GNU C Library Version 2.3

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

May 2, 2002

Abstract

The next major release of the GNU C library will be 2.3 which marks the beginning of a new phase of the development of the project. The level of functionality and standard compliance of other C libraries in the industry was reached or even surpassed. The main aspects of development today is to fix holes in the implementations (and standards) and optimization. The paper will introduce what is done in these two areas.

1 Preface

Version 2.3 of the GNU C Library is the beginning of a new phase of the development. Up to version 2.2 the main objective was to catch up with existing standards and industry practice. Since the implementation was not yet finished a lot of optimizations were not implemented.

With the next release all this changes. First, additions to the library go beyond what standards describe. Features users requested and which are found necessary in other projects were implemented. Additionally extensive optimizations of the runtime environment were performed and implemented with the result that program execution can be much faster.

The following text will first introduce the new features and then describe the various optimizations performed. Some of the optimization techniques are not only applicable to the C library. They are useful for all ELF shared libraries.

2 New Features

The number of new features the new release provides is high again. Not all of them can be explained here in great detail. We will limit the description of these features to a general overview. The next section will describe the most important new features in detail.

2.1 Thread-local Locale Model

The locale model in the ISO C and POSIX standard was designed in simpler times. Computers were often not connected or only connected locally which normally meant that all users of the computers are using is the same locale. Computers at different sites use different locales but at any one site there are normally no differences.

This all changed with the advent of globally connected machines with the help of the Internet. Suddenly one machine could service clients from all over the world with widely different requirements on the locales. These service programs are often written in a way that allows servicing multiple clients from one process (e.g., using threads).

To enable such programming it is necessary to allow the use of more than one locale at the same time in the same process. This is not possible in the moment. Switching between global locales using `setlocale` is not practical since this would have to happen before every single use of the locale (even for often used functions like `isspace`). In a multi-threaded process the `setlocale` call and the following use must be protected using a critical region which effectively reduces the process to serialize the threads for a large part of the time. When no threads are used the synchronization is easier but the costs are still prohibitively high. A different solution is needed.

Another source of problems is the need of the ISO C++ library implementation. The ISO C++ library implements locales as objects. This is not surprising and by itself no problem. But it is also required that locale objects can be constructed based on named POSIX locales. I.e., the functionality the C++ standard defines which is also available in ISO C must be implemented based on the C locale content. For instance, the `tolower` member function of the `ctype` facet of the C++ locale must produce the same result as a call to the `tolower` C function.

A partly complete solution for these problems has existed for quite some time in glibc. The 2.1 version of the GNU C library introduced the concept of locale objects with the introduction of the `__locale_t` type and the functions `__newlocale`, `__duplocale`, and `__freelocale`. These functions allow to create locale objects which have the same functionality as a locale which is selected using `setlocale`. To use these functions, special versions of the functions, which the standard defines to implicitly use locale information, are needed. To get the same results as a call to `isspace` but with locale objects the new function `__isspace_l` must be used. The interface is almost the same except for an additional argument which is the reference to the locale object:

```
  extern int __isspace_l (int, __locale_t);
```

Now the advantages of using locale objects should be obvious: it is possible to call `__isspace_l` in the same or in different threads with different locale objects. The function does not depend on any global state anymore like `isspace` did.

There is a price for this, though. The implementation has to provide a large number of new functions. The number of functions which depend implicitly on locale settings is pretty high. Second, the programmer has to rewrite code to take advantage of the new interfaces. This last point is valid only when the C interfaces are used directly. The standard C++ library implementation transparently uses these interfaces if they are available to provide the standardized locale functionality. I.e., the availability of the new interfaces does not require any changes in the C++ code; it will only exhibit more standard-conforming execution.

The functions which are available with the locale-object interface are:

`__isalnum_l`	`__isalpha_l`	`__isascii_l`
`__isblank_l`	`__iscntrl_l`	`__isdigit_l`
`__isgraph_l`	`__islower_l`	`__isprint_l`
`__ispunct_l`	`__isspace_l`	`__isupper_l`
`__iswalnum_l`	`__iswalpha_l`	`__iswblank_l`
`__iswcntrl_l`	`__iswctype_l`	`__iswdigit_l`
`__iswgraph_l`	`__iswlower_l`	`__iswprint_l`
`__iswpunct_l`	`__iswspace_l`	`__iswupper_l`
`__iswxdigit_l`	`__isxdigit_l`	`__strtol_l`
`__strcasecmp_l`	`__strcoll_l`	`__strfmon_l`
`__strncasecmp_l`	`__strtod_l`	`__strtof_l`
`__nl_langinfo_l`	`__strtold_l`	`__strtoll_l`
`__strtoul_l`	`__strtoull_l`	`__strxfrm_l`
`__toascii_l`	`__tolower_l`	`__toupper_l`
`__towctrans_l`	`__towlower_l`	`__towupper_l`
`__wcscasecmp_l`	`__wcscoll_l`	`__wcstod_l`
`__wcsncasecmp_l`	`__wcstof_l`	`__wcstol_l`
`__wcstold_l`	`__wcstoll_l`	`__wcstoul_l`
`__wcstoull_l`	`__wcsxfrm_l`	`__wctrans_l`
`__wctype_l`		

While this list is already long it is not complete. There are a lot more functions which use the global locale. It would be possible to define new versions (even though sometimes the interface would be awkward, consider `printf`). Using the functions will be a problem since the users have to rewrite a lot of code.

Instead of going the route of adding even more functions a completely new approach is used. It is backward compatible and requires almost no changes to the user code. The idea is to replace the concept of the process-global locale with that of a thread-local locale. A new function is introduced to select the thread-local locale:

```
  locale_t uselocale (locale_t newloc)
```

Yes, no leading underscores. Together with these new interfaces the already mentioned types and interfaces will be declared official and the leading underscores will be removed.

`uselocale` takes a locale object which was previously created using `newlocale` or `duplocale`. The information from this locale object will be used by all functions in the C library which implicitly use locale information. Note that this does *not* include the `*_l` interfaces mentioned above.

Before `uselocale` is used for the first time (or after it is called with the parameter `LC_GLOBAL_LOCALE`) the thread-local locale is the same as the global locale. The change of the locale with `setlocale` is visible. This maintains 100% compatibility for all threads which do not call `uselocale`.

For the user this new function means that no existing code has to be rewritten to take advantage of thread-local locales. For instance, existing code can be used to service multiple requests with different locales in separate threads. Even single-threaded applications can benefit. Since a call to `uselocale` is cheap (normally perhaps a dozen assembler instructions long) it is possible to often switch between different locales to service requests.

With the introduction of thread-local locales the locale information can finally again be relied on without making the code non-thread-safe. And it is also now possible to implement ISO C++ correctly.

2.2 Transliteration in `localedef`

The data used by the locale runtime functions is generated with a tool named `localedef`. The input for this file consists of a textual description of the locale which is independent of the character set used, and a textual representation of the character set for the locale. The `localedef` is a compiler which transforms the abstract locale description using the concrete character encoding into a format which can easily and efficiently be used at runtime.

```
int pthread_key_create (pthread_key_t *, void (*) (void *))
int pthread_key_delete (pthread_key_t)
int pthread_setspecific (pthread_key_t, const void *)
void *pthread_getspecific (pthread_key_t)
```

Figure 1: POSIX thread functions to handle thread-local variables

This is how locales were handled since the earliest days of glibc 2 even if there were some quite drastic internal changes. This does not mean however that there are no problems with this.

The biggest problem is writing the locale specification in a way which always provides the best results while keeping it usable with different character sets. These two goals cannot always be achieved at once.

A prominent example in recent times is the handling of the Euro. The locale specification contains a field which specifies the national currency symbol. For countries participating in the Euro this means '€'. The locales for these countries before the Euro were predominantly using the ISO-8859-1 or ISO-8859-2 character sets. These character sets predate the Euro decision and do not contain this character. To support the world with the Euro new character sets were created, mainly ISO-8859-15. This character set could be used to generate, say, the German locale `de_DE` but this would break compatibility. Existing environments and programs assume that the `de_DE` locale is encoded using ISO-8859-1 and that `de_DE@euro` is the locale with Euro support using ISO-8859-15.

What has this to do with transliteration you ask? How does one write the source for the `de_DE` and `de_DE@euro` locales? It is obviously best if the same source is used for the generation of both locales. To generate the correct output for the `de_DE@euro` locale the field for the currency symbol should contain the Euro symbol. But this cannot work when using ISO-8859-1, can it?

This is where transliteration steps in. Transliteration allows to compensate for missing mappings of characters. This is not a new concept to glibc: the wide character to multi-byte mapping functions from ISO C are using transliteration since glibc 2.2 and the same mechanism is available in `iconv` through to use of the `//TRANSLIT` suffix to character set names. The information for the transliteration comes from the `LC_CTYPE` category of the currently selected locale which in turns is generated from the locale specification.

To support transliteration in `localedef` it was therefore "only" necessary to use the transliteration information from the locale specification twice. In our Euro example this would allow `localedef` to map the Euro character in the currency symbol string to `"EUR"`.

With this change the authors of locale specifications have much more room to optimize even if the locale is used with different character sets. All that is needed is a sufficiently complete list of transliterations which in any case if useful and necessary.

2.3 Thread-local Storage Support

Threads have always been an afterthought in C and C++. The languages were not defined with threads in mind and appeared before POSIX threads were standardized. The POSIX thread library which is used on Linux systems is not the worst design there is. It is reasonably simple to run multiple threads and with the help of some higher-level libraries the job is made even easier when C++ is used.

This is all for the handling of threads but there is more to multi-threading than this. The weakest point is the handling of thread-local variables. The POSIX standard defines four interfaces (see figure 1).

To use a thread-local variable the `pthread_key_create` function must be used to get a unique key for a thread-local variable which is valid during this program run. There is no guarantee that the same key is provided in other program runs and there is also the chance that the request for a key failed since the maximum amount of thread-local variables is already allocated.

Once a key is obtained it is possible to set and retrieve the thread-local value of this variable. Only values of type `void*` are allowed and calls to `pthread_setspecific` and `pthread_getspecific` resp. are necessary to set and retrieve the value. This call need not and usually is not very cheap.

For a user this mechanism is very cumbersome. Allocation and deallocation of the key has to be handled correctly. The actual use of the variable normally has two phases due to the possibly high cost of accessing the variable: in one phase the thread-local variable is accessed and the value is kept in some automatic variable (which by definition is thread-local). The regular work is then done using this copy. In practice this is not too much of a problem since the type of the thread-local variable is always `void*`.

This limitation on the type is itself probably the biggest problem. If anything other than a variable pointer has to be stored it is necessary to dynamically allocate memory and store the pointer to this memory block in the thread-local variable[1] For the user this is a pain and a good place to make mistakes. For the compiler it is something which it cannot do by itself. One situation where it would want to use thread-local storage is handle spilling. In compiler generated code register spills normally happen to places on the stack. In general the compiler cannot allocate new global variables and spill into them since these global variables are shared by all threads. Spilling on the stack has the problem that the stack size can be very limited. Therefore the compiler cannot spill arbitrary amounts of memory. Some optimizations (such as pipelining) do need larger amount of memory for which thread-local variables would have to be allocated. But this isn't possible since the compiler cannot generate calls to the POSIX thread functions.

To solve the problem the ELF ABIs in use on Linux got extended. In addition to the normal variable sections like `.data` and `.bss` there will be new sections `.tdata` and `.tbss`. These new section contain the initialized and uninitialized thread-local variables the code is using. Every object (application, DSOs) can have its own set of these sections. To define a variable for these new sections the new keyword `__thread` is proposed:

```
__thread int foo = 42;
```

This defines a thread-local variable `foo` which is initialized with 42. The type of the variable is not fixed; any valid C type can be used making this approach much easier to use than the POSIX thread-local variable handling. The modification of a thread-local variable in one thread is not visible in another thread. For every new thread the variable is initialized with the value 42. The value is *not* inherited from the thread with created the new thread.

The compiler side of this extension is not yet reality, as the `__thread` keyword is not yet implemented. Today it is possible to imitate the use of thread-local variables using a few `asm` statements. The GNU C library sources contain some example code.

The other necessary changes, in the linker and in the runtime environment, are in place. In the C library the affected parts are the dynamic linker which has to set up the thread-local storage for the initial thread and the thread library which has to allocate, initialize, and deallocate the thread-local storage memory for each thread.

All this happens completely transparent to the user. Both the linker and the runtime environment perform quite a few optimizations which make the user of thread-local variable efficient, and writing code one does not have to worry about thread-local variables which are only used in a few threads. The allocation is optimized for this. There is no reason not to use the ELF TLS support.

[1]OK, it is possible to cast integers to pointers but this is no portable C code.

2.4 `GLIBC_PRIVATE` Version Name

The single biggest problem of providing backward compatibility for new versions of the C library is the use of internal interfaces. Although constantly reminded that this is not allowed and even though no prototypes and declarations for the internal objects is given people keep using them. The fact that the sources for glibc are available seems to make it possible to rely on the information from the sources.

What has to be realized is that internal interfaces are just that: *internal*. They are not meant to be used by user programs since the compatibility guarantees do not extend to them. Internal interfaces can go away if they are no longer needed or the semantics are changed. All these kind of changes are disastrous and it is unfortunately possible to observe the effects after every major release of glibc.

Programmers until now used the excuse that it is not easy to recognize internal interfaces. This is of course not true since the names of all internal interfaces begin with an underscore character and only very few, well-documented, and supported interfaces also have names beginning with an underscore. It is true, though, that recognizing internal interfaces programmatically needs some effort.

A change in glibc 2.3 provides a solution for this. All internal interfaces are renamed. The version name, which is part of the interface name, is changed to `GLIBC_PRIVATE`. This change will definitely break all programs using internal interfaces. But this breakage is for a good reason. Once glibc 2.3 is used in program development (well, for linking) it is very easy to determine whether an application or DSO uses an internal interface:

```
$ objdump -T program|grep GLIBC_PRIVATE
```

If this command is producing any output *`program`* is violating the rules. Sun Microsystems has developed a tool (ABIcheck) which performs a test like this. Future versions of RPM (Red Hat Package Manager) will also help to iron out the use of internal symbols. When trying to package an offending binary, RPM will notice the version name during the step in which it determines version dependencies and will make the packaging fail. In case somebody works around this (or uses an old version of RPM which does not implement the extra checks) there is a mechanism to prevent installation of such a package. A correctly packaged glibc binary will avoid advertising the availability of the `GLIBC_PRIVATE` version which means that RPM cannot resolve a requirement for this symbol for new packages and will therefore refuse to install them. Note that the glibc package "lies" about the version not being available. This all means that users have to go to great length to violate the rules if RPM is used throughout the development and program deployment.

2.5 New `regex`

Besides the thread library implementation the biggest obstacle to compliance with the POSIX standard was the regular expression matcher implementation. The implementation in glibc 2.2 is very old and nobody really understands it anymore. One of the last extensions was the addition of support for multi-byte characters which finally made the code unmaintainable. In addition the implementation fails a number of tests in the test suite and has some restrictive limitation which are reported as bugs.

glibc 2.3 will feature a complete rewrite, contributed by Isamu Hasegawa from IBM Japan. The new implementation has no ties whatsoever to the old implementation except that it tries to provide the same interface (i.e., the extensions over POSIX the old implementation has are still supported). The new implementation had basic POSIX compliance as the primary goal. The extensions did not influence the design which helped to keep down the complexity.

Something which did influence the design was the multibyte character support. Instead of adding it afterward as it happened with the old implementation good support for internationalization is built-in from the beginning. Another design decision was to use an optimization which currently is performed outside `regex`: the use of deterministic finite automata (DFA). A large part of the expressions which people use `regex` for do not need all the expressive power of regular expressions. In these cases a DFA is sufficient. It is easy to recognize expressions which fall into this category. The benefit of making this distinction is that the handling of DFAs is *a lot* cheaper. Packages like GNU grep performed this optimization themselves for years since the old `regex` implementation did not do it automatically.

The new implementation is believed to be fully compliant with POSIX. All required features, including full support for `[: :]` (character classes), `[. .]` (collating symbols), and `[= =]` (equivalent classes), are available. This is believed to be the first implementation which has all the pieces in `regex` and the system's locale implementation.

While performance of the implementation is already good there is certainly room for improvements. The code which will be released with glibc 2.3 will definitely change in future. Optimizations which are (currently) not included are those which optimize the expression before they are compiled in `regcomp`.

The work on `regex` never really has to stop. There are countless possibilities for optimizations and if the state of the old implementation was the deterrent for work in this area the reason is gone now.

2.6 Miscellaneous New Features

A usual a new major release features a plethora of minor additions. The set of `iconv` modules is still growing. The list of newly support character sets includes IBM1163, IBM1164, EUC-JISX, and SHIFT_JISX0213. At the same time the existing modules are extended to handle Unicode 3.2 which once again introduces a number of new characters. This revision of Unicode is especially important for East-Asian languages since the mapping from character sets used in that area and Unicode was improved.

3 Optimizations

Now that the functionality of the C library is almost complete work on optimizations can start. Earlier versions were also optimized to some extent. True to good software engineering practices premature optimizations on a larger scale were avoided, though.

The following sections describe three kinds of optimizations: optimizations in the runtime (dynamic linking process), in the generation of the DSOs, and in interfaces of the library. The third kind is good to know since it might help making good programming decisions. The first kind is very noticeable since every program benefits. The second kind is of interest to everybody who writes code which consists of DSOs.

3.1 Pre-Linking

Regardless of how big a program is there always is a minimum price to pay for the execution: the program has to be loaded and the runtime environment has to be initialized. Large programs might need many different DSOs and these have to be located and prepared for use. The bare minimum set of DSOs consist of the dynamic linker and the C library.

The dynamic linker was optimized significantly for the 2.2 release of glibc. The time spent in the dynamic linker before the transfer of control to the user code for the same application was reduced by 20% or more since glibc 2.1, but there is only that much what can be done here. The majority of time is spent in either the kernel (for loading the code) or for relocating the DSOs. Relocation is the process by which the dynamic linker prepares the application and the DSOs for its load position and the load position of the DSOs they depend on.

The work done in the kernel is already cut to the minimum. The different segments have to be mapped appropriately which requires normally three `mmap` system calls in addition to one `mprotect`, `fstat`, and `read` system call. Some operating systems implement a system call which perform all this work in one but it is highly questionable whether the gain is noticeable enough to justify the added complexity.

The only area with possibility for improvements left is the relocation process. The glibc 2.2 implementation performed as good as possible with the ELF files generated by the available tools. All DSOs and the dependencies on them require relocation since the load address of the DSO is not known in advance. The possibility to place DSOs at arbitrary addresses in the address space is the major advantage ELF DSOs have over a.out shared libraries which have a fixed load address. To save on the relocation processing it would be necessary to introduce preferred load addresses for the DSOs and pre-linking them for these addresses. The important part is that the load address is only *preferred* and not required. If for some reason the address space the DSO would require when loaded at the preferred address is already used, it must be possible to load the DSO at another address. This way we get the best of both worlds: the speed of the old a.out shared libraries due to skipping the relocation and the flexibility of ELF DSOs since the load address is only a hint.

To implement the pre-linking a great deal of work and planning is necessary. On machines with 32-bit address space it is not possible to pick for each DSO on the system a separate load address. From the theoretical 4GB of address space only a fraction can be used for mapping DSOs and this should be happening in a way which does not hinder the allocation of large amounts of dynamic memory. Therefore the allocation of preferred load addresses should look at the actual use of the DSOs and determine which of them are not used together in a program and therefore can have the same preferred load address. This operation (and the others to prepare a program or DSO for pre-linking) are performed by a program which looks at all files on the systems and works on all applications and DSOs. The code is available at ftp://people.redhat.com/jakub/prelink/.

The second problem to solve is how to correct the prelinking in case the preferred load address is not available. The solution for this requires knowledge of the internals of the ELF format which cannot be provided here in detail. The short answer is that for architectures which are using RELA relocations in binaries nothing special has to be done. All the information is contained in the relocation records. Architectures which use REL relocations are missing information which is needed in some situations. For them the tool, which performs the pre-linking, replaces the relocation section with one which is using the RELA format. This increases the code slightly size but is not avoidable and the relocation sections are read-only which means the physical memory can therefore be shared by all processes.

The third problem is how to preserve the ELF semantics of symbols being looked up in scope order. This is the biggest problem of all. This again requires intimate knowledge of ELF internals but it is worth at least some explaining since the issue is very important. All loaded DSOs (except those loaded using `dlopen` without `RTLD_GLOBAL`) and the main application are put in a list which is used to search for any symbol. The first definition seen while traversing the list is used. This way definitions later in the list are normally not seen at all. The consequence for the problem at hand is that the name resolution is always relative to the actual application. I.e., if a DSO needs a symbol which it defines itself the `prelink` tool will match the dependency with the self-definition. When used in a program a definition from another object might be used if that object is found earlier in the lookup path. The solution consists of an addition to the ELF format: conflict lists. These conflict lists are added to applications only (since they define the lookup scopes). Each entry in the conflict list describes one relocations against a symbol from any of the loaded DSOs which requires special care because the symbol lookup differs from what `prelink` determined when handling the DSO. These lists are normally short and much faster to handle than the complete relocation of the application and the DSO. The programmer can avoid creating conflicts by writing DSOs correctly. Some of the techniques which help doing this are described in the next section.

Those familiar with the ELF symbol resolution might now notice one more problem: pre-loading. Do not use `LD_PRELOAD` or `/etc/ld.so.preload`! Pre-loading introduces the same problems which required conflict list to correct. This time no such conflict list is available since the problems are created at runtime. Therefore pre-loading will disable pre-linking altogether. This normally should not be a problem since pre-loading never was advertised as a solution for any problem but instead only as a hack to temporarily work around problems or to help debugging. Unfortunately not everybody listened to this and some programs used pre-loading as part of their normal mode of operation. These programs will continue to work but they will not experience any speed-up.

To summarize, pre-linking saves time at startup by replacing the relocation of the application and all DSOs with the processing of the conflict list. Not only is this less work, it also means that potentially a lot more memory pages can be left unchanged and don't have to be copied-on-write because of relocations.

3.2 Writing Position-Independent Code

Other work related to speeding up startup but also to speed up normal operation include changing the code to be more position-independent. Many optimizations in this area were made in the previous revisions (such as converting arrays of string pointers to simple strings). For glibc 2.3 a couple of important additional optimizations were made.

```
static int local;
extern int in_dso __attribute__ ((visibility ("hidden")));
extern int elsewhere;
int add (void) {
  return local + in_dso + elsewhere;
}
```

Figure 2: Variable Visibility Example

Some of them were just made possible by improvements in the tools. All of them are generally applicable to all DSOs so that spending some time on explaining them is well spent.

The costs associated with using ELF fall into two categories: startup costs, as already mentioned in the last section, and runtime costs. The use of pre-linking helps to reduce the costs of the relocations which have to be performed but is of course better to avoid relocations in the first place. Along with eliminating relocations we get the benefit of improving the generated code. This stems mainly from the fact that all modifications of the code segment at runtime must be avoided to enable sharing the code with other processes.

To avoid relocation it is necessary to know what is causing them. Almost all code, when compiled, creates relocations but we are here only interested in those which are carried forward into the DSO. All the others are handled by the static linker and therefore have no runtime penalty. We will now go into the details how all this affects variables. Functions are handled in the next section.[2]

When generating code the compiler has to generate code to cope with any possible situation the generated code might be placed in. The general code can deal with situations where the location of a variable is not known until runtime. In this case it could be found in a different DSO or the application. Therefore it is necessary to use indirect addressing: the dynamic linker determines the address and stores it in the data segment from where the compiler-generated code loads it and finally dereferences it. The use of direct addressing would require to change the code segment which is out of question.

The problem with generating code optimal for the situation is that the compiler is not the only party involved. The compiler may have to count in the possibility that a variable is defined in some other DSO, the linker knows the truth. Via command line options or more likely via version maps the user can instruct the linker to not export certain symbols. This also means that the reference for a symbol which is defined in the same object always resolves locally. In this case the generic runtime relocation to lookup a named object is transformed into a relative relocation.

In case the variable is known to be in the same DSO as the code referencing it, the situation gets simpler. At link-time the offset of the variable from any point in the data segment is known. The global offset table, pointed to by what is generally called the “PIC register”, is also in the data segment. From this follows that with the help of the PIC register the compiler is able to generate code which directly accesses the variable in question. This code is not only simpler since it can compute the address directly (and not via a memory load), the data segment also gets smaller since the memory the dynamic linker stores the computed address in can be saved.

We can learn two things from this. The first is something which should already be known to everybody: if possible, all non-automatic variables should be defined with file-locale scope. I.e., in C programs they should be defined with `static`. This will tell the compiler that the variable is in the same object as the code using it which allows it to generate the simplified code to directly access the variable using the PIC register.

The second thing learned is that the standard C language is missing the power to express the fact that a variable is not declared as `static` but still can be assumed to be local to the object referencing it. A global variable can either be `static` or not. The GNU C compiler knows, beginning with version 3.1 but also in later 2.96 versions, the `visibility` variable attributes. These attributes are derived from the visibility classes for symbols as defined in the generic ELF ABI. A concrete example should make this clearer.

The code in figure 2 features one variable of each of the three classes mentioned above. It is assumed that this code is part of a DSO. `local` is a variable declared with file scope and is therefore addressed using the PIC register. The variable `elsewhere` is declared as a global variable. Nothing is known about the possible position in the final program so the compiler has to assume the worst. An entry in the data segment has to be allocated and the dynamic linker has to be instructed to determine the address at program start time. The most interesting and new case is the variable `in_dso`. It is declared with the visibility attribute `hidden`. This is translated to the assembler pseudo-op `.hidden` for the definition which will tell the linker to not export the symbol. Since this attribute is also visible to the compiler it can avoid gener-

[2]Some architecture, mainly embedded ones, limit the range of jump instructions enough to make it impossible for the compiler to generate code using these instructions. In these cases functions are handled like variables and what is said in this section applies.

ation the generic code and can instead generate the same code as for the access to `local`.

What should be learned from this section is that the compiler should be given as much information as possible so that it can generate the best possible code. This requires some efforts on the programmer's side but they are worth it. The next section provides another example for this rule.

3.3 Avoid Calling Exported Functions

The final optimization for the ELF handling in glibc deals with function calls. The problem for them is different from that of variables. The code the compiler has to generate is the same regardless of whether the called function is in the same object or not.[3] The linker is responsible for generating the needed code. If the function called is possibly not in the same object a procedure linkage table (PLT) entry has to be created. This code also take care of loading the new PIC register value if the calling conventions of the architecture require the caller to do this.

If the called function is in the same object the jump can normally be performed using PC-relative addressing. Almost all processors support such an addressing mode. The range of such a jump instruction might be limited (especially on embedded technology processors, see the footnote in the previous section). In this case the compiler might have to generate more generic code. In any case there will be no relocations to be handled at runtime since the linker has all the information it needs.

We see now the difference between handling variables and functions. To handle variables optimally the compiler needs more information than can be expressed in standard C. For functions the generated code is always the same and the optimizations are performed in the linker.

But there is something the programmer can do to generate better code. This is nothing the linker can do on its own since the semantics of the code changes. As with the variable handling the optimization is to avoid using runtime relocations to lookup symbols which are referenced in the same object they are defined in. If the symbol in question is exported from the DSO the dynamic linker looks for a definition along the lookup path defined for the application. There can be more than one definition and it is not clear that at link-time which is used.

This can be changed if the programmer is sure that it is always the local definition that is wanted. This is the case more often than not. What has to happen is that the definition which is used is not exported. The originally used symbol still is exported (because it is part of the DSO's ABI) but a newly created alias need not. This is especially true for the C library. Functions like `memcpy` are used internal but they also must be exported. There is no reason to not use the internal definition since the semantics of the function must always be the same.

[3]This is at least true for the supported architectures.

Along these lines glibc 2.3 contains a large amount of changes which cause the libc implementation to avoid using exported interfaces. Not all functions are treated this way: `malloc` and friends are often intentionally overloaded. For the rest, the changes made consist of the addition of another alias for the function and all callers are changed to use this alias. The name is normally constructed by adding the suffix `_internal`. These aliases are then declared with the visibility attribute `hidden` and are *not* mentioned in the version scripts. This causes the functions not to be exported and therefore the linker can construct the necessary code which does not require runtime relocations. Note that the `hidden` attribute is only of importance for platforms which treat functions like variables.

The result of this optimization is that

- no PLT entry is created for the function,
- the dynamic linker does not have to handle the PLT entry at startup, and
- that direct jumps are used instead of the indirect jumps caused by using the PLT (which can mean quite big gains on architectures with deep pipelines).

3.4 Object File Reordering

Object file reordering is one of the features which will probably not be enabled by default since the results are mixed. Support will be there however and users are encouraged to experiment with it so it should be introduced here.

The problem to solve is CPU cache locality. Modern processors have a cache hierarchy. Normally code that is executed has to be transfered to the cache, mostly even to the first-level cache. Transfer from the main memory to the highest cache level is the most expensive operation and should be avoided if possible. This can be achieved by not using too many cache lines and memory pages during execution.

The compiler takes care of following these rules when generating code for functions. Optimized code is cache line-aligned, infrequently used basic blocks are moved out of the main stream of operation. Seldomly a function is self-contained, though: it calls other functions and library interfaces.

These inter-function dependencies are what object file reordering is trying to address. Functions, which are often used together should be grouped in a way which allows the most efficient use of virtual memory and CPU caches. This kind of relationship can only be determined reliably by observing program runs. A static analysis of dependencies does not provide a clear image since many function uses will be infrequent (such as a call to `fprintf` in case of an error).

Note that we leave "grouped" intentionally vague here. It can mean a lot of different things. It's not necessarily the case that the code for the function is as close as possible together in memory. This is not how virtual memory systems and CPU caches work. Often it is important to look at cache associativity to determine whether the called function is throwing the cache lines for the calling function out of the cache. These are all details which might become important at some point. At present there is no support for this.

In the explanation we mentioned so far functions and their dependencies. This makes a difference since we cannot reorder individual functions since glibc is not built with gcc's `-ffunction-sections` option. This isn't necessary anyway since the glibc sources pretty strictly follow the one-function-per-file rule. If there is more than one function in one file they are closely related and normally cannot be used without one another. Reordering happens on object file basis.

Without reordering the layout of the DSO is determined by the order in which the object files are added to the `libc_pic.a` archive. This in turn is determined by the order in which make visits the various subdirectories of the glibc sources and below that level, by the order the various files are mentioned in the Makefiles. The resulting order is mostly random. The reordering phase steps in after the `libc_pic.a` file is created and reorders the file in the archive. The reordered archive is then used to construct the DSO; this process accepts the input in the order the object files are in the archive.

Initially the reordering instructions are simple. They just mention files which are to be placed together. How this is determined is left to the user. A default ordering file will be provided but the user can replace it. Later more sophisticated methods will be implemented. It'll be necessary to look at the cache lines, at cache associativity. The reordering process will require adding dummy files which align the next file, e.g., to avoid crossing cache lines. These optimizations will be very CPU-specific and will require a great deal of experimentation. Participation is welcome.

3.5 Stream I/O with `mmap`

The `FILE` stream interface is often used out of convenience. The programmer does not have to worry about not making too many file operations since the implementation tries to optimize this by using buffers and filling them in larger chunks. The file operations work on these buffers and request refilling them if the end if reached. This means quite a few tests of boundary conditions in addition to the copying.

People looking for the highest performance will therefore prefer to not use the `FILE` interface since the general-purpose buffering might introduce to higher than necessary costs. They rather invest in code which does the reading or writing itself, perhaps with the use of `mmap`.

The status quo can be changed, though. There is no reason for the `FILE` implementation to not be optimized to use `mmap` itself. It is only a matter of complexity of the implementation. For streams which are writable complex relations between using the file descriptor, flushing the stream, and reading/writing the stream make the use of `mmap` quite costly. Maybe too costly for the gains. For a subset of the problem such an implementation is feasible, though.

glibc 2.3 will introduce code which avoids using buffers and reading a file in chunks by `mmap`ing the whole file if the `FILE` stream is read-only. The benefits should be obvious. First, no buffer management is necessary. The functions which are handling buffer underflow are simple no-ops. They always signal end-of-file. The actual buffer operations, such as `fgetc`, do not require any call to an expensive function unless the end of the file is reached. The whole code (if the `_unlocked` variants are used) is inlined and therefore very fast.

The second benefit is that the file content does not have to be copied around. The kernel provides access to the file content using the `mmap`ed buffers and the stream implementation directly uses these buffers. The file content is loaded when it is actually used, i.e., read. Jumping between different locations in the file is a simple pointer operation.

For 32-bit architectures we might run into some problems, though. The 32-bit address space is not really large, especially when the reserved bits are taken out. Therefore the implementation restricts the `mmap` method to files up to a certain sizes. Larger files are handled as before. On 64-bit architectures such a limitation does not seem to be necessary.

The results of this operation are quite noticeable. Initial test showed an improvement in the file operations of 10%. Since now the file operations are almost as fast as they can be (they are mostly only pointer operations) it should be not necessary anymore to write specialized code to read files. The read-only stream operation should operate now almost at the highest speed made possible by the kernel.

3.6 Reduce Number Of Locale Files

The locale implementation in glibc 2.2 features locales with 12 categories. Every loaded locale requires 12 files to be mapped or loaded into memory. This represents a significant number of kernel operations and takes an noticeable amount of time. The loading of the locale data requires between 5% and 7% of the total startup time.

To avoid these costs glibc 2.3 introduces a locale archive. This is a single file which contains all the locale data the system has. It is still possible to have separate locale files in appropriate directories but this source is only secondary, it will looked for only if a locale cannot be found in the archive.

The benefit of the archive is that only one file has to be mapped into memory. It does not have to be mapped in total, only the necessary bits need to be available. Since the data for several of the categories is very small it is possible to have all the data available by mapping only a single page of memory. So in total we save eleven calls to `open` and `close` and several `mmap` calls. The latter also results in a reduced used of virtual memory for locale data.

The locale archives are created using `localedef`. Instead of creating the individual files the program is modified to be able to write to the locale archive directly. It is also possible to add already generated locale data to the archive. While doing this identical copies of data locale files are recognized. This happens quite frequently for some of the locale categories. Recognition is made easy by keeping checksums of all the files in the archive.

The modification of the archive is a pretty complicated issue. Since it might be and probably is in use while being modified or extended changing existing data is not allowed. Also, the administrative data structures must be laid out in a way which allow efficient use at runtime. Fast runtime access means that the administrative data structures should be mapped into memory using `mmap`. This in turn requires that the data structures do not change in size. `localedef` therefore allocates all data structures larger than necessary in the moment. This resizing requires creating a completely new file to not disturb programs which use the current version of the locale archive. The resizing is automatically taken care of but if one regenerated a lot of locales it is important to regenerate all of the locale archive from scratch every once in a while. This way all the accumulated garbage in the archive gets disposed off.

This optimization, while speeding-up the program start, is becoming more important with the thread-local locale model mentioned before since the mapped files do also consume file descriptors, lowering the number that is available to the program.

3.7 New `malloc`

The `malloc` implementation in glibc 2.2 was already good. It features good multi-threaded performance due to the use of multiple arenas which can be used simultaneously by different threads. The implementation had some problems, though, when it comes to fragmentation and to the handling of boundary cases.

The new implementation in glibc 2.3 comes from the same source. It is based on Doug Lea's latest creation and once again was modified by Wolfram Gloger for glibc. It again features arenas for the use in different threads. Locking is now in parts done using spinlocks instead of the heavier POSIX mutexes. The details about the implementation can be learned from the source code which has extensive documentation.

Overall the new implementation should have about the same speed as the old implementation. But it features a much better handling of fragmentation. Even after a large number of allocations and de-allocations the memory consumption should not increase unduly because memory blocks with odd sizes are requested. Applications like Mozilla will show a much reduced total memory consumption.

Drastic examples of improvements are programs which free a lot of the memory which was allocated. It was possible (and not too uncommon) to find code spending an unproportional amount of time in calls to `free`. The allocation itself is hardly measurable. The new implementation does away with this. In one specific test case the speed improvement exceeded 10,000%.

There have been some slight performance decreases been measured for normal operations but these are really insignificant. Given the avoidance of extremely bad performing cases and especially the reduced fragmentation this is a worthwhile trade-off.

References

Thread-aware Locale Model, Ulrich Drepper, Red Hat, Inc., 2001,
`http://people.redhat.com/drepper/tllocale.ps.bz2`

ELF Handling For Thread-Local Storage, Ulrich Drepper, Red Hat, Inc., 2002,
`http://people.redhat.com/drepper/tls.pdf`

`malloc.c` *sources in glibc*, Wolfram Gloger and Doug Lea, glibc CVS archive at `http://sources.redhat.com`

ISO DTR 14652, Proposed transliteration support for POSIX locales,
`http://std.dkuug.dk/jtc1/sc22/wg20/docs/n822-dtr14652.pdf`

Sun Microsystem's ABI Checker,
`http://abicheck.sourceforge.net/`