Nsswitch development: nss-modules and libc separation and caching daemon extensions

Michael Bushkov
bushman@freebsd.org

Southern Federal University,
Rostov-on-Don, Russia

Abstract

This paper describes the extensions to the FreeBSD nsswitch subsystem, that should be committed to the source tree in the nearest future and the issues that had to be solved to make them. These changes are:

1) The separation of the libc and nsswitch modules, which makes libc code much lighter and nsswitch subsystem more dynamic. It also allows proper use of the nsdispatch(3) calls from the userland.
2) New features, that were added to the caching daemon (full “perform-actual-lookups” option support, “precache” and “check-files” option). They make it much more usable and similar in functionality to Linux/Solaris nscd, while having its own unique features.

Preface

The work, described below, was made during and after the Google Summer Of Code 2006, which I was lucky to participate in, working for the FreeBSD community. It is not yet committed to the -CURRENT, but I hope it to be finally reviewed and committed in the nearest future.

Nss-modules and libc separation

The idea of nss-modules and libc separation is quite straight-forward: we should make several dynamic libraries (nss_files, nss_dns, nss_compat, nss_nis) and move appropriate code from libc to them. Appropriate code is the functions which were specified as the sources during nsdispatch(3) calls.

Several issues had to be solved to separate nss-modules from the libc.

Issue 1. Common functionality

Common functionality was the almost ubiquitous problem of all nss-modules. As all nsswitch sources for the particular database usually reside in 1 file (getpwent.c, for example), their functions usually use some common routines (pw_scan, for example). To move such modules from the libc with minimal changes, common functions were moved to the internal libnssutil static library. This library is compiled with ${PICFLAG} to allow linking with shared libraries - i.e. nsswitch modules. It contains quite general routines (like copy_htent() and copy_netent()) and is used from nss_files, nss_nis, nss_dns, and nss_compat. It can also be useful if some new nss-module is introduced.

Getipnodeby**(3) functions had a lot of common functionality issues. It turned out that the simplest way to solve them is to implement getipnodeby**(3) functions not through nsdispatch(3) calls but through gethostby**(3) calls. Such modifications were made and tested for compatibility with current implementations (nsswitch regression tests, that are described below were used to ensure that the behavior of these functions didn't change).

Issue 2. Threading and private libc includes issues

All nss-modules use thread specific storage (thread local storage) by using either NSS_TLS_HANDLING or NETDB_THREAD_ALLOC macros from nss_tls.h and netdb_private.h respectively. Both of these files are libc-internal. And they both require all pthread-related calls to be hidden with namespace.h/un-namespace.h includes. To allow nss-modules to be

moved out of the libc with minimal changes, <pthread.h> and <pthread_np.h> includes are enclosed with "namespace.h" and "un-namespace.h" in their source code. Path to libc/include is added to the standard include path for each module to allow "nss_tls.h", "netdb_private.h" and other libc-private files inclusion.

Such an approach allows to move out the modules from the libc to separate libraries with minimal changes to their sources, which is very useful, until this work is finally committed. The drawback of such decision is the dependency of the nss-modules code on the libc code. This dependency can surely be broken after the modules are separated in –CURRENT. For example, if all modules use only NSS_TLS_HANDLING macro to handle thread local storage data, then it will make netdb_private.h unneeded. The nss_tls.h can be modified not to use hidden versions of pthread calls and placed in the libnssutil folder (it would have to be left in libc also - as it is used not only from nss-modules but also from the libc itself). Other libc-private includes can also be easily eliminated from the modules' sources.

Issue 3. Statically linked binaries

Statically linked binaries can't call dlopen(3). But when all nss-modules are moved out from the libc, dlopen(3) is the only way to use them. To solve this issue, not only the dynamic versions of the nss-modules, but also their static versions, should be built. Libc's Makefile was modified to link statically built nss-modules in (please see Appendix A for details).

Nsdispatch.c has the nss_load_builtin_modules() function, which loads the statically linked modules into the libc at program startup. In the shared libc.so each modules' entry function is now replaced with an empty stub. In static libc.a each modules' real entry functions are used. nsdispatch.c was slightly modified to correctly distinguish real module entry functions from a stub.

The approach, that was used to link-in nss-modules into the static libc.a is quite flexible - new module can be added to the list of linked-in modules without any problems as long as it can be built as a static library (plus some 1-line changes would need to be made to the libc). The possible extension of this approach is to:

1. Make the list of the linked-in modules extendable via macro definitions, that can be defined during the buildworld.
2. Add an option to the nss-modules ports to build statically linked libraries along with shared ones.

With these changes made, the user will be able to link-in any prebuilt nss-module into the libc during the buildworld process. This would allow him to use this module's functionality with any of the statically linked binaries (/rescue is the most important example, probably) without any restrictions

Benefits of separating nss-modules from the libc

1. The code of both libc and nsswitch modules became much cleaner. The common functionality was placed into the libnssutil library and the number of interdependencies between libc and nsswitch modules sources was reduced to minimum. The code of the particular nsswitch module is not spread over several libc files, but is located in one library.

2. The described above ability to add the particular module support to the libc without any pain is now present.

3. There is now an ability to actually use nsdispatch(3) routine not only from the libc. The use of nsdispatch(3) was limited because of the number of the opaque pointers (in the dtab structure, that describes the list of nss-modules and their entry-points), that were needed to be passed in order for nsdispatch(3) to use libc built-in modules. When all nss-modules are standalone, the need in these pointers became obsolete, so nsdispatch(3) can be used and will properly work not only in the libc, but also in any other place. That gives an ability to properly support "perform-actual-lookups" option for all nsswitch databases in the caching daemon (please see the details below).

Nsswitch Regression Tests

The basic idea of the regression tests is to check that the expected functions behavior doesn't change after their sources modification. The idea of the regression testing for nsswitch is that the nsswitch query results should be generally the same after the system or nss-modules upgrade (if we don't change the databases, of course). The test procedure itself is very simple: we make a set of nsswitch queries (get**ent(3), get**byname(3) and get**byid(3)

calls) and store their results in a file. When the test is done next time, it does the same queries in the same order and checks that their results are equal to the stored ones.

So, the testing is done in 2 stages.

First stage is the snapshot creation stage. We run the test and it builds a snapshot file of the nsswitch queries results. It also checks these results for correctness – numerical values must be in the correct range, (char *) strings that should not be NULL must not be NULL. For the resolver functions we can check that the ip address length corresponds to ip address type and, if the address was mapped from ipv4 to ipv6, that it was mapped correctly.

During stage 2 we use the already created snapshot to perform the same set of queries and then compare their results to the ones in the snapshot. We also check all results for correctness on this stage.

Such kind of testing can be used to test any existent nsswitch module. For example, we can take FreeBSD6-STABLE, run the first stage of the test, then upgrade to CURRENT and run the second stage of the test. The test will show all the compatibility issues between versions of nsswitch-dependent functions.

All nsswitch regression tests are C programs, that use the same testutil.h file, which carries most of the common logic (mostly in the form of macro definitions). The command line arguments are the same for almost all tests:

-d - enables debug output, which helps to debug the test itself and to get more information in case of test failure

-n - runs test for the get**byname(3) function

-e - runs the test for the get**ent(3) functions

-g, -u, -p – run the test for getgrgid(3), getpwuid(3) or getservbyport(3) functions accordingly

-s <file> - causes the snapshot file to be created or, if it already exists, to be used to check the equality of the nsswitch queries results

The described regression tests were used while work on libc and nsswitch modules separation was being done. Their output was used to ensure that the behavior of the system with all modules built into the libc is equal to its behavior with all modules separated. They've especially helped during the getipnodeby**(3) functions reimplementation through the gethostby**(3) calls.

Regression tests can also be used to ensure that the caching daemon works correctly. To do that, we make a snapshot, when the caching for the particular nsswitch database is turned off, then we turn it on, run the stage 1 again (without rewriting the snapshot file), so that all necessary data are cached and then run stage 2 test with the snapshot file. If any error occurs during the caching process or caching daemon's marshalling/demarshalling process, it will most probably be mentioned in the test output.

Cached performance analysis

Cached gives tremendous and easily explainable performance boost for network-related nsswitch queries – LDAP is the best example, probably. That's why comparing the performance of the, for example, "passwd" nsswitch database queries to LDAP with and without caching is not of much interest. Much more interestingly is to compare caching daemon speed with the speed of the fastest nsswitch source: "files".

To do the comparison, we've used the "passwd" and "services" databases, which are quite different in their current implementation: "passwd" relies on BDB and "services" – on plain files.

We modified the sources of the getent utility so that it began to write getrusage(2) information to the stdout after each nsswitch query. Then, for each test we ran getent multiple times, forcing it to do 2 queries at 1 run. Only the speed of second query from each run was taken into account, because the first query always involves much overhead for reading nsswitch.conf file, loading nsswitch modules, caching the results, when caching was enabled and so on. The results were collected in the files and then processed by python script. For each type of testing (we used getpwnam(3) for "passwd" testing and getservbyname(3) for "services" testing), total of 10000 requests were made, 5000 of them were taken into account. For "services" database, half of requests were made for the data in the top part of the /etc/services file and half – for the data in the bottom of this file, because the time of the getservbyname(3) call is proportional to position of the needed data in /etc/services.

Here are the numbers (in microseconds), evaluated in different caching conditions:

Caching turned off

“passwd” nsswitch database	
Total time:	44880.00
Average time:	44.88
Median time:	47.00
Standard deviation:	13.39
Minimal time:	27.00
Maximum time:	157.00
“services” nsswitch database	
Total time:	5529766.00
Average time:	552.98
Median time:	1069.00
Standard deviation:	492.91
Minimal time:	30.00
Maximum time:	1209.000

Caching turned on (caching daemon is in single threaded mode):

“passwd” nsswitch database	
Total time:	102717.00
Average time:	102.72
Median time:	100.00
Standard deviation:	21.58
Minimal time:	71.00
Maximum time:	197.00
“services” nsswitch database	
Total time:	1010379.00
Average time:	101.04
Median time:	169.00
Standard deviation:	22.31
Minimal time:	71.00
Maximum time:	214.00

Caching turned on (caching daemon is in multithreaded mode – 8 threads):

“passwd” nsswitch database	
Total time:	124147.00
Average time:	124.15
Median time:	150.00
Standard deviation:	27.58
Minimal time:	78.00
Maximum time:	232.000
“services” nsswitch database	
Total time:	1213242.00
Average time:	121.32
Median time:	137.00
Standard deviation:	28.25
Minimal time:	80.00
Maximum time:	257.00

While showing good results (about 5,5 times faster) with caching enabled for "services" database, this test shows the ugly truth - it's nearly impossible to beat BDB query time with caching daemon’s query time. This fact makes using cached for local sources very questionable (not impossible, though). BDB is obviously the fastest solution, but caching daemon caches all plain files information in the uniform way, it can perform checks on local files to update cache if they are changed (with all precautions of not flushing the old data if something is wrong with the updated file) and do precaching on startup (please see below), it is more lightweight solution, that does not require BDB in tree. But, once again, if the speed is the main and only concern, then BDB is the choice.

Actually there are 3 areas, where cached’s speed can be improved:

1) Socket I/O
2) Multithreading
3) Lack of performance-improvement features

Socket IO optimizations appeared very hard to be done without major changes of the cached's architecture. And, most of the socket I/O-related calls have normal execution time, which however is much longer than BDB-related calls time. Because of these 2 reasons, no significant changes were made to the socket I/O part.

Multithreading issues doesn’t seem (according to the numbers above) to affect the caching daemon’s speed much.

Because of the described reasons, Item 3 was considered to be the most perspective way to improve cached's performance in certain cases., so the precaching feature was added to the caching daemon (please see below).

Cached extensions

"perform-actual-lookups" option full support

The nss-modules and libc separation allowed adding full support for the "perform-actual-lookups" option to the FreeBSD caching daemon. With this option turned on, cached acts exactly like Linux/Solaris nscd daemon for the particular nsswitch database - i.e. it makes requests by itself and not only caches the results, supplied by the user.

"precache" option support

"precache [cachename] [yes|no]" option support was added to the caching daemon. With this option turned on, the caching daemon precaches the specified database at startup (and, possibly, recaches it in case of local file change – please see below).

Precaching can be very useful for such databases as "services" when "perform-actual-lookups" method is turned on. If we precache data on startup, all queries to the cached would be read_request-search-read_response queries (without any write operations). And this type of queries is the fastest one in the caching daemon. It has no overhead of writing to cache, or of performing the nsdispatch(3) lookup.

This option proper support was also made possible only by the libc and nsswitch modules separation.

"check-files" option support

"check-files [cachename] [yes|no]" option is now also supported by the caching daemon. With this option turned on, cached flushes the cache for the particular nsswitch database automatically when its corresponding local file is changed. For example, cache for groups is flushed in case of /etc/group file change.

The lack of this option made caching daemon sometimes unusable during several ports installation process and required system administrator to flush the cache manually after any local database update.

FreeBSD caching daemon and nscd

The libc and nss-modules separation and cached extensions, that were made possible because of it, are directed to make nsswitch subsystem more powerful and flexible.

With all its current features FreeBSD caching daemon became similar in many terms to the nscd daemon, used in other OSes. It has its unique feature, though - the ability to rely all the nsswitch requests on the user side, and only cache their results by itself. However, because of the similar functionality and compatible configuration files, caching daemon will be probably renamed to nscd, when the work, described in this paper is committed.

Conclusion

Most notable features, that the work, described here, gives to developers are: cleaner libc and nsswitch-modules code, the easy process of adding a particular module to the list of libc's built-in modules and ability to use nsdispatch(3) not only in the libc. The latter was used to add several useful options to the caching daemon and can be possibly used to build specific nsswitch tools (like the mentioned caching daemon or getent command, for example). The described regression tests can be used in future nsswitch development to ensure the invariance of the nsswitch-related libc functions behavior.

Appendix A

```
# Include nss-modules's sources so that statically linked apps can work
# normally
NSS_STATIC+= ${.OBJDIR}/../nss_files/libnss_files.a
NSS_STATIC+= ${.OBJDIR}/../nss_dns/libnss_dns.a
NSS_STATIC+= ${.OBJDIR}/../nss_compat/libnss_compat.a
.if ${MK_NIS} != "no"
NSS_STATIC+= ${.OBJDIR}/../nss_nis/libnss_nis.a
.endif
NSS_STATIC+= ${.OBJDIR}/../libnssutil/libnssutil.a

# NSS-modules should be linked into the libc.a
nss_static_modules.o:
        ${LD} -o ${.TARGET} -r --whole-archive ${NSS_STATIC}

# libc.so should have stubs instead of module-load
# functions
nss_stubs.So:
        ${CC} ${PICFLAG} -DPIC ${CFLAGS}\
        -c ${.CURDIR}/net/nss_stubs.c -o ${.TARGET}


.if ${MK_PROFILE} != "no"
nss_static_modules.po:
        ${LD} -o ${.TARGET} -r --whole-archive ${NSS_STATIC}
.endif

DPSRC=  nss_static_modules.c nss_stubs.c
STATICOBJS+= nss_static_modules.o
SOBJS+= nss_stubs.So
CLEANFILES+= nss_static_modules.o nss_stubs.So
```

Appendix B

The details of the described work along with the patches can be found on the FreeBSD wiki:
`http://wikitest.freebsd.org/LdapCachedDetailedDescription`
`http://wikitest.freebsd.org/MichaelBushkov`

The code is located in the perforce branch:
http://perforce.freebsd.org/depotTreeBrowser.cgi?FSPC=//depot/projects/soc2006/nss%5fldap%5fcached/src&HIDEDEL=YES