Abstract

The 4.4BSD operating system introduced the `sysctl` system call to get or set the state of the system, the kernel exposes the available parameters for `sysctl` as objects of a Management Information Base. Nowadays FreeBSD has thousands of `sysctl` parameters, moreover, they can also be added or deleted dynamically, so the kernel has to provide additional features for exploring the MIB, converting the name of a parameter in its corresponding MIB identifier and getting the info of an object (e.g., name, description, type, etc.). Currently the kernel provides an undocumented interface to fulfill these tasks, it was introduced over twenty years ago, this paper presents a new interface providing new features and improving the efficiency to access to the MIB.

1 Introduction

The FreeBSD [1] kernel maintains a Management Information Base ("MIB") where a component ("object") represents a parameter of the system. The MIB provides a convenient hierarchical notation to describe the kernel namespace [2], each object has a number so an Object Identifier ("OID") is a series of integers separated by periods. The `sysctl` system call [3] explores the MIB to find an object by its OID then it can retrieve or set the value of the corresponding parameter.

The MIB is implemented by a collection of trees, the root nodes are the top level objects and are stored as entries of a SLIST [4], each node represents an object and is defined by a `struct sysctl_oid` [Listing 1]; the complete MIB data structure is known as `sysctl MIB-Tree` or `sysctl tree`.

Listing 1: `sysctl tree` node

```c
struct sysctl_oid {
    struct sysctl_oid_list oid_children;
    struct sysctl_oid_list *oid_parent;
    SLIST_ENTRY(sysctl_oid) oid_link;
    int oid_number;
    u_int oid_kind;
    void *oid_arg1;
    intmax_t oid_arg2;
    const char *oid_name;
    int (*oid_handler)(SYSCTL_HANDLER_ARGS);
    const char *oid_fmt;
    int oid_refcnt;
    u_int oid_running;
    const char *oid_descr;
    const char *oid_label;
};
```

The `sysctl` syscall [Listing 2] represents an OID by an array of integers and an unsigned integer, when the node with the specified OID is found its handler is called: it can pass the values between the kernel and userspace via two buffers.

Listing 2: `sysctl()` system call

```c
int sysctl(const int *id, u_int idlevel,
    void *oldp, size_t *oldlenp,
    const void *newp, size_t newlen);
```

It is often necessary finding an object not for its value (calling the handler) but to retrieve its information (e.g., name, description, type, next node, etc.), so the kernel provides an undocumented interface, `sysctlinfo` is a new interface to visit the `sysctl tree` and to pass the info of a node to userland.
The rest of the paper is organized as follows: Section 2 gives a description of the current interface and its limitations, Section 3 introduces sysctlinfo and explains its design and implementation, real world use cases are shown in the successive section. The work is concluded with some consideration and future directions.

2 The current interface

Currently the sysctl MIB consists of thousands of objects, they have various info: types, formats, flags, etc., furthermore thesysctl(9) interface [6] allows to add or delete an object dinamically. The sysctl syscall finds an object by its OID then can get or set its value, only this functionality is not sufficient, for example the sysctl(8) utility [5] needs to explore the MIB, convert the name of an object in its corresponding OID and finally to get the info of an object to display properly its value [Listing 3].

Listing 3: sysctl(8)

% sysctl kern.ostype
kern.ostype: FreeBSD
% sysctl -t kern.ostype
kern.ostype: string
% sysctl -a
kern.ostype
...
compat.ia32.maxdsiz

During the years new members were added to struct sysctlOid [Listing 1]: oid_descr and oid_label, they allow to know the description of an object and to address the modern cloud computing requirements [11], [Lising 4].

Listing 4: object description and label

% sysctl -d kern.ostype
kern.ostype: Operating system type
%prometheus_sysctl_exporter kern.
features_compat.freebsd7
sysctl.kern.features{feature="compat.
freebsd7"} 1

The FreeBSD kernel provides an undocumented interface, introduced over twenty years ago [8], to retrieve the info of an object: name, type, format, description, next leaf and OID by name, later: description [9] and label [10]. The interface is implemented in kern_sysctl.c by a set of internal nodes: sysctl.name, sysctl.next, sysctl.oidfmt, sysctl.oiddescr, sysctl.oidlabel and sysctlname2oid. The internal nodes, except sysctl.name2oid, are CTLTYPE_NODEs with a not-NULL handler, so the desired node is specified extending the OID of the internal node, [Listing 5] shows how getting the description of a node via sysctl.oiddesc.

Listing 5: current interface API -1-

ioid [0] = CTL_SYSCTL;
oid [1] = CTL_SYSCTL_DESC;
mempcpy(ioid +2, oid, oidlen + sizeof(int));
sysctl(ioid, oidlen +2, buf, &bufsize, 0, 0);

The sysctl.name2oid internal node uses the newp and oldp buffers [Listing 6].

Listing 6: current interface API -2-
ioid [0] = CTL_SYSCTL;
oid [1] = CTL_SYSCTL_NAME2OID;
sysctl(ioid, 2, oid, oidlen ,
name, strlen(name) +1);

Limitations of the current interface

The CTL_MAXNAME constant, in sys/sysctl.h, defines the max level of an OID, actually it is 24, so sysctl(9) can add a node of 24 levels:

x1.x2.x3.x4.x5.x6.x7.x8.x9.x10.x11.
x12.x13.x14.x15.x16.x17.x18.x19.x20.
x21.x22.x23.x24

and the sysctl() syscall can get or set its value. Unfortunately, the current interface can manage an object up to CTL_MAXNAME-2 levels because the internal nodes, except sysctl.name2oid, use 2 levels for their OID (see sysctl() of [Listing 5]), consequently an utility like sysctl(8) fails with an object of 23 or 24 levels [Listing 7].

Listing 7: sysctl(8) false negative

% /sbin/sysctl x1
sysctl: sysctl(getnext) -1 88: Cannot allocate memory
% /sbin/sysctl x1.x2.x3.x4.x5.x6.x7.x8.
x9.x10.x11.x12.x13.x14.x15.x16.x17.x18.
x19.x20.x21.x22.x23.x24
sysctl: sysctl fmt -1 1024 22: Invalid argument
The current interface provides `sysctl.next` to explore the MIB, it finds the specified object and gets the next leaf. However a MIB explorer [Figure 1] needs to retrieve also the next internal node. The early versions of `sysctlview` [18], a graphical `sysctl MIB` explorer, wasted computation in userspace comparing the OIDs of two consecutive leaves to retrieve the internal nodes.

Figure 1: sysctlview

The `sysctl.name` node finds an object by its OID and gets its name, example: [1.1] → "kern.ostype". However if no object has the specified OID the internal node builds a "fake" name depending on the input OID and returns always '0' false positive [Listing 5], example: [1.1.100.500.1000] → "kern.ostype.100.500.1000" or a totally non-existent OID [3000.4000.5000] → "3000.4000.5000". This behavior is described as a bug by the `sysctlmibinfo` library [14], it could be useful to have an internal node returns error if no node exists with the specified OID.

The `sysctl.name2oid` convert a name of an object in its OID, it is used internally by `sysctlbyname()` [3]. Unfortunately this internal node can not manage an extended name for the handler of a `CTLTYPE_NODE` with a not-NULL, so unlike `sysctl(3), sysctlbyname()` can not get or set the value of an object like "kern.proc.pid.1".

Furthermore `sysctl.name2oid` finishes to build the OID if a level-name is just the "NULL string", so `sysctlbyname()` could get or set the value of an unwanted object. Consider [Listing 8], the `sysctl(8)` utility uses `sysctl.name2oid` to retrieve the OID of "security.jail.param.allow.mount.", so it receives an incomplete OID, in fact it shows the requested node and its brothers.

Listing 8: sysctl(8) shows unwanted objects
%
security.jail.param.allow.mount.
security.jail.param.allow.mount.tmpfs: 0
security.jail.param.allow.mount.debugfs: 0
security.jail.param.allow.mount.anon_inodefs: 0
security.jail.param.allow.mount.proofs: 0
security.jail.param.allow.mount.devfs: 0
security.jail.param.allow.mount.: 0

Finally, the current interface does not take care of security: in capability mode [13] it exposes the info of a nodes without the `CTLFLAG_CAPRD` or `CTLFLAG_CAPWR` flag.

3 A new interface

This paper presents a new interface: `sysctlinfo` [16], its purpose is to address the limitations of the current interface, to improve the efficiency and to implement new features; moreover the project provides: a README, a manual, helper macros, examples, and converted tools. Obviously the interfaces can coexist, the utilities and libraries can continue to use the current kernel interface while the converted tools can take the advantages by using `sysctlinfo`.

Features

Primarily `sysctlinfo` provides a new set of internal nodes correspondig to the current interface, [Table 1] for a comparison, the new nodes: `sysctl.entryfakename`, `sysctl.entrydesc`, `sysctl.entrylabel`, `sysctl.entrykind`, `sysctl.entryfmt`, `sysctl.entrynextleaf` and `sysctl.entryfakeidbyname` can manage an object up to `CTL_MAXNAME` levels; [Listing 9] displays the output of the `sysctl(8)` utility converted to use `sysctlinfo`, compare with [Listing 7].
Moreover new features were implemented. The support for the capability mode (the info of a node without CTLFLAG_CAPRD or CTLFLAG_CAPWR are not passed to the userland after a cap_enter() call [13]). Unlike sysctl.entryfakename or sysctl.name, sysctl.entryname does not build a fake name and returns an error if no object has the specified OID. sysctl.entrynextnode avoids useless computation in userspace by getting the next leaf or next internal node. sysctl.entrybyname builds a correct OID also if some level-name is just the "NULL string", compare [Listing 10] with [Listing 8].

The new interface is still inefficient: it can pass to the userland only a single info at a time, then the kernel needs to find the same objects many times, so new nodes were implemented: sysctl.entryallinfo, sysctl.entryallinfo withnextnode and sysctl.entryallinfo withnextleaf, they are 30% more efficient to get all info of a node [Figure 2].

Finally, *byname nodes were added: sysctl.entryidinputbyname, sysctl.entrydescbyname, sysctl.entrylabelbyname, sysctl.entrykindbyname, sysctl.entryfmtbyname, sysctl.entryallinfobyname withnextnode and sysctl.entryallinfobyname withnextleaf, they search the object by its name avoiding to call sysctl.name2oid (or similar) to explore the MIB just to find the corresponding OID.

Note, sysctl.entryidinputbyname [17] can manage an extended name with the input for the handler of the object, example: "kern.proc.pid.1", then it allows to sysctblbyname() to get or set the value a CTLTYPE_NODE with a not-NULL handler.

**API**

The sysctlinfo interface provides a new API, it defines two main macros [Listing 11], so the request for info instead of value is obvious, compare with [Listing 5] and [Listing 6].

```
Listing 11: sysctlinfo API

int SYSCTLINFO(int *id, size_t idlevel, int prop[2], void *buf, size_t *buflen);
int SYSCTLINFOBYNAME(char *name, int prop[2], void *buf, size_t *buflen);
```

The macros seek the node with id/idlevel or name, then the information specified by prop is copied into the buffer buf. Before the call buflen gives the size of buf; after a successful call buflen gives the amount of data copied; the size of the info can be determined with the NULL argument for buf, the size will be returned in the location pointed to by buflen. The value of prop[0] should be CTL_SYSCTL and prop[1] can specify the desired info, the possible values are defined like constants corresponding to the sysctlinfo nodes [16]. SYSCTLINFO and SYSCTLINFO_BYNAME return the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.
Implementation note

The core of sysctlinfo is just the sysctlinfo_interface() function, it implements all the nodes using nothing from kern/sysctl.c so sysctlinfo can be loaded as a module or merged anywhere in the kernel (possibly kern_mib.c).

In capability mode sysctlinfo checks if the node has the CTLFLAG_CAPRD or CTLFLAG_CAPWR flag before to pass its info to the userland, the exceptions are: sysctl.entryfakename for compatibility with sysctl.name and the explores sysctl.entrynextnode and sysctl.entrynextleaf to allow to traverse the MIB-Tree.

The *byname nodes are almost implementation free, they search the node by its name then the code of sysctlinfo_interface() remains unchanged.

The sysctl MIB-tree is a critical section, while sysctlinfo_interface() explores the tree and passes the info to the userland no nodes can be added or deleted, unfortunately sysctl(3) releases the lock (properly sysctl_root_handler_locked()) before to call the handler of the a node, correctly the nodes of the current interface use SYSCTL_RLOCK(tracker) to take the reader-lock.

The solutions of sysctlinfo are:
• Using sysctl_wlock() and sysctl_wunlock() to get the writer-lock, actually a reader-lock is sufficient but the kernel does not provide this KPI outside kern_sysctl.c so this solution is suitable for the kernel module
• Building a kernel patch, sysctlinfo.diff [16], to provide sysctl_rwlock() and sysctl_rwlock() as KPI to use SYSCTL_RLOCK(tracker) outside kern_sysctl.c.

The *allinfo nodes serialize the info of a node, it is not possible to pass the struct sysctl_oid explicitly because the struct has not idlevel, moreover oid_number and oid_name are not absolute but relative to the node.

4 Real world use cases

The sysctlinfo interface is available via a FreeBSD port sysutils/sysctlinfo-kmod or by applying the sysctlinfo.diff patch, the latter is more efficient because uses a shared-lock, moreover some BASE utility is been converted: sysctl, sysctlbyname() and sysctlname2oid(), they should be used to manage an object with an OID with 23 or 24 levels or if some level-name is just the NULL string.

The tools using sysctlinfo are: sysctlview [18] and nsysctl [19] (a sysctl(8) clone supporting LibXo [7] and extra options [Listing 12]).

Listing 12: nsysctl utility
% nsysctl --libxo=xml,pretty -NdF
 kern.features.compat_freenbsd_32bit
 <object>
   <name>kern.features.compat_freenbsd_32bit</name>
   <label>description</label>
   <description>Compatible with 32-bit FreeBSD</description>
   <type>integer</type>
   <format>[1</format>
   <true-flags>
     <flag>RD</flag>
     <flag>MPSAFE</flag>
     <flag>CAPRD</flag>
   </true-flags>
   <value>1</value>
 </object>

Table 1: Interfaces comparision.

<table>
<thead>
<tr>
<th>Current interface</th>
<th>sysctlinfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysctl.name</td>
<td>sysctl.entryfakename</td>
</tr>
<tr>
<td>sysctl.next</td>
<td>sysctl.entrynextname</td>
</tr>
<tr>
<td>sysctl.oidfmt</td>
<td>(divided into entrykind and entryfmt)</td>
</tr>
<tr>
<td>sysctl.oiddescr</td>
<td>sysctl.entrydesc</td>
</tr>
<tr>
<td>sysctl.oidlabel</td>
<td>sysctl.entrylabel</td>
</tr>
<tr>
<td>sysctl.name2oid</td>
<td>sysctl.entryallinfowithnextnode</td>
</tr>
<tr>
<td>sysctl.oidinputbyname</td>
<td>sysctl.entryidinputbyname</td>
</tr>
<tr>
<td>sysctl.idbyname</td>
<td>sysctl.entryidbyname</td>
</tr>
<tr>
<td>sysctl.entrydescbyname</td>
<td>sysctl.entrydescbyname</td>
</tr>
<tr>
<td>sysctl.entrylabelbyname</td>
<td>sysctl.entrylabelbyname</td>
</tr>
<tr>
<td>sysctl.entrykindbyname</td>
<td>sysctl.entrykindbyname</td>
</tr>
<tr>
<td>sysctl.entryfmtbyname</td>
<td>sysctl.entryfmtbyname</td>
</tr>
<tr>
<td>sysctl.entryallinfobyname</td>
<td>sysctl.entryallinfobyname</td>
</tr>
<tr>
<td>sysctl.entryallinfobynamewithnextnode</td>
<td>sysctl.entryallinfobynamewithnextleaf</td>
</tr>
<tr>
<td>sysctl.entryidinputbynamewithnextnode</td>
<td>sysctl.entryidinputbynamewithnextleaf</td>
</tr>
</tbody>
</table>

Table 1: Interfaces comparision.
The `sysctlbyname-improved` project \[17\] uses the code of `sysctlinfo` to provide an improved clone of `sysctlbyname()`, its implementation core is a new internal node to resolve the OID of a node by its name eventually extended with an input for the handler.

Finally the `sysctlmibinfo2` library \[15\] implements a high level API by wrapping `sysctlinfo` and `sysctlbyname-improved`.

5 Conclusion

This paper presented `sysctlinfo` a new interface to explore the `sysctl MIB` and to get the info about an object. The new interface tries to improve the efficiency, to implements new features and to address the limitations of the current interface, the latter is used by a multitude of tools and libraries so both interfaces have to coexist in the same kernel, this requirement is respected.

The interfaces are implemented by internal nodes, the `sysctl` syscall has to find them, then their respective handlers have to explore the `MIB` again to find the specified object. This approach suffers overhead, however it is not excessive because the internal nodes belong to the first sub-tree of the `MIB`.

In the future, a different solution could be a `sysctl-SNMP` design: the OID is extended to specify a desired info, then the `sysctl` syscall has to find just the wanted object. This efficient solution requires non-trivial changes to the `sysctl` implementation in `kern_sysctl.c`, therefore the internal nodes are a right trade-off between efficiency and simplicity.

6 Acknowledgements

I would like to thank the members of the FreeBSD community to build an awesome operating system sharing their code and providing excellent documentation.

References


