How to break long-term compatibility in NetBSD

Jörg Sonnenberger <joerg@NetBSD.org>

Abstract

NetBSD has been maintaining ABI compatibility of libc since 1997. The implementation has undergone major changes in the mean time. Thread-awareness or the move to 64bit ino_t and time_t are just a few examples with huge impact on the external ABI. Some desirable changes like supporting more than 32767 open files in stdio are impossible without breaking this compatibility. If the compatibility is broken, it allows fixing a number of historic mistakes and putting better abstractions in place for future improvements. The high price mandates a detailed strategy, covering necessary changes, desirable changes and a plan for testing. This paper aims at identifying the problems and providing choices for their solution. It allows a clear understanding of the process and improves collaboration between the interested parties.

1 Introduction

NetBSD has been maintaining ABI compatibility on the kernel level all the way back to NetBSD 0.9. The first intentional break of compatibility on the system call level was the removal of Scheduler Activations with NetBSD 6.0 after libpthread changed to a much simpler 1:1 threading model with NetBSD 5.0. The dynamic linker maintains compatibility back to NetBSD 1.3, when NetBSD switched from a.out to ELF. The libc ABI has been kept backwards compatibility since 1997. It is easy to see why such compatibility is useful. When maintaining production systems, it is commmon to have some binaries without matching source code or even binaries where the source code was never accessible at all. Breaking compatibility makes such programs more difficult or impossible to use. Removing compatibility on the kernel level requires keeping a (virtual) machine with the old kernel running; removing compatibility in libc or the dynamic linker might require a separate chroot. The compatibility comes with a cost though:

1. Code bloat for conversion functions or redundant implementations.
2. Code bugs due to rarely exercised code paths.
3. Security issues due to bugs and conflicts with modern mitigation techniques.
4. Frozen design choices.
5. Difficult to change assumptions.

It is not surprising that the different members of the BSD family evaluate the cost/benefit relation differently. OpenBSD decided to implement the migration to 64bit time_t with a flag day, where NetBSD's earlier implementation involved a significant amount of glue in libc. The flag day approach limits the work for developers as they have to spend less time on identifying the full impact of a change on the ABI. The time_t type is used as part of many structures and has even been used as part of some on-disk formats in the past. It is not always easy or even possible to rebuild programs, since e.g. the source code might be owned by a third party or lost due to an accident.

For NetBSD the balance has slowly been moving toward breaking compatibility on the userland side. Such a move allows addressing a great variety of issues:

1. Removal of compatibility code.
2. Moving legacy interface like `gets` into separate libraries.
3. Changing interfaces to be explicitly versioned or more friendly to future extensions.
4. Moving non-essential functionality into separate libraries or plugins.

5. Providing a consistent mechanism for linking plugins into static binaries.

6. Clarifying architectural choices like ownership of global resources or the interaction between libc and libpthread.

7. Catch up with platform specific changes in the runtime ABIs like moving from init sections to init array sections.

This paper aims at identifying the problems in the libc and dynamic linker ABI contract that should be addressed in one logical step. Approaches for solving the problems either incrementally or by using a flag day are discussed. Based on this analysis, a migration strategy is proposed. This provides a detailed foundation for developers, release engineering and third parties to agree on one approach.

2 Interface versioning

Interface versioning is necessary when changing the (physical) layout of a structure by changing the size of individual members. Adding new fields can work, if the total size is not part of the ABI contract, i.e. if all instances are created and copied using functions of the library "owning" the interface. Special care is needed for global variables since access from the main program can implicitly leak the size of the variable via copy relocations.

In this section, two specific variants of the problem are investigated. The first part concerns userland ABI contracts between shared libraries as well as shared libraries and the main program. The second part concerns the ABI contract between userland and kernel.

2.1 Shared libraries and interface versioning

The problem interface changes creates for shared libraries is best illustrated by an example. Consider NetBSD's switch to 64bit time_t. The `time` system call takes a pointer to time_t as argument, returns the current time and modifies the location referenced by the pointer. If the application using this function has been compiled for a 32bit time_t, it will expect that only that much memory is modified. It will generally also provide a location aligned

```
typedef int32_t  off_t;
typedef int64_t  off64_t;

int open(const char *, int, ...);
int open64(const char *, int, ...);
off_t lseek(int, off_t, int);
off64_t lseek64(int, off64_t, int);

#if _LFS_LARGEFILE - 1 == 0
#define open open64
#define lseek lseek64
#define off_t off64_t
#endif
```

Listing 1: Symbol renaming for LFS

```
typedef int64_t time_t;

time_t time(time_t *) asm("__time2");
```

Listing 2: Symbol renaming via assembler name

for 32bit access. As libc can't know if the caller has been compiled for 32bit or 64bit time_t, it has to provide two different versions of the function with different name on the object code side. It must also ensure that new code gets the 64bit version during compilation. Two different approaches have been used to address this:

1. Symbol renaming in the C headers,

2. Symbol versioning during linking.

Symbol renaming itself can be implemented using macros (figure 1) or via the GCC extension of assembler names (figure 2). Macros are very intrusive and can easily break valid source code in surprising ways. The GCC extension on the other hand is exactly that, a language extension that purists will scoff at the pollution of headers with it. NetBSD has been using assembler names since 1997 to implement symbol renaming. The author is not aware of any other OS making extensive use of this feature. Macro based symbol renaming is most prominently used by the Large File Summit (LFS) option for 64bit off_t. The optional LFS support can still be found in GNU libc and most commercial Unix variants. The main advantage of symbol renaming is that it allows building older source code

```
typedef int32_t time32_t;
typedef int64_t time64_t;

asm(".symver time,time@NetBSD_5.0");
asm(".symver time2,"
    "__time@@NetBSD_6.0");

time64_t time2(time64_t *ts) {
  /* do magic */
}

time32_t time(time32_t *ts) {
  time64_t t = time2(NULL);
  if (ts)
    *ts = t;
  return t;
}
```

Listing 3: Symbol versioning, source part

against a newer library by providing it with the same old header files. On the other hand, it can interact badly with broken configure scripts that don't include headers and assembler names don't provide a good way to extract the symbol name as string for use with `dladdr` or other situations like Foreign Function Interfaces in Python, Java or other non-C derived languages. A compiler built-in like `__builtin_asm_name` could be used to address part of the `dladdr` problem, but there is no easy answer for the other use cases.

A special purpose variant of symbol renaming has been introduced with GCC 5.1 in the form of the `abi_tag` attribute. It is used to allow coexistance of the old copy-on-write string implementing and the new implementation needed for C++11 compliance in one shared library. Existing means like C++11 inline namespaces (as used by libc++) would have solved most of the problems as well with one exception. If two functions share the same argument types, the Itanium name mangling convention will produce the same assembler name. This makes it impossible to have a function returning the old string type and a function returning the new string type in the same namespace.

Symbol versioning is implemented by a combination of function annotations in the implementation (figure 3) and an additional linker input file,

```
NetBSD_5.0 {
};
NetBSD_6.0 {
} NetBSD_5.0;
```

Listing 4: Symbol versioning, version map

the version map (figure 4). The version map specifies a tree of symbol lists via inheritance. Every symbol in the `NetBSD_5.0` category is also part of `NetBSD_6.0`.

In the example, the `NetBSD_6.0` version of `time` is the default version as it is using `@@`. If a symbol has no default version, it can only be used by existing shared libraries or by specifying a version explicitly. This can be used for obsoleting interfaces without breaking existing programs.

A single implementation can have more than one version associated with it. This is useful for release engineering when backporting new interfaces to older version. It would look strange to have a `NetBSD_6.0` symbol in libc from NetBSD 5.1, so a new version tag `NetBSD_5.1` would be needed in this case. To allow forward compatibility, the same version tag is also needed in NetBSD 6.1, so that binaries from NetBSD 5.1 can run on NetBSD 6.1. Assigning more than one version to a single implementation is more tricky than necessary though, as GNU as insists on a one-version-per-alias rule. It is still open for discussion whether this (mis)feature should be just disabled and whether it should possible to create versioned symbols without exposing the intermediate artifacts in the symbol table.

Another limitation of the symbol versioning definition from GNU is the interaction between relocations using versioned symbols and unversioned overrides. Normal ELF rules for unversioned symbols specify a search order starting from the main program and the first match wins. This allows f.e. replacing the libc memory allocator with a custom version that better reflects the specific needs of an application. In the world of versioned symbols, it is difficult to see this as desirable. If a symbol has more than one version, overriding all of them with a single replacement seems to be wrong most of the time. If the replacement rules are changed to consider the version of a symbol an inherent part of the identifier, new applications appear as well.

Tagging symbols in headers explicitly with versions would create a kind of name space system for C, so that an application program can use identifiers like `dprintf` without interfering with the libc definition as long as it is built f.e. with strict C11 header visibility. An unversioned or default fallback symbol in libc would allow autoconf-like feature detection to still work and the existing linker warnings when linking against certain symbols can detect cases of using system functions without including the proper headers. The downside is that `LD_PRELOAD` replacements would have to include the system headers, so that they know which versions to replace.

Over all, the primary advantages of symbol versioning are less pollution of headers for the common case of using the most recent version of a symbol and better support in the toolchain. The additional run-time validation adds a slight increase to the load time. Handling release branches gets a bit more complicated, but also safer. Switching to symbol versioning is therefore a net gain.

2.2 Kernel interfaces

Kernel and userland interact via a number of different mechanisms:

1. System calls,
2. System controls,
3. I/O controls.

System calls follow the C function call ABI and generally are optimised for raw performance. Change to the argument lists or the return type are implemented by adding a new system call with glue in the kernel to convert the old types to the new ones as needed. This is the same process as used for running 32bit processes on a 64bit kernel. Historically, extended system calls only added to the list and never deprecated the existing ones. The addition of `openat` for example didn't move `open` to a compatibility list, even though libc can easily provide `open` in userland. The advantage of keeping both system calls is that the kernel normally has to keep them for compatiblity anyway, so the translation in libc would be duplicated code. On the other hand, especially for embedded use, compatibility options in the kernel are often disabled, so a leaner kernel can effectively reduce the total memory foot print.

The exceptions to the rule of system calls being optimised for raw performance are `ptrace`, `getcontext` and `setcontext`. While the performance is still highly relevant, they also have to deal with semi-regular changes on living architectures like x86. Support for the SSE2 and AVX instruction sets requires space in the platform-dependent register set definitions. Vector instructions on other platforms like MIPS, ARM or PowerPC provided similar issues in the past. Unlike other system calls, it is therefore important to (re)design these interfaces to allow for future growth. The next iteration will therefore include two important changes:

- The kernel exports the space needed for the full context via `sysctl`.
- The structures themselve are prefixed by their size and version field.

Extensions are still supposed to happen backwards compatible, even if that means duplication of data. Special purpose tools can still use a plain C structure, but are unable to access later platform additions. Libraries wanting to do signal frame inspection and other operations can allocate space dynamically and only parts wanting to deal with the new features have to fully support the new fields.

A review of other system calls is planned to check if they can benefit from explicit versioning. Some good candidates like the `stat` and `statvfs` families cannot directly benefit from versioning as their use is partially dictated by POSIX. Nevertheless providing a NetBSD-specific version for internal use in libc and an external userland-only view without the versioning fields can help pushing ABI stability concerns fully from the kernel/userland boundary into libc.

For system controls, API and ABI stability is handled inconsistently. Many driver and subsystem specific controls are created and removed at will following the development of the code. Other interfaces are used by many programs for querying the system state and are considered a central part of the kernel ABI contract. Querying the process table, the state of the VM subsystem or statistics of all network interfaces are just a few example of those. Just like system calls, NetBSD provides

compatibility handlers for dealing with older versions or 32bit programs running on a 64bit kernel for those. Performance constraints still apply in so far as the amount of data queried can be large and query rate can be high. The data itself is well structured, but often a mix of text and counters. As alternative to implementing versioning manually, using tools like Protobuf[Cam15] can automate the process. Schema validation helps to reduce the attack surface of the kernel for writeable interface. The schema themselve help the documentation and allow automatic human-friendly decoding during debugging.

The final category, I/O controls is the area with the least amount of systematic compatibility. The controls are identified by a single integer and interpreted according to device referenced. The BSDs encode the size of the argument structure into the integer, so addition of fields naturally result in a new I/O control number. The interface is problematic for a number of reasons:

- The context dependent interpretation of the request number makes it difficult to automatically decode the argument for debugging purposes. The `kdump` sources have hard-coded preferences to resolve conflicts. On NetBSD/AMD64 for example, the `meteor` driver and dtrace use overlapping requests.
- Passing variable-length data is difficult and error prone. Address space annotations could improve the situation a bit by allowing the compiler to flag direct access to userland pointers inside the kernel, but this is currently not used in NetBSD.
- No tools currently exist to automatically detect ABI incompatible changes.

The same approaches discussed for system controls apply here as well. A potential solution could involve a global registry of Protobuf schemas, identifying each one with a UUID. Due to the amount of legacy code using I/O controls to query the network stack alone, this is a daunting task though.

3 Performance vs. flexibility

The libc interfaces are often used in many performance sensitive functions. As such, the abstractions of the interface are sometimes broken by leaky implementations in the form of macros or inline functions. The two most important cases are the `ctype.h` predicates and the parts of the standard I/O interface related to character based I/O. The former uses bit masks into an array, making the bitmasks and element size part of the libc ABI contract. The latter exposes the FILE layout directly, so third parties can investigate or change things (gnulib has such hacks) or encode size, position and content of certain fields via the macros in `stdio.h`.

```
int true_elements;

void TEST(void) {
  int ch;

  for (ch = 0; ch < 256; ++ch) {
    if (isalpha((unsigned char)ch)) {
#ifdef FORCE_RELOAD
      asm volatile("" ::: "memory");
#endif
      ++true_elements;
    }
  }
}
```

Listing 5: Benchmark function for isalpha

Another issue of wide spread implications is keeping the POSIX thread interface as separate library or merging it into libc. They already have a very tight coupling as thread cancellation has to interact with system calls and libc requires mutexes in a number of thread-safe interfaces like `malloc`.

3.1 `isalpha` as representative of the `ctype.h`

The `ctype.h` predicates are commonly used in parsers. The current NetBSD implementation is of `isalpha` and friends is a macro and involves loading a pointer from a global variable and indirection with the character to be tested. As `isalpha` uses the process locale, the pointer load often needs to be redone as the compiler can't know if `setlocale` has been called. Using the `isalpha_l` interface for accessing an explict locale like the C locale doesn't have this constraint as a data associated with a locale_t instance doesn't change during its life time.

	isalpha macro	isalpha function	isalpha_l function
Dynamic program:	252 µs	856 µs	863 µs
...with reload:	288 µs	856 µs	863 µs
Static program:	246 µs	614 µs	615 µs
...with reload:	286 µs	614 µs	614 µs

Table 1: Run time of figure 5 on an AMD64 system

The current implementation of `isalpha_l` as function call doesn't exploit this knowledge though. A micro-benchmark (figures 5 and 1) shows the optimisation is still highly desirable as the function call overhead and the associated code generation changes slow down the function version by a factor of 2.5 even on a platform with very fast function calls. Two options are available for implementing this optimisation for the explicit locale variants:

- Make the position of the classification tables public parts of locale_t.
- Wrap the logic for obtaining the classification table in a function that the compiler can optimise.

The second option is clearly more desirable as it keeps the ABI contract small. GCC and Clang both allow tagging functions as immutable for a given set of arguments, but there is currently no attribute that cleanly matches the expections of arguments with a limited life-time. Discussions with both groups are underway to clarify and/or rectify this situation. The end result would be that a call to `isalpha_l` in a function would look up the locale specific classification table once and reuse it for all further calls, independent of any function calls in the function. For hot loops, this would cut the number of memory loads in half. As seen in figure 1, just the pointer reload is 10% of the run time of the simple test case. Other architectures benefit a lot more.

3.2 `getc` as representative of the standard I/O interface

Historically, NetBSD has provided macro versions of `feof`, `ferror`, `clearerr`, `getc` and `putc`. Even though `getchar` and `putchar` are macros as well, they only add the appropiate arguments for calling `getc` and `putc`. As such, they are not relevant for the purpose of ABI exposure. Of the five functions initially mentioned, `clearerr` is used much less often and generally only when handling error conditions or end-of-file. There doesn't seem to be a good reasonn for keeping it on the list as it is not performance sensitive. `feof` and `ferror` are commonly used in two ways:

```
void TEST(void) {
  int ch;
  FILE *f = fopen("test.txt", "r");

  while ((ch = getc(f)) != EOF) {
    if (ch != '0')
      abort();
  }
}
```

Listing 6: Benchmark function for getc

- Before closing a stream to see if everything has been processed without error.
- After individual operations indiciated a problem.

Neither case is typically the hot path, so dropping them from the macro list would only affect code using them directly as exit conditions of a loop or similar cases. Optimising for badly written code doesn't sound like a good justification. That leaves `getc` and `putc`.

A simple test function (figure 6) was used to evaluate the impact of the macro when processing one MB of zeroes. AMD64 is used as platform to represent modern CPUs with efficient function call handling. HPPA is special as it is the NetBSD with the highest function call overhead, especially across shared library boundaries.

The first notable result is that the `getc` macro on NetBSD is clearly too fast to be correct. A look at the code confirms that it is not thread-safe

	getc macro	getc function	getc_unlocked
AMD64 Dynamic program:	2.89 ms	11.3 ms	3.57 ms
AMD64 Static program:	2.91 ms	8.43 ms	3.39 ms
HPPA Dynamic program:	0.290 s	2.293 s	0.825 s
HPPA Static program:	0.350 s	1.587 s	0.644 s

Table 2: Run time of figure 6 on a AMD64 and HPPA

and cannot be used correctly in multi-thread programs. The design of the test case ensures that the buffer can be filled as much as possible, so the fallback path in the macro version to refill the buffer is rarely hit. Even then it is less than 20% faster than the getc_unlocked function on AMD64. The story looks different on HPPA though. The function version needs three times as long for dynamic linking, highlighting just how slow the architecture is for this use case.

The 20% gain of the macro version relative to the unlocked function call by itself would not justify exposing the internals of the FILE buffering. The factor of three seen on HPPA is significant enough to not make it a clear cut decision. Careful measurements on the other NetBSD architectures have to be done to verify whether HPPA is singular exception and to decide if it should affect the design here.

Another important result from the tests is that the macro version must be made thread-safe by default, but it is also clear that a build-time switch is desirable for programs that want to use the unlocked interface without code changes when they know that they are not going to use the same stream from different threads.

If macro versions of the `getc_unlocked` and `putc_unlocked` functions are to be kept, a minimal buffer interface must be exposed. For `getc_unlocked`, the best choice seems to be to expose the current position and the end of the read buffer, so that only one increment is needed for consuming input. For `putc_unlocked`, a slight complication is introduced by the difference between line buffering and full buffering. If the former is used, the buffer has to be flushed on newlines. As such, a flag is needed in addition to the current position and of the write buffer.

3.3 Multi-threading and libc

The history of multi-threading in NetBSD reflects the general of the last two decades toward multi-processor environments. When the native POSIX thread support started, new hooks in libc where introduced to allow a thread library to replace mutex stubs in a way that a *non-threading* program gets as small a performance penalty as possible. Over time, many (third party) libraries have become thread-aware by expecting stub implementations for mutex and conditional variables. When badly designed libraries appeared that mix thread-creating code with thread-aware code and expect to load such libraries via `dlopen`, even more parts of libpthread had to move into libc. At this point, it becomes a valid question on whether a separate thread library is useful or not.

One complicated part of the POSIX thread interface is the concept of cancellation points. One thread of a program can ask another thread to terminate whenever that threads hits the next cancellation point. Cancellation points are functions like `read` or `write`, so a full merge would introduce additional checks in many system calls. At the same time, a good amount of the weak aliasing of system calls in libc could go away. When using a hidden counter of all threads with outstanding cancellation requests, the common case of not using cancellation gets a penalty of one instruction relative load and predicted-true branch. Compared to the cost of system calls in general, that should be in the noise.

The other important part of the thread interface is the earlier question of avoiding or minimizing overhead for non-threaded programs. Two approaches are possible here. Keeping libpthread as library containing just `pthread_create` would allow declaring a program as non-threaded, if libpthread was not initially loaded. In that case, libc can check whether mutex functions with atomic operations are required or the cheaper non-atomic versions. Indirect functions can be set appropiately.

This makes the single threaded mutex operations as expensive as a function call and whatever basic sanity checking is desired. The other option is to have a hidden variable to determine if threading is active. This variable would be set when `pthread_create` is used first. The advantage is that it allows `dlopen` of libpthread without any restrictions, but it means that all atomic operations have to be protected by a check, which slows down every operation slightly. It is currently not clear which choice is better.

4 Balancing modularity vs. essential functionality

Over the years, NetBSD's libc has accumulated a number of interfaces that (by themselve) are clearly not essential for many programs. A major bump provides a good chance to verify which components really belong into libc and which don't. Four candidates for moving out of libc come immediately to mind:

- NIS (a.k.a. Yellow Pages),
- SUN-RPC,
- ISC's resolver interface,
- Berkeley DB (BDB).

The main reason for NIS keeping in libc so far are short-comings of the `Name Service Switch`(NSS) implementation. With the improvements outlined in section 6, the necessity for keeping it in libc goes away.

SUN-RPC is primarily used by NIS and NFS and with NIS no longer being needed in libc, the RPC code can move out as well.

The DNS resolver is a bit more involved. On one hand, the ability to resolve DNS entries is clearly required by central interfaces like `getaddrinfo` and moving those to a separate library like Solaris's libnsl seems a move in the wrong direction. On the other hand, just because libc must be able to resolve DNS entries, it doesn't have to publically expose its resolver. A practical approach is completely hidding the libc resolver under a prefix like `__resolver_` and using weak aliases (static linking) or indirect functions (dynamic linking) to provide a public interface library. Concrete, libc contains `res_nquery` under the name `__resolver_res_nquery`. In the static version, a weak alias `res_nquery` is provided, that can be replaced by other programs if necessary without getting a duplicate definition error from `ld`. For dynamic linkage, libresolv.so provides `res_nquery` as indirect function (STT_GNU_IFUNC). An indirect function will be called by the dynamic linker to determine the address for the GOT entry, i.e. `res_nquery` is called and returns returns the address of `__resolver_res_nquery`. Non-reentrant interfaces like `res_query` don't have to be implemented the dynamic libc at all. The advantage of this approach is that libc can move to a completely different resolver implementation and the only coordination is keeping libresolv.so in sync as it now has to provide the full implementation by itself. Candidates for alternative DNS resolvers exist, but none provide a good enough or well established enough interface that it.

The last item is BDB. The existing code base is ancient and code quality is still somewhat questionable. Bugs for edge cases still pop up every now and then in the BSDs. Newer versions of BDB exist, but Sleepycat and later Oracle decided to change the code license to something GPL-like. A code update is therefore not acceptable for legal reasons. As a general purpose storage backend, the BSD licensed version of BDB has serious limitations like the lack of transactions, very coarse locking schemes and no crash safety. NetBSD's libc had a number of users of BDB, to avoid parsing text files over and over again or building adhoc hash tables as cache. Most instances like `getservbyname` have been converted to use the NetBSD Constant Database (CDB) format instead. As the name implies, it is a read-only format. It features true $O(1)$ look-up complexity and the creation is typically significantly faster than rebuilding a BDB file from scratch. The two remaining users of BDB in libc are the password database and the `utmpx` record database.

The password database is somewhat complicated to replace due to the way NIS support has been integrated. Specifically, the NIS compatibility of `+` overrides and `-` exclusions significantly increase the code complexity. While explicit name based overrides and exclusions can be reasonably handled via additional hash-friendly look-ups, netgroup support is somewhat different. The existing code has thread-safety issues and the number of look-ups scaled linearly with the number of netgroup en-

tries in passwd. For new code, this would clearly not be acceptable and implementing it on top of a hash database like CDB requires explicit record chains. With NSS it would be possible to introduce a general overlay layer, specifying which backend to query and how to combine the results. It would require breaking the legacy passwd files though. A discussion of both approaches is still needed.

The `utmpx` database finally is a special issue as it is accessed as part of the login process and any form of locking is problematic when processes can be suspended via `SIGSTOP` by an attack while potentially holding a write lock. The older `utmp` file was a fixed-size record file indexed directly by the user id. This worked well enough when user ids were 16bit, but sparse files in the GB range are handled badly by many tools. The current BDB use on the other hand has the problem that a suspended process can hold the file lock and prevent other modifications. The author has devised a storage efficient file format for semi-sparse integer key access requiring compare-and-swap (CAS) on memory mapped files for quantities as the key. All NetBSD platforms except SPARC qualify for this. SPARC is special as it has both available and supported SMP hardware and lacks hardware support for CAS or equivalent functionality. No benchmarks comparing BDB and the new format exist yet.

In summary, moving BDB out of libc is a feasible project, but decisions about NIS compatibility in the password database have to be made first. The `utmptx` database can be replaced in the near future. Components like NIS and SUN-RPC primarily depend on fixing the NSS implementation, which can be tackled incrementally first. The resolver implementation can be easily isolated, hidden and optionally provided via a stub library to third parties while adding only initialisation overhead.

5 The dynamic linker and dynamic linking

The dynamic linker naturally forms a special contract with the rest of the system. Some of the interface changes are adjustments of historic mistakes, the NetBSD legacy interface of calling the dynamic linker via pointers in the object handle is a good example of this and detailed in the first subsection. Sometimes platform create strange requirements. PowerPC and HPPA are examples of platforms using function labels, but HPPA is special as it requires dynamic allocation in dladdr for them unlike every other platform. Memory allocations are tricky as they must be signal safe due to lazy TLS allocation. Finally, copy relocations are a problem when designing interfaces involving global variables with ABI compatibility in mind.

5.1 The legacy dlfcn.h interface

When ELF was originally introduced in NetBSD, `dlopen` and friends were provided by `crt0.o` as indirect function calls to addresses stored in the object handle passed from the dynamic linker. This was fixed in 2003 by providing only weak symbols in libc and strong versions directly from the dynamic linker. The old interface has to be retained since binaries created before 2000 still use the same `ld.elf_so`. This means removing fields from the main object definition in the dynamic linker or shuffling them around to better group them is not possible when they are early enough in the structure.

5.2 HPPA and inter-object calls

HPPA (or PA-RISC) has a somewhat peculiar calling convention for inter-library calls. Some (especially newer) architectures make it very easy to create Position Independent Code by providing both instruction-relative jump/call instructions as well as instruction-relative load/store. Other architectures like i386 use a relative call instruction to push the current instruction pointer on the stack and load it back to a general purpose register. Either way allows access to functions and variables of the current shared object, including the Global Offset Table (GOT) for indirect access to global functions and variables. For all those architectures, the PIC setup is the responsibility of the callee, the caller doesn't know about the details at all. For various historic reasons, the HPPA ELF ABI preserved many of the pecularities of the older HP-UX calling convention.

Calls to non-local functions on HPPA use the special `$$dyncall` helper function with the target as argument. If the second-lowest bit is not set

in the target addres, the helper function jumps directly to the target. This is used if the dynamic linker resolved the argument to a function within the same shared object. Otherwise, the argument is a procedure label (plabel), a pair of target address and the address of the GOT of the target. The helper function will now load the new GOT and call the real target. Every PIC function is responsible for reloading the previously saved value of the GOT register from the stack.

It is not obvious why this calling convention is any better than the approach used i.e. on i386, but it creates a specific challenge when plabels are used for indirect calls. The C ABI allows testing function pointers for equality, even across shared object boundariese. This means that taking the address of a (global) function must provide the same value from the main program and a shared library. Normally, the plabel is stored in the Procedure Linking Table (PLT) of the importing shared object and two shared objects referencing the same global function would therefore use different addresses as they have different PLTs. For even more questionable reasons, shared objects on HPPA do not contain space for the necessary plabels in the exporting library, so whenever the dynamic linker has a relocation that might be used as a function pointer, it has to search an internal table for an existing plabel and otherwise allocate additional memory for a new one. The same requirements apply to `dlsym` and similar functions. This is problematic as it costs time and requires an exclusive lock. It also leaks HPPA internals into generic platform-independent code.

There are two different approaches to fix this problem:

- Reserve space for plabels for every function in the dynamic symbol table.
- Fix the calling convention to work more like other platforms.

While the second approach would simplify things a lot, it requires changes to linker, compiler and various assembler modules. It is currently not seen as feasible. The other option is much more contained as the linker already has most of the pieces necessary in place. The 64bit HPPA linker already has some logic for creating a `.opd` section with the plabels, but retrofitting it into the 32bit creation is not that difficult either. It requires introducing either a new segment type in the program header to mark a part of the PLT. This part of the PLT contains one plabel for exported function, sorted by address. Normal PLT relocations can be used for resolving the relative function address and for storing the GOT reference in the plabel. Whenever the dynamic linker has to find find a plabel now, it can check if the defining shared object contains the new segment. If it does, all it needs to do is a binary search to find the authoritive plabel. A missing entry is a bug and doesn't have to be handled.

For objects without the new segment, it just has to allocate entries as before. If the dynamic linker has access to the section table of the binary, it can also process the `.dynstr` section on startup and build the same data structure dynamically. Since providing a section header is optional and no other way to determine the size of the dynamic symbol table exists, the old fallback path of individual allocation has to be retained as long as old binaries have to be supported.

5.3 Copy relocations

Copy relocations in ELF are a side effect of the desire to link the same object files of the main binary either dynamically or statically. For normal function calls, this is not a problem as the linker will insert a local stub in the PLT to do the actual call via the GOT. Problematic is the access to global variables or attemps to load the address of a function explicitly. When linking against a static library, the linker knows the precise absolute address of the symbol. When linking against a dynamic library on the other hand, the absolute address is only known at link time. If location containing the not-yet-known address is in a writeable section, the linker can just create a relocation, but this doesn't work for the text segment. Depending on the platform specific load/store instructions and the relocations, the linker might be able to turn the absolute address into a load via GOT, but if it can't, it has to cheat. This final solution is the dreaded copy relocation. What it means is that the linker duplicates the variable from the shared library in the uninitialized data section of the main program and instructs the dynamic linker to copy the content at run-time. Possible complications are changes of

alignment (which broke libc++ with Clang 3.7 on FreeBSD) and changes in size.

This problem can be solved in two different ways. The first approach is switching to position independent executables (PIE). Since the main program is a shared object as well, it will always use the GOT for accessing global variables. As PIE comes with both a performance and size penalty on most architectures, it is not always an option. The alternative is to selectively tag the moderately few variable exports from libraries with an attribute, similar to what is already done for ELF visibility. Now the compiler can generate code for indirect access to these variable and depend on the linker to remove most of the associated overhead when linking statically. When combined with an explicit header flag to forbid copy relocations, the problem would be gone.

6 Plugins and static linking

Plugins are a common part of modern system design. They allow extending the functionality at run-time without having to rebuild programs. Examples of interfaces using or even depending on plugins in the base system are NSS, PAM and the Citrus I18N framework. The natural way to implement plugins in ELF systems is via the `dlopen` interface of the dynamic linker, but this naturally is a problem with statically linked programs. GNU libc contains an attempt of providing dynamic modules for statically linked binaries, but it tends to create more problems than it actually fixes as the binaries are no longer standalone and the version restrictions are often not well understood.

NetBSD provides an ad-hoc workaround for linking a fixed set of PAM modules into static binaries. Patches for similar functionality in Citrus exist, but are not part of the base system. A common motive is that they require a non-trivial amount of code in each plugin framework, so a more systematic approach is desirable.

`crunchgen` is a special build tool used for merging a number of utilities into a single program, ideal for static linking. On NetBSD it is used for building the rescue binary, which includes most of /bin and /sbin in a single program. It works by linking the object files i.e. of ls together into a single relocatable object and then mangling the symbol table so to effectively hide all defined symbols except `main`. When renaming `main` to `ls_main` and adding a dispatcher based on called program name, various tools can be mixed together.

The `crunchgen` approach can be tweeked for an emulation of `dlopen` with minimal source changes. The first step is to link all object files of the modules together with `ld -r`. The second step is to extract a list of symbols for use with `dlsym` and friends. There are two basic policies here:

- Modules use ELF visibility to identify public symbols.
- Modules provide an explicit export list.

Independent of the source of the list, a helper module is built to create a linker set with the module name as used for `dlopen` and all exported symbols as strings and a reference to the symbol. The third step is to mark all symbols as local. As final step, all the helper module and the relocatable object from the first step are linked together again. The final result can be added to the linker invocation explicitly or in a bundle of other things with a linker script. No changes to the module source or the plugin framework itself are necessary. There is one important restriction that one module can't depend on another. This is generally considered bad design though, the primary example being X.org. Video drivers often depend on other modules to resolve some references, where as normally shared libraries would be used to provide common functionality. Even worse, the way the X server loads modules can interact badly with valid compiler optimisations like tail call elimination, when those optimisations prevent lazy linking. The implementations of `dlopen` and `dlsym` in libc are straight-forward. For performance concerns it might be useful to presort all symbol list in the helper modules, so that `dlopen` can extract the first and last symbol of a module for a binary search.

7 Overall migration strategy

This paper has outlined a number of changes with different levels of disruption. It is highly desirable to implement as many changes as possible iteratively on the main development branch to get wide spread testing and minimize development overhead.

Symbol versioning can be introduced in top of the existing symbol renaming (section 2.1) by keeping the existing (renamed) symbols and just tagging the oldest symbol as unversioned and the current symbol as default. The export list of libc can be trimmed conditionally to allow bulk build testing of a version without legacy support. Userland wrappers for deprecated kernel interfaces (section 2.2) can be introduced at any time as well. The discussion option of switching to Protobuf for the kernel interaction can be implemented without flag day as well.

The implement of the `ctype.h` family without explicit locale argument has proven to be nearly optimal (section 3.1) if thread-safety is a concern. The discussed optimisations for the versions with explicit locale argument are waiting on clarifications of function attributes. No need for further expansion of the current 16bit character classification is seen.

For the `stdio.h` family, the testing has shown two issues (section 3.2). The current macro versions of `getc` and `putc` should be used to implement `getc_unlocked` and `putc_unlocked`, without a fast path for the reentrant functions. A new option for requesting the non-reentrant behavior via compile time define should be introduced. All other macros can be dropped without replacement. The fallout from fully hiding the implementation of FILE can tested by moving the members into a new structure and making FILE have this structure as single member. This will create some churn in the libc code, but preserve the ABI. A new version of the buffer access is harder to test without ABI changes, so the final step for information hiding will likely have to happen on a branch.

Modularisation of libc (section 4) can be prepared by removing internal consumers and introducing (nearly) empty wrapper libraries. Converting BDB users like the `utmpx` database is an example for the former. Moving i.e. the resolver library from an internal namespace and providing the legacy interface via aliases and as separate library can be prepared, but the aliases in libc can only be removed via a non-default option or on a flag day. Fallout is expected for the resolver and BDB; NIS and SUN-RPC are less likely to be a problem.

Removal of the legacy interface of the dynamic linker (section 5.1) only involves cleanup, it doesn't require an actual code change. The HPPA plabel section (section 5.2) can be introduced as optional improvement, only the removal of the old interface requires a flag day. No clear strategy for copy relocations currently exists.

Merging libpthread into libc can be done at any time, the ABI contract of both libraries can be kept in place due to the ELF symbol look-up rules.

The new framework for plugins (section 6) only involves API, not ABI compatibility. Impact on third party software is estimated to be low.

One of the starting points for investigating a libc major bump was the libgcc dependency. The sources of libgcc have been tricky to use for a long time by making excessive use of conditional compiliation and other compile time tricks. The code is often written to make subtile use of GCC optimisations, so a different optimiser can introduce dependency cycles and other issues. The dependency on GCC's libgcc is already conditional and most platforms could switch at any point in time. The remaining exceptions are platforms with special "milli code" requirements, i.e. HPPA's division routine, which is linked into every shared library. The goal is to try to merge as much of the code upstream into LLVM's compiler-rt, but the provenience of the libkern source is sometimes difficult for the necessary MIT license change.

8 Summary and conclusions

In this paper, issues in the current NetBSD userland and kernel interfaces have been identified. Some of those choices can be incrementally addressed at the cost of keeping legacy code around, others are more fundamentally limiting. A strategy has been proposed for moving forward and preparing as many of the tasks as possible without breaking the ABI compatibility, as well as how testing can be done in preparation for the final ABI break. Based on the identified tasks, collabaration can be coordinated and the feasibility of finishing them in time of NetBSD 8.0 can be evaluated. Open questions around the migration towards Protobuf or other schema based extensible data formats remain and await a decision.

Further coordination with other platforms using ELF on the specifics of symbol versioning rules is desirable as well as how to make versioning anno-

tations less painful for developers. Compiler support has been identified as another area with shortcomings for specific performance sensitive interfaces.

At the end of the road, the userland will contain a separate dynamic linker and libc, so that old binaries can continue to work. Interoperability will be limited in some areas like having separate `utmpx` databases, but central files like the password database files can be provided in multiple versions.

References

[Cam15] Taylor R. Campbell. Protobufs for kernel/user interface. EuroBSDcon, October 2015.