address, masklen and af_family for pfx; nil on errors.

`iptable.neighbor`

Given a prefix, return the neighbor prefix, masklen and af_family, such that both can be combined into a supernet with masklen -1; nil on errors.

`iptable.broadcast`

Return broadcast address, masklen and af_family for pfx; nil on errors.

`iptable.mask`

```
mask, af, mlen = iptable.mask(af, mlen [,true])
```

Returns the mask, as a string, for the given address family `af` and mask length `mlen` Inverts the mask if the optional 3rd argument evaluates to true.

TODO:

- ☐ add invert parm instead of using -mlen to convey desire for inversion
- ☒ restore mlen<0 to mean IPx_MAXMASK

`iptable.hosts`

Iterate across host adresses in prefix pfx.

The optional incl argument defaults to false, when given & true, the network and broadcast address will be included in the iteration results. If pfx has no mask, the af's max mask is used.

iter_hosts_f

The iter_f for iptable.hosts(pfx), yields the next host ip until stop value is reached. Ignores the stack: it uses upvalues for next, stop

instance methods

iptm_gc

Garbage collect the table: free all resources Assigned to metable's `__gc` method.

iptm_newindex

Implements `t[k] = v`

If v is nil, t[k] is deleted. Otherwise, v is stored in the lua_registry and its ref-value is stored in the radix tree. If k is not a valid ipv4/6 prefix, cleanup and ignore the request.

iptm_index

Given an index `k`:

- do longest prefix search if `k` is a prefix without mask,
- do an exact match if `k` is a prefix with a mask
- otherwise, do metatable lookup for property named by `k`.

iptm_len

Return the sum of the number of entries in the ipv4 and ipv6 radix trees as the 'length' of the table.

iptm_tostring

Return a string representation of the iptable instance (with iptm_counts).

iptm_counts

Return both ipv4 and ipv6 iptm_counts as two numbers.

`iptm_error`

Returns nil if there is no error, otherwise the error number and its description, in which case the error number is also cleared.

`iter_kv`

Setup iteration across key,value-pairs in the table. The first ipv4 or ipv6 leaf node is pushed. The iter_pairs iterator uses nextleaf to go through all leafs of the tree. It will traverse ipv6 as well if we started with the ipv4 tree first.

`iter_kv_f`

The iter_f for __iter_kv(), yields all k,v pairs in the tree(s).

- upvalue(1) is the leaf to process.

`t:more`

Iterate across more specific prefixes. AF_family is deduced from the prefix given. Note this traverses the entire tree if pfx has a /0 mask.

`iter_more`

Iterate across more specific prefixes one at a time.

- top points to subtree where possible matches are located
- rn is the current leaf under consideration

`t:less`

Iterate across prefixes in the tree that are less specific than pfx.

- search may include pfx itself in the results, if incl is true.
- `iter_less_f` takes (pfx, mlen, af)

`iter_less_f`

Iterate across less specific prefixes relative to a given prefix.

`t:masks`

Iterate across all masks used in AF_family's radix tree. Notes:

- a mask tree stores masks in rn_key fields.
- a mask's KEYLEN == nr of non-zero mask bytes, rather LEN of byte array.
- a /0 (zeromask) is never stored in the radix mask tree (!).
- a /0 (zeromask) is stored in ROOT(0.0.0.0)'s *last* dupedkey-chain leaf.
- the AF_family cannot be deduced from the mask in rn_key.

So, the iter_f needs the AF to properly format masks for the given family (an ipv6/24 mask would otherwise come out as 255.255.255.0) and an explicit flag needs to be passed to iter_f that a zeromask entry is present.

`iter_masks_f`

iter_f for `t:masks(af)`, yields all masks used in af's radix mask tree. Masks are read-only: the Lua bindings donot (need to) interact directly with a radix mask tree.

`t:merge`

Iterate across groups of pfx's that may be combined, to setup just return the first leaf of the correct AF_family's tree.

`iter_merge`

Iterate across prefixes that may be combined.

Returns the supernet and a regular table with 2 or 3 pfx,value entries.

Caller may decide to keep only the supernet or parts of the group. If the supernet exists in the tree it will be present in the table as well as a third entry.

The next node stored as upvalue before returning, must be the first node of the next group.

`iptL_pushitrgc`

During iteration(s), delete operations only flag radix nodes (leafs holding a prefix) for deletion so they are not actively removed, only flagged for deletion. This way, the 'next leaf' node, stored as an upvalue by the iter_xxx functions, is guaranteed to still exist on the next iteration.

Such ‘inactive’ leafs are still part of the tree, but are seen as absent by regular tree operations.

The iterator garbage collector only actually removes those flagged for deletion when there are no more iterators active (ie. itr_count reaches zero).

`ipt_itr_gc`

The garbage collector for iterator functions, pushed by the factory function and executed when Lua decides to do some garbage collection.

`t:radix`

Iterate across all nodes of all types in af’s radix trees.

Only includes the radix_mask tree if the optional maskp argument is true; The iterator will return the C-structs as Lua tables.

`t:iter_radix`

- return current node as a Lua table or nil to stop iteration
- save next (type, node) as upvalues (1) resp. (2)

`iptT_setfstr`

TODO: doc and move

`iptT_setint`

TODO: doc and move

`iptT_setkv`

TODO: move set (rn->rn_key, v) on Table on top of L, based on pfx provided and its value stored (indirectly) in the corresponding radix tree.

`iptL_pushrnh`

push_rdx_’s each translating a struct to a lua table. Will translate and push IPTF_DELETE’d nodes for completeness sake. The lua caller may decide herself what to do with those.

iptL_pushrn

TODO: [ ] doc [ ] move

iptL_pushrh

TODO: [ ] doc [ ] move

push a radix head onto L (as a table)

iptL_pushrmh

TODO: [ ] doc [ ] move push a radix mask head onto L (as a table)

iptL_pushrm

TODO: [ ] doc [ ] move