Thread-aware Locale Model

A Proposal

Ulrich Drepper
Red Hat, Inc.

1325 Chesapeake Terrace
Sunnyvale
California
94089
drepper@redhat.com

The locale model currently used on POSIX (and especially Unix) systems in the C libraries was designed in a time when threads were not used at all or at least were just appearing. Therefore the interface to the locale information was designed like many other interfaces in the C and Unix library: with process-global access.

This is a big problem nowadays when threads are widely deployed but we still live on with the global POSIX locales. For other programming languages (C++, Java) the language (or better: library) designers thought about this problem in advance and standardized a locale model which can be used in multi-threaded applications.

To finally catch up with these programming languages this paper proposes an extension of the C library to enable the use of locales in multi-threaded applications and for implementations of the locale models in C++. The goal was to require as little change in existing applications as possible to use the new features.

1. Introduction

Assume writing a server application which is supposed to handle cultural differences of the user on the other end of the connection by using the information from locales. The POSIX locale model of today restricts the implementation to have exactly one thread handling the services since the locale information in process-global. The only alternative is to protect the use of all locale-sensitive information by locks which defeats the purpose of using multiple threads in the first place. There can be many processes but this might be more expensive if data has to be shared between the instances. Also, switching between different locales inside a process does not come without a cost.

There is another important aspect: the new locale model should allow implementing the C++ locale model and possibly others. Especially the C++ model sets the direction for the implementation: it must be object based to allow integrating it in the C++ locale classes. The locale objects also obviously solve the problem of thread-safety with appropriate additions to the API to make functions use those objects instead of the global locale.

This is a quite radical change in the C API which would require all programs to be rewritten. Ideally this wouldn't be necessary and existing code would be reusable. This requires that the existing functions will make use of the thread-local locale information if the new locale model is activated (which must not be the default).

In the following chapter we will show in a few steps how fulfilling the requirements above will lead to a simple new locale model with all the wanted properties.

2. Locale Model Evolution

In this section we will first introduce the POSIX locale model as it exists. Then we will change the implementation in two steps to get the result we want. We have to introduce a few fundamentals first, though.

A locale in the POSIX sense is a named entity which contains information about cultural differences in several categories. Data for at most one locale must be loaded for each category at any time, defaulting to a special locale (named C or POSIX locale) if none is explicitly loaded. This means that data from different locales can be present but one cannot use the same category from different locales at the same time. Different categories can use data from different named locales at the same time.

Figure 1. POSIX Locale Model

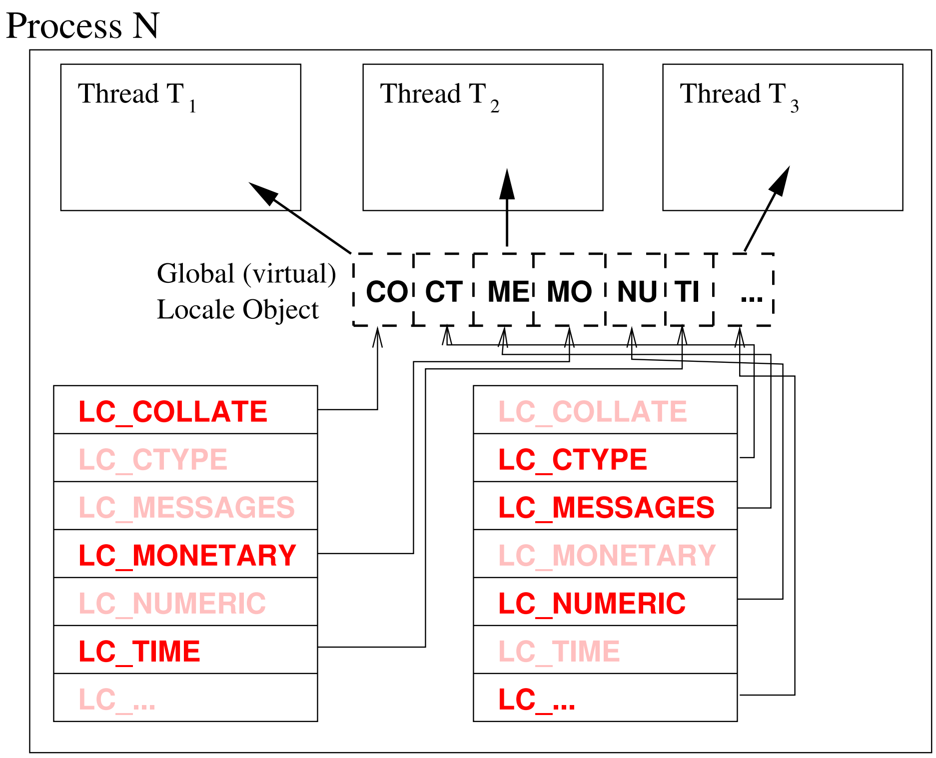

Figure 1 shows how multiple loaded locales make up one virtual locale object which consists of non-overlapping subsets of the two locales. The user cannot see anything like this virtual locale object. All he sees is a global locale of which he can control the individual categories using the `setlocale()` function. I.e., all he can do is load more locale data and rearrange the arrows into the global locale object. Nothing can be done about the relation of the threads to the locale object.

In a first step we will create real locale objects. Since these locale objects will have a real memory representation it means that there is no reason anymore to restrict the number to one. Therefore we arrive at something like the following:

Figure 2. Introduction of Locale Objects

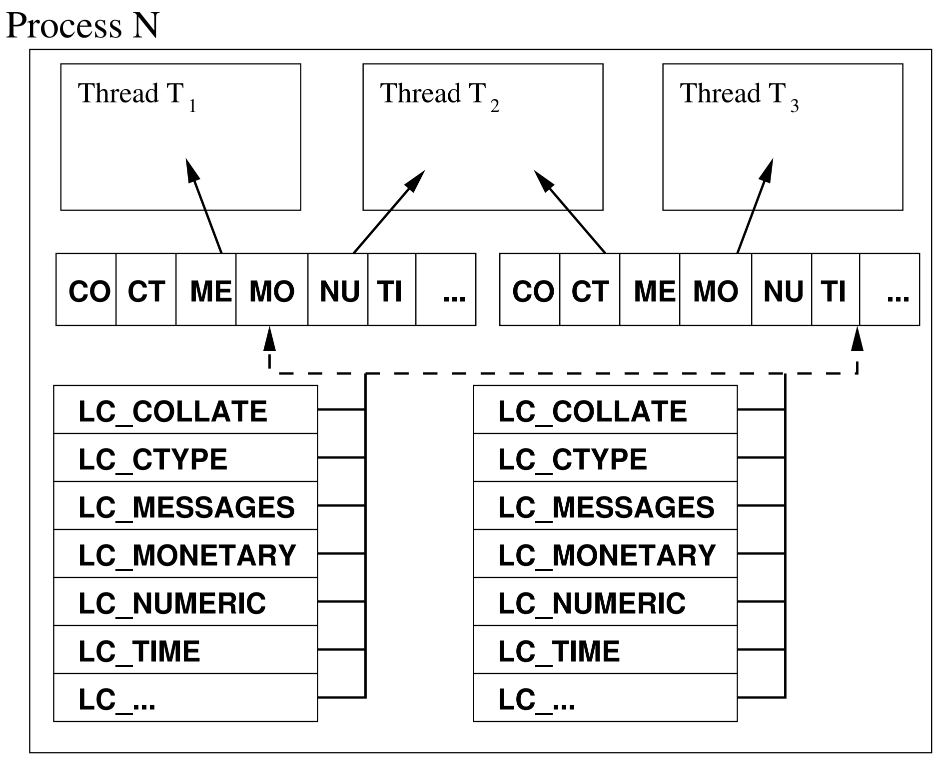

Figure 2 shows data from more than one locale loaded and making up the content of more than one locale object. For both, the locale data and the locale objects, the count is only limited by the implementation. Selecting the content of the different categories of the locale objects should be up to the user. There are no inherent limitations in the locale model which would require using only non-overlapping subsets. What is therefore needed is an interface to create locale objects from different named locale for the individual categories.

To make locale objects useful there must be APIs which can use them. All interfaces which are currently implicitly using the global locale are candidates. For those interfaces which should be used with locale objects a variant must be defined which takes another parameter which is a handle to the locale object. E.g., one candidate is the `strcoll()` function which is defined as

```
int strcoll (const char *, const char *)
```

Changing this interface to use a locale object is simple. With the conventions introduced later in this document it would look like this:

```
int strcoll_l (const char *, const char *, locale_t)
```

The function name is transformed using a consistent rule (the proposal is to consistently add a _l suffix) and at an appropriate place in the parameter list a parameter of type locale_t is added.

Implementing these variants of the standard interfaces can be very easy. If the global locale in the POSIX locale model is also represented using a locale object the standard functions *could* be implemented as simple wrappers around the new functions. In the case of `strcoll()` this would mean:

```
int
strcoll (const char *s1, const char *s2)
{
  return __strcoll_l (s1, s2, __global_locale_object);
}
```

The double-underscore characters are necessary for namespace reasons. The increase of the overall size of the libraries implementing is therefore potentially minimal.

As it turns out this new model is also sufficient to implement the C++ locale model. The requirements come from classes like collate_byname. The constructors for these classes take a string parameter specifying the system locale to be used. In a sense this is equivalent to setting the specified locale for a certain category.

The locale class in the C++ standard library allows to construct locale objects by replacing facets (which correspond to the categories in the POSIX model) in existing locale objects. An interface to do this with the locale model proposed here will therefore be necessary. Additional interfaces required by the C++ locale model are a function to duplicate a locale object (for copy constructors and similar uses) and a function to free all the resources allocated for a locale object. The next section will introduce these interface in detail.

We can conclude from this that the new thread model would fit nicely in the C++ locale model and no disruption of the existing code base would be caused since the interface is all based the C++ standards. The situation is quite different for C programs. The whole problem started out from the C standard not containing sufficient interfaces. Therefore something has to change.

When looking at the interfaces which would have to get equivalents with an _l suffix it quickly becomes clear this is not the ideal solution. Accessing the locale data is very pervasive in the C library. Most of the non-trivial functions access the locale in one way or the other. Since a function like `vfprintf()` must access the locale (e.g., for the thousands separator) this also means that every other function of this family (like `printf()`, `snprintf()`, etc.) does it as well. This would mean that every single application would have to be changed. The changes would effect source code compatibility since not all systems would provide an implementation of the new locale model (right away). Even worse is the situation for libraries which have to work in both situations, with and without the new locale support.

For these reasons it would be good to have a solution which does not have these problems. It should not be necessary to rewrite application and still take advantage of locale objects and their thread-safe use. The new interfaces still have to exist (at least for the C++ implementation) but they should not be the primary interface. Fortunately this is possible.

The actual underlying problem is that the locale is process-global and not thread-local.

If we now simply declare the locale as a property of the thread the problems with using locales in multi-threaded applications simply disappears. Each thread could have associated a locale object and all the functions of the C library use this thread-local data instead of the global locale. There would be almost no changes necessary to applications which want to use the new locale model.

Before declaring victory we have to line out a few more solutions for problems:

Backward-compatibility

> How can we assure backward compatibility with existing code? Code today is written with the concept of a global locale in mind. If we decide that all C library function use thread-local locales this can break existing applications.

Performance

> The standard was to access thread-local data is via the `pthread_getspecific()`/`pthread_setspecific()` interface. Using this interface is often quite slow. This would be a big problem for functions like `isdigit()`.

The compatibility problem can be easily solved. The thread-local locales used by default are simply the same as the global locale. Changing the global locale with `setlocale()` has the effect of changing also the thread-local locale for all threads using the default locale.

To change only the thread-local locale a new function has to be used (see `uselocale()` below). This guarantees that unless the program actually knows about the new locale functionality the behavior is identical to what ISO C and POSIX prescribe. Once the `uselocale()` is called the behavior is different but this is OK since `uselocale()` is a function of this new locale model.

The bigger problem is that of performance. An implementation of `isdigit()` with standardized functionality would have to look somewhat like this:

```
int
isdigit (int __c)
{
```

```
  locale_t *__loc =
    (locale_t *) __pthread_getspecific (__LOCALE_KEY);
  return __isdigit (__c, __loc);
}
```

This is a potentially very portable implementation (and should be kept in mind for platforms which allow no other implementation) but even if this code is transformed into a macro it is much slower than the regular `isdigit()` implementation which is simply an access of a global array. Some features of the GNU C compiler can help to reduce the penalties (`__pthread_getspecific()` can be declared const which allows hoisting the call out of loops) but this still is not competitive.

Fortunately ABI designers learned in the last years the importance of thread-local data and they designed into the ABI support for fast access of the same. Instead of using the POSIX ABI to get thread-local data information about it is kept in processor registers or nearby:

- On IA-32 machines the GS segment register is often used to access the thread descriptor and the included thread-local data.
- On SPARC machines the %g6 register points the the thread descriptor.
- On Alpha machines a PAL call (which is not much slower than a register access) is used to retrieve the pointer to the thread descriptor.
- On IA-64 machines the r13 register is reserved to serve as a pointer to the thread descriptor.

There are other architectures with similar solution which means that most of the important architectures already have mechanisms to provide fast access to the thread descriptor and therefore thread-local data.

The way the thread-local data part of the thread descriptor is organized is getting standardized. At least for systems using the ELF binary format. A proposed extension for the IA-64 ABI will be applicable as well for other architecture. The design will explicitly allow fast access to data like locale objects.

To illustrate the differences in the implementation of `isdigit()` we examine briefly the implementation of this function in the GNU C library. Ignoring the macro magic used the implementation would look like this:

```
#define isdigit(c) \
  (__ctype_b[(int) (c)] & _ISdigit)
```

This is pretty fast when executed since it a simple memory access with a masking operation. Rewritten to use the thread descriptors this could look like this (using GNU CC syntax and assuming the IA-32 architecure):

```
struct thread_descr
{
  ...
  const unsigned short int *__ctype_b;
  ...
};


#define isdigit(c) \
  ({ const unsigned short int *__ctype_b;                   \
     asm ("movl %%gs:%P2,%0"                                \
          : "=r" (__ctype_b)                                \
          : "i" (offsetof (struct thread_descr, __ctype_b)));\
     __ctype_b[(int) (c)] & _ISdigit; })
```

This implementation is not much (if at all) slower than the original code. Plus there is one less relocation necessary (no global symbol used). The versions of this code on most systems with support for thread registers will look similar and is similar in performance.

There is only one drawback of this approach: not all architectures have thread registers designed into their ABIs. How this can be solved has to be decided on a case by case basis. Possibilities are to use the above mentioned standard POSIX API or breaking the existing ABI and introduce a thread register.

3. API specification

The proposed changes to the API are in three areas:

- New functions to add basic locale object handling. Functions to create, modify, duplicate, and release objects must exist.
- Additions to existing interfaces to allow using locale objects.
- Support for the new, thread-aware locale model.

We will cover the new interfaces and changes to existing interfaces in the following three sections. But first here are the summarizing changes to header files.

3.1. Changed header files

<locale.h>

Name

`<locale.h>` — category macros

Description

The `<ctype.h>` header shall contain macros representing bitmask for each supported locale category.

LC_COLLATE_MASK
LC_CTYPE_MASK
LC_MESSAGES_MASK
LC_MONETARY_MASK

LC_NUMERIC_MASK
LC_TIME_MASK

In addition a macro to with the bits for all categories set shall be defined.

LC_ALL_MASK

The `<ctype.h>` header shall provide a definition for a type locale_t representing a locale object.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
locale_t newlocale (int, const char *, locale_t);
locale_t duplocale (locale_t);
void freelocale (locale_t);
locale_t uselocale (locale_t);
```

<ctype.h>

Name

`<ctype.h>` — character types

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
int isalnum_l(int, locale_t);
int isalpha_l(int, locale_t);
int isblank_l(int, locale_t);
int iscntrl_l(int, locale_t);
int isdigit_l(int, locale_t);
int isgraph_l(int, locale_t);
int islower_l(int, locale_t);
int isprint_l(int, locale_t);
int ispunct_l(int, locale_t);
int isspace_l(int, locale_t);
int isupper_l(int, locale_t);
int isxdigit_l(int, locale_t);
int tolower_l(int, locale_t);
int toupper_l(int, locale_t);
```

<wctype.h>

Name

`<wctype.h>` — wide-character classification and mapping utilities

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
int iswalnum_l(wint_t, locale_t);
int iswalpha_l(wint_t, locale_t);
int iswblank_l(wint_t, locale_t);
int iswcntrl_l (wint_t, locale_t);
int iswdigit_l (wint_t, locale_t);
int iswgraph_l (wint_t, locale_t);
int iswlower_l (wint_t, locale_t);
int iswprint_l (wint_t, locale_t);
int iswpunct_l (wint_t, locale_t);
int iswspace_l (wint_t, locale_t);
int iswupper_l (wint_t, locale_t);
int iswxdigit_l (wint_t, locale_t);
int iswctype_l (wint_t, wctype_t, locale_t);
wint_t towctrans_l(wint_t, wctrans_t, locale_t);
wint_t towlower(wint_t, locale_t);
wint_t towupper(wint_t, locale_t);
wctrans_t wctrans_l(const char *);
wctype_t wctype_l(const char *);
```

<strings.h>

Name

`<strings.h>` — string operations

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
int strcasecmp_l(const char *, const char *, locale_t);
int strncasecmp(const char *, const char *, size_t, locale_t);
```

<string.h>

Name

`<string.h>` — string operations

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
int strcoll_l(const char *, const char *, locale_t);
size_t strxfrm_l(char *restrict,
   const char *restrict, size_t, locale_t);
```

<wchar.h>

Name

`<wchar.h>` — wide-character types

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
int wcscasecmp_l(const wchar_t *, const wchar_t *, locale_t);
int wcsncasecmp_l(const wchar_t *, const wchar_t *, size_t,
   locale_t);
int wcscoll(const wchar_t *, const wchar_t *, locale_t);
size_t wcsxfrm(wchar_t *restrict,
   const wchar_t *restrict, size_t, locale_t);
```

<monetary.h>

Name

`<monetary.h>` — monetary types

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
ssize_t strfmon_l(char *restrict, size_t,
   locale_t, const char *restrict, ...);
```

<time.h>

Name

`<time.h>` — time types

Description

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
size_t strftime_l(char *restrict, size_t,
   locale_t, const char *restruct, struct tm *restrict);
```

3.2. Handling Locale Objects

In Section 2 we already explained briefly what determines the interfaces to handle locale objects. Basically we have to have the functionality of the `setlocale()` function and additionally have to be able to implement the operations necessary for a C++ locale object. This boils down to the following list:

- A function to create a locale object from a name. The name is that of the system locale supported by `setlocale()`. By designing the interface of the function cleverly, the same function can be used to modify existing locale objects.
- A function to duplicate an existing locale. This function will be used to implement the copy constructor of the C++ class and to create slightly modified versions of existing locale objects.
- A function to free the resource allocated for a locale object.

newlocale

Name

`newlocale` — create or modify a locale object

Synopsis

```
#include <locale.h>
locale_t newlocale(int category_mask, const char *locale,
locale_t base);
```

Description

The `newlocale()` function can create a new locale object or modify an existing one. If the *base* parameter is a null pointer a new object is created. Otherwise the object pointed to by *base* is modified.

What is added to the locale object is decided by the first two parameters. The first parameter specifies which category is set or modified. To do this the symbolic constants LC_CTYPE_MASK, LC_NUMERIC_MASK, LC_TIME_MASK, LC_COLLATE_MASK, LC_MONETARY_MASK, LC_MESSAGES_MASK are available, each corresponding to an equivalent LC_* symbol, and can be combined to a bitset containing one bit for each category.

> **Note:** We are using new symbols LC_*_MASK instead of using LC_* since the latter are not guaranteed to be usable directly in bitset. Instead they are often consecutive numbers.

> **Note:** If the system provides support for more categories there are equivalent LC_*_MASK symbols available as well.

> **Note:** Using int for the type of the first parameter restricts the number of supported locale categories on most systems to 32. This should not be a problem. If somebody envisions it is we better use a typedef so that those implementors can define a larger type (such as long long int).

For each category with the corresponding bit set in *category_mask* the data from the locale named by *locale* is used. In case an existing locale is modified the data from this locale replaces the existing allocation. If a completely new object is create the data for all sections *not* requested by *category_mask* is taken from the default locale.

If *base* is not a null pointer and the call of `newlocale()` succeeds the old locale handle passed in *base* is not valid anymore.

If the function call fails and the *base* parameter is not a null pointer the old handle passed in *base* remains valid and unchanged.

Return Value

If the function call succeeds the return value is a handle for a new locale object. It should be released by a call to `freelocale()` when it is not needed anymore. If the function call fails the return value is (locale_t) 0 and `errno` is set.

> **Note:** Should we introduce a symbol for (locale_t) 0?

Errors

The `newlocale()` function shall fail if:

ENOMEM

Not enough memory available to create the locale object or load the locale data.

EINVAL

The *category_mask* contains bit which do not correspond to a valid category.

EINVAL

The *locale* parameter is a null pointer.

> **Note:** Should we allow *locale* being a null pointer? When using `setlocale()` this returns the name of the locale which then can be fed back

to `setlocale()`. This is not necessary with `newlocale()` since the locale objects provide a direct handle to the currently selected set of locales.

ENOENT

For any of the categories mentioned in *`category_mask`* the locale data is not available.

Application Usage

Constructing a locale object from different locales is possible although one has to be very careful because the categories are not independent. Constructing a locale where the LC_CTYPE category data comes from a locale "loc1" and the LC_TIME category data from a locale "tok2" one can use code as follows:

```
locale_t loc, new_loc;

/* Get the "loc1" data.  */
loc = newlocale (LC_CTYPE_MASK, "loc1", NULL);
if (loc == (locale_t) 0)
  abort ();

/* Get the "loc2" data.  */
new_loc = newlocale (LC_TIME_MASK, "loc2", loc);
if (new_loc != (locale_t) 0)
  /* We don't abort if this fails.  In this case
     this simply used to unchanged locale object.  */
  loc = new_loc;
```

Every locale object allocated with `newlocale()` should be freed using `freelocale()`:

```
locale_t loc;
```

```
/* Get the locale.  */
loc = newlocale (LC_CTYPE_MASK | LC_TIME_MASK, "locname", NULL);

/* ... Use the locale object ... */

/* Free the locale object resources.  */
freelocale (loc);
```

freelocale

Name

`freelocale` — Free resources allocated for locale object

Synopsis

```
#include <locale.h>
void freelocale(locale_t locobj);
```

Description

The `freelocale()` function is used to free the resources allocated for a locale object returned by a call to `newlocale()` or `duplocale()`.

Return Values

None.

Errors

None.

duplocale

Name

`duplocale` — Duplicate locale object

Synopsis

```
#include <locale.h>
locale_t duplocale(locale_t locobj);
```

Description

The `duplocale()` function can be used to duplicate an existing locale object. A locale object can be used at any time in multiple places (except `newlocale()` calls) but if the lifetime can possibly end before all uses are finished one has to create a duplicate. Another reason to duplicate an object is if a slightly modified form is needed. This can be achieved by a call to `newlocale()` following the `duplocale()` call.

Return Value

If the function succeeds it returns a handle for a locale object identical to the one represented by the handle passed in *`locobj`*. If it fails the return value is (locale_t) 0.

Errors

The `newlocale()` function shall fail if:

ENOMEM

Not enough memory available to create the locale object or load the locale data.

The `newlocale()` function may fail if:

EINVAL

`locobj` is not a valid locale object handle.

Application Usage

To create a slightly altered version of an existing locale object code as follows can be used. The function takes a locale object and a locale name and it replaces the LC_TIME category data in the locale object with that from the named locale.

```
locale_t
with_changed_lc_time (locale_t obj, const char *name)
{
  locale_t retval = duplocale (obj);
  if (retval != (locale_t) 0)
    {
      locale_t changed = newlocale (LC_TIME_MASK, name, retval);
      if (changed == (locale_t) 0)
        /* An error occurred.  Free all allocated resources.  */
        freelocale (retval);

      retval = changed;
    }
  return retval;
}
```

3.3. Extended Interfaces

The second set of interfaces added include those which is derived from standard interfaces but they are using a passed in locale object instead of the global locale to handle localization. Not all interfaces are duplicated here, only those which are deemed necessary to implement the C++ library interface. As mentioned above, the normal C application is not expected to use these interfaces.

This does not mean we are 100% sure these interfaces are sufficient. If it proves that other interfaces are needed they can be added easily.

The proposed interfaces fall into three categories:

- character classification and mapping
- string functions
- direct locale data access

3.3.1. Extended Character Classification and Mapping Interfaces

The base forms of the following interfaces are the classification functions for single-byte and wide characters from the `<ctype.h>` and `<wctype.h>` headers. The extensions are also available from the corresponding headers.

isalnum_l

Name

`isalnum_l` — test for an alphanumeric character

Synopsis

```
#include <ctype.h>
int isalnum_l(int c, locale_t locale);
```

Description

The `isalnum_l()` function tests whether *c* is a character of class alpha or digit in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isalnum_l()` function shall return non-zero if *c* is an alphanumeric character; otherwise, it shall return 0.

Errors

The `isalnum_l()` function may fail if:

EINVAL

> *locale* is not a valid locale object handle.

isalpha_l

Name

`isalpha_l` — test for an alphabetic character

Synopsis

```
#include <ctype.h>
int isalpha_l(int c, locale_t locale);
```

Description

The `isalpha_l()` function shall test whether *c* is a character of class alpha in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isalpha_l()` function shall return non-zero if *c* is an alphabetic character; otherwise, it shall return 0.

Errors

The `isalpha_l()` function may fail if:

EINVAL

> *`locale`* is not a valid locale object handle.

isblank_l

Name

isblank_l — test for a blank character

Synopsis

```
#include <ctype.h>
int isblank_l(int c, locale_t locale);
```

Description

The isblank_l() function shall test whether *c* is a character of class blank in the locale represented by *locale*. In all cases *c* is a type int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The isblank_l() function shall return non-zero if *c* is a <blank> character; otherwise, it shall return 0.

Errors

The isblank_l() function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iscntrl_l

Name

`iscntrl_l` — test for a control character

Synopsis

```
#include <ctype.h>
int iscntrl_l(int c, locale_t locale);
```

Description

The `iscntrl_l()` function shall test whether *c* is a character of class cntrl in the locale represented by *locale*.

In all cases *c* is a type int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iscntrl_l()` function shall return non-zero if c is a control character; otherwise, it shall return 0.

Errors

The `iscntrl_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

isdigit_l

Name

isdigit_l — test for a decimal digit

Synopsis

```
#include <ctype.h>
int isdigit_l(int c, locale_t locale);
```

Description

The isdigit_l() function shall test whether *c* is a character of class digit in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The isdigit_l() function shall return non-zero if *c* is a decimal digit; otherwise, it shall return 0.

Errors

The `isdigit_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

isgraph_l

Name

`isgraph_l` — test for a visible character

Synopsis

```
#include <ctype.h>
int isgraph_l(int c, locale_t locale);
```

Description

The `isgraph_l()` function shall test whether *c* is a character of class graph in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isgraph_l()` function shall return non-zero if *c* is a character with visible representation; otherwise, it shall return 0.

Errors

The `isgraph_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

islower_l

Name

`islower_l` — test for a lowercase character

Synopsis

```
#include <ctype.h>
int islower_l(int c, locale_t locale);
```

Description

The `islower_l()` function shall test whether *c* is a character of class lower in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `islower_l()` function shall return non-zero if *c* is a lowercase character; otherwise, it shall return 0.

Errors

The `islower_l()` function may fail if:

EINVAL

> *`locale`* is not a valid locale object handle.

isprint_l

Name

`isprint_l` — test for a printing character

Synopsis

```
#include <ctype.h>
int isprint_l(int c, locale_t locale);
```

Description

The `isprint_l()` function shall test whether *c* is a character of class print in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isprint_l()` function shall return non-zero if *c* is a printing character; otherwise, it shall return 0.

Errors

The `isprint_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

ispunct_l

Name

`ispunct_l` — test for a punctuation character

Synopsis

```
#include <ctype.h>
int ispunct_l(int c, locale_t locale);
```

Description

The `ispunct_l()` function shall test whether *c* is a character of class punct in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `ispunct_l()` function shall return non-zero if *c* is a punctuation character; otherwise, it shall return 0.

Errors

The `ispunct_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

isspace_l

Name

`isspace_l` — test for a white-space character

Synopsis

```
#include <ctype.h>
int isspace_l(int c, locale_t locale);
```

Description

The `isspace_l()` function shall test whether *c* is a character of class space in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isspace_l()` function shall return non-zero if *c* is a white-space character; otherwise, it shall return 0.

Errors

The `isspace_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

isupper_l

Name

`isupper_l` — test for a uppercase character

Synopsis

```
#include <ctype.h>
int isupper_l(int c, locale_t locale);
```

Description

The `isupper_l()` function shall test whether *c* is a character of class upper in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isupper_l()` function shall return non-zero if *c* is an uppercase character; otherwise, it shall return 0.

Errors

The `isupper_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswalnum_l

Name

`iswalnum_l` — test for an alphanumeric wide-character code

Synopsis

```
#include <wctype.h>
int iswalnum_l(wint_t wc, locale_t locale);
```

Description

The `iswalnum_l()` function tests whether *wc* is a wide-character code representing a character of class alpha or digit in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswalnum_l()` function shall return non-zero if *wc* is an alphanumeric wide-character code; otherwise, it shall return 0.

Errors

The `iswalnum_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswalpha_l

Name

`iswalpha_l` — test for an alphabetic wide-character code

Synopsis

```
#include <wctype.h>
int iswalpha_l(wint_t wc, locale_t locale);
```

Description

The `iswalpha_l()` function tests whether *wc* is a wide-character code representing a character of class alpha in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswalpha_l()` function shall return non-zero if *wc* is an alphabetic wide-character code; otherwise, it shall return 0.

Errors

The `iswalpha_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswblank_l

Name

`iswblank_l` — test for an blank wide-character code

Synopsis

```
#include <wctype.h>
int iswblank_l(wint_t wc, locale_t locale);
```

Description

The `iswblank_l()` function tests whether *wc* is a wide-character code representing a character of class blank in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswblank_l()` function shall return non-zero if *wc* is a <blank> wide-character code; otherwise, it shall return 0.

Errors

The `iswblank_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswcntrl_l

Name

`iswcntrl_l` — test for a control wide-character code

Synopsis

```
#include <wctype.h>
int iswcntrl_l(wint_t wc, locale_t locale);
```

Description

The `iswcntrl_l()` function tests whether *wc* is a wide-character code representing a character of class cntrl in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswcntrl_l()` function shall return non-zero if *wc* is a control wide-character code; otherwise, it shall return 0.

Errors

The `iswcntrl_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswctype_l

Name

`iswctype_l` — test character for a specified class

Synopsis

```
#include <wctype.h>
int iswctype_l(wint_t wc, wctype_t charclass, locale_t locale);
```

Description

The `iswctype_l()` function shall determine whether the wide-character code *wc* has the character class *charclass*, returning true or false. The `iswctype_l()` function is defined on WEOF and wide-character codes corresponding to the valid character encodings in the locale represented by *locale*. If the *wc* argument is not in the domain of the function, the result is undefined. If the value of *charclass* is invalid (that is, not obtained by a call to `wctype_l()`) the result is unspecified.

Return Value

The `iswctype_l()` function shall return non-zero (true) if and only if *wc* has the property described by *charclass*. If *charclass* is 0, `iswctype_l()` shall return 0.

Errors

The `iswctype_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

Application Usage

The twelve strings "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", and "xdigit" are reserved for the standard character classes. In the table below, the functions in the left column are equivalent to the functions in the right column.

iswalnum_l(wc, loc)	iswctype_l(wc, wctype("alnum"), loc)
iswalpha_l(wc, loc)	iswctype_l(wc, wctype("alpha"), loc)
iswblank_l(wc, loc)	iswctype_l(wc, wctype("blank"), loc)
iswcntrl_l(wc, loc)	iswctype_l(wc, wctype("cntrl"), loc)
iswdigit_l(wc, loc)	iswctype_l(wc, wctype("digit"), loc)
iswgraph_l(wc, loc)	iswctype_l(wc, wctype("graph"), loc)
iswlower_l(wc, loc)	iswctype_l(wc, wctype("lower"), loc)
iswprint_l(wc, loc)	iswctype_l(wc, wctype("print"), loc)
iswpunct_l(wc, loc)	iswctype_l(wc, wctype("punct"), loc)
iswspace_l(wc, loc)	iswctype_l(wc, wctype("space"), loc)
iswupper_l(wc, loc)	iswctype_l(wc, wctype("upper"), loc)
iswxdigit_l(wc, loc)	iswctype_l(wc, wctype("xdigit"), loc)

iswdigit_l

Name

`iswdigit_l` — test for a digit wide-character code

Synopsis

```
#include <wctype.h>
int iswdigit_l(wint_t wc, locale_t locale);
```

Description

The `iswdigit_l()` function tests whether *wc* is a wide-character code representing a character of class digit in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswdigit_l()` function shall return non-zero if *wc* is a decimal digit wide-character code; otherwise, it shall return 0.

Errors

The `iswdigit_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswgraph_l

Name

iswgraph_l — test for a visible wide-character code

Synopsis

```
#include <wctype.h>
int iswgraph_l(wint_t wc, locale_t locale);
```

Description

The iswgraph_l() function tests whether *wc* is a wide-character code representing a character of class graph in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The iswgraph_l() function shall return non-zero if *wc* is a wide-character code with a visible representation; otherwise, it shall return 0.

Errors

The `iswdigit_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswlower_l

Name

`iswlower_l` — test for a lowercase letter wide-character code

Synopsis

```
#include <wctype.h>
int iswlower_l(wint_t wc, locale_t locale);
```

Description

The `iswlower_l()` function tests whether *wc* is a wide-character code representing a character of class lower in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswlower_l()` function shall return non-zero if *wc* is a lowercase letter wide-character code; otherwise, it shall return 0.

Errors

The `iswdigit_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswprint_l

Name

`iswprint_l` — test for a printing wide-character code

Synopsis

```
#include <wctype.h>
int iswprint_l(wint_t wc, locale_t locale);
```

Description

The `iswprint_l()` function tests whether *wc* is a wide-character code representing a character of class print in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswprint_l()` function shall return non-zero if *wc* is a printing wide-character code; otherwise, it shall return 0.

Errors

The `iswprint_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswpunct_l

Name

`iswpunct_l` — test for a punctuation wide-character code

Synopsis

```
#include <wctype.h>
int iswpunct_l(wint_t wc, locale_t locale);
```

Description

The `iswpunct_l()` function tests whether *wc* is a wide-character code representing a character of class punct in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswpunct_l()` function shall return non-zero if *wc* is a punctuation wide-character code; otherwise, it shall return 0.

Errors

The `iswpunct_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswspace_l

Name

`iswspace_l` — test for a white-space wide-character code

Synopsis

```
#include <wctype.h>
int iswspace_l(wint_t wc, locale_t locale);
```

Description

The `iswspace_l()` function tests whether *wc* is a wide-character code representing a character of class punct in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswspace_l()` function shall return non-zero if *wc* is a white-space wide-character code; otherwise, it shall return 0.

Errors

The `iswspace_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswupper_l

Name

`iswupper_l` — test for a uppercase letter wide-character code

Synopsis

```
#include <wctype.h>
int iswupper_l(wint_t wc, locale_t locale);
```

Description

The `iswupper_l()` function tests whether *wc* is a wide-character code representing a character of class upper in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswupper_l()` function shall return non-zero if *wc* is a uppercase letter wide-character code; otherwise, it shall return 0.

Errors

The `iswupper_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

iswxdigit_l

Name

`iswxdigit_l` — test for a hexadecimal digit wide-character code

Synopsis

```
#include <wctype.h>
int iswxdigit_l(wint_t wc, locale_t locale);
```

Description

The `iswxdigit_l()` function tests whether *wc* is a wide-character code representing a character of class xdigit in the locale represented by *locale*.

In all cases *wc* is a wint_t, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the locale represented byte *locale*, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Return Value

The `iswxdigit_l()` function shall return non-zero if *wc* is a hexadecimal digit wide-character code; otherwise, it shall return 0.

Errors

The `iswxdigit_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

isxdigit_l

Name

`isxdigit_l` — test for a hexadecimal digit character

Synopsis

```
#include <ctype.h>
int isxdigit_l(int c, locale_t locale);
```

Description

The `isxdigit_l()` function shall test whether *c* is a character of class xdigit in the locale represented by *locale*.

In all cases *c* is an int, the value of which the application shall ensure is a character representable as an unsigned char or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

Return Value

The `isxdigit_l()` function shall return non-zero if *c* is an uppercase character; otherwise, it shall return 0.

Errors

The `isxdigit_l()` function may fail if:

EINVAL

> *`locale`* is not a valid locale object handle.

tolower_l

Name

`tolower_l` — transliterate uppercase characters to lowercase

Synopsis

```
#include <ctype.h>
int tolower_l(int c, locale_t locale);
```

Description

The `tolower_l()` function has as a domain a type int, the value of which is representable as an unsigned char or the value of EOF. If the argument has any other value, the behavior is undefined. If the argument of `tolower_l()` represents an uppercase letter, and there exists a corresponding lowercase letter (as defined by character type information in the category LC_CTYPE in the locale represented by *`locale`*), the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

Return Value

Upon successful completion, `tolower_l()` shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

Errors

The `tolower_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

toupper_l

Name

`toupper_l` — transliterate lowercase characters to uppercase

Synopsis

```
#include <ctype.h>
int toupper_l(int c, locale_t locale);
```

Description

The `toupper_l()` function has as a domain a type int, the value of which is representable as an unsigned char or the value of EOF. If the argument has any other value, the behavior is undefined. If the argument of `toupper_l()` represents an lowercase letter, and there exists a corresponding loweuppercase letter (as defined by character type information in the category LC_CTYPE in the locale represented by *`locale`*), the result is the corresponding uppercase letter. All other arguments in the domain are returned unchanged.

Return Value

Upon successful completion, `toupper_l()` shall return the uppercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

Errors

The `toupper_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

towctrans_l

Name

`towctrans_l` — wide-character transliteration

Synopsis

```
#include <wctype.h>
wint_t towctrans_l(wint_t wc, wctrans_t desc, locale_t locale);
```

Description

The `towctrans_l()` function transliterates the wide-character code *`wc`* using the mapping described by *`desc`*. The setting of the LC_CTYPE category in the locale represented by *`locale`* should be the same as the one passed to `wctrans_l()` that returned the value *`desc`*. If the value of *`desc`* is invalid (that is, not obtained by a call to `wctrans_l()` with the same locale object *`locale`*), the result is unspecified.

Return Value

If successful, the `towctrans_l()` function shall return the mapped value of *`wc`* using the mapping described by *`desc`*. Otherwise, it shall return *`wc`* unchanged.

Errors

The `toupper_l()` function may fail if:

EINVAL

`desc` contains an invalid transliteration descriptor.

EINVAL

`locale` is not a valid locale object handle.

towlower_l

Name

`towlower_l` — transliterate uppercase wide-character code to lowercase

Synopsis

```
#include <wctype.h>
wint_t towlower_l(wint_t wc, locale_t locale);
```

Description

The `towlower_l()` function has as a domain a type wint_t, the value of which the application shall ensure is a character representable as a wchar_t, and a wide-character code corresponding to a valid character in the locale represented by *`locale`* or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of `towlower_l()` represents an uppercase wide-character code, and there exists a corresponding lowercase wide-character code (as defined by character type information in the locale category LC_CTYPE of the locale *`locale`*), the result is the corresponding lowercase wide-character code. All other arguments in the domain are returned unchanged.

Return Value

Upon successful completion, `towlower_l()` shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

Errors

The `towlower_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

towupper_l

Name

`towupper_l` — transliterate lowercase wide-character code to uppercase

Synopsis

```
#include <wctype.h>
wint_t towupper_l(wint_t wc, locale_t locale);
```

Description

The `towupper_l()` function has as a domain a type wint_t, the value of which the application shall ensure is a character representable as a wchar_t, and a wide-character code corresponding to a valid character in the locale represented by *`locale`* or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of `towupper_l()` represents an lowercase wide-character code, and there exists a corresponding uppercase wide-character code (as defined by character type information in the locale category LC_CTYPE of the locale *`locale`*), the result is the corresponding uppercase wide-character code. All other arguments in the domain are returned unchanged.

Return Value

Upon successful completion, `towupper_l()` shall return the uppercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

Errors

The `towupper_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

wctrans_l

Name

`wctrans_l` — define character mapping

Synopsis

```
#include <wctype.h>
wctrans_t wctrans_l(const char *charclass, locale_t locale);
```

Description

The `wctrans_l()` function is defined for valid character mapping names identified in the locale represented by *`locale`*. The *`charclass`* is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales: tolower and toupper.

The function shall return a value of type wctrans_t, which can be used as the second argument to subsequent calls of `towctrans_l()`. The `wctrans_l()` function determines values of wctrans_t according to the rules of the coded character set defined by character mapping information in the locale *`locale`* (category LC_CTYPE). The values returned by `wctrans_l()` are only valid in calls to `towctrans_l()` with the same locale object *`locale`*.

Return Value

The `wctrans_l()` function shall return 0 and may set `errno` to indicate the error if the given character mapping name is not valid for the locale *locale* (category LC_CTYPE); otherwise, it shall return a non-zero object of type wctrans_t that can be used in calls to `towctrans_l()`.

Errors

The `wctrans_l()` function may fail if:

EINVAL

> The character mapping name pointed to by *charclass* is not valid in the locale represented by *locale*.

EINVAL

> *locale* is not a valid locale object handle.

wctype_l

Name

`wctype_l` — define character class

Synopsis

```
#include <wctype.h>
wctype_t wctype_l(const char *property, locale_t locale);
```

Description

The `wctype_l()` function is defined for valid character class names as defined in the locale represented by *`locale`*. The *`property`* is a string identifying a generic character class for which codeset-specific type information is required. The following character class names are defined in all locales:

alnum	digit	punct
alpha	graph	space
blank	lower	upper
cntrl	print	xdigit

Additional character class names defined in the locale definition file (category LC_CTYPE) can also be specified.

The function shall return a value of type wctype_t, which can be used as the second argument to subsequent calls of `iswctype_l()`. The `wctype_l()` function determines values of wctype_t according to the rules of the coded character set defined by character type information in the locale represented by *`locale`* (category

LC_CTYPE). The values returned by `wctype_l()` are valid only in calls to `iswctype_l()` with the same *`locale`*.

Return Value

The `wctype_l()` function shall return 0 if the given character class name is not valid for the *`locale`* (category LC_CTYPE); otherwise, it shall return an object of type wctype_t that can be used in calls to `iswctype_l()`.

Errors

The `wctype_l()` function may fail if:

EINVAL

> *`locale`* is not a valid locale object handle.

3.3.2. Extended String Functions

strcasecmp_l, strncasecmp_l

Name

`strcasecmp_l, strncasecmp_l` — case-insensitive string comparisons

Synopsis

```
#include <strings.h>
int strcasecmp_l(const char *s1, const char *s2, locale_t
locale);
int strncasecmp_l(const char *s1, const char *s2, size_t n,
locale_t locale);
```

Description

The `strcasecmp_l()` function compares, while ignoring differences in case, the string pointed to by *s1* to the string pointed to by *s2*. The `strncasecmp_l()` function compares, while ignoring differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2*. The information about the case of the characters comes from the locale associated with *locale*.

Return Value

Upon completion, `strcasecmp_l()` shall return an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is, ignoring case, greater than, equal to, or less than the string pointed to by `s2`, respectively.

Upon successful completion, `strncasecmp_l()` shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by `s1` is, ignoring case, greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`, respectively.

Errors

The `strcasecmp_l()` and `strncasecmp_l()` functions may fail if:

EINVAL

> `locale` is not a valid locale object handle.

strcoll_l

Name

`strcoll_l` — string comparison using collating information

Synopsis

```
#include <string.h>
int strcoll_l(const char *s1, const char *s2, locale_t locale);
```

Description

The `strcoll_l()` function shall compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC_COLLATE category of the locale represented by *locale*.

The `strcoll_l()` function shall not change the setting of `errno` if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strcoll_l()`, then check `errno`.

Return Value

Upon successful completion, `strcoll_l()` shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the locale represented by *locale*. On error, `strcoll_l()` may set `errno`, but no return value is reserved to indicate an error.

Errors

The `strcoll_l()` function may fail if:

EINVAL

The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

EINVAL

locale is not a valid locale object handle.

strxfrm_l

Name

`strxfrm_l` — string transformation

Synopsis

```
#include <string.h>
int strxfrm_l(char *s1, const char *s2, size_t n, locale_t
locale);
```

Description

The `strxfrm_l()` function shall transform the string pointed to by *s2* and place the resulting string into the array pointed to by *s1*. The transformation is such that if `strcmp()` is applied to two transformed strings, it returns a value greater than, equal to, or less than 0, corresponding to the result of `strcoll_l()` applied to the same two original strings with the same *locale*. No more than *n* bytes are placed into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The `strxfrm_l()` function shall not change the setting of `errno` if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strcoll_l()`, then check `errno`.

Errors

The `strxfrm_l()` function may fail if:

EINVAL

The string pointed to by the *s2* argument contains characters outside the domain of the collating sequence.

EINVAL

locale is not a valid locale object handle.

wcscasecmp_l, wcsncasecmp_l

Name

`wcscasecmp_l, wcsncasecmp_l` — case-insensitive wide-character string comparisons

Synopsis

```
#include <wchar.h>
int wcscasecmp_l(const wchar_t *ws1, const wchar_t *ws2,
locale_t locale);
int wcsncasecmp_l(const wchar_t *ws1, const wchar_t *ws2, size_t
n, locale_t locale);
```

Description

The `wcscasecmp_l()` function compares, while ignoring differences in case, the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*. The `wcsncasecmp_l()` function compares, while ignoring differences in case, not more than *n* wide-characters from the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*. The information about the case of the characters comes from the locale associated with *locale*.

Return Value

Upon completion, `wcscasecmp_l()` shall return an integer greater than, equal to, or less than 0, if the wide-character string pointed to by *ws1* is, ignoring case, greater than, equal to, or less than the wide-character string pointed to by *ws2*, respectively.

Upon successful completion, `wcsncasecmp_l()` shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *ws1* is, ignoring case, greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*, respectively.

Errors

The `wcscasecmp_l()` and `wcsncasecmp_l()` functions may fail if:

EINVAL

   locale is not a valid locale object handle.

wcscoll_l

Name

wcscoll_l — wide-character string comparison using collating information

Synopsis

```
#include <wchar.h>
int wcscoll_l(const wchar_t *ws1, const wchar_t *ws2, locale_t
locale);
```

Description

The `wcscoll_l()` function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, both interpreted as appropriate to the LC_COLLATE category of the locale represented by *locale*.

The `wcscoll_l()` function shall not change the setting of `errno` if successful.

An application wishing to check for error situations should set `errno` to 0 before calling `wcscoll_l()`. If `errno` is non-zero on return, an error has occurred.

Return Value

Upon successful completion, `wcscoll_l()` shall return an integer greater than, equal to, or less than 0, according to whether the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, when both are interpreted as appropriate to the locale represented by *`locale`*. On error, `wcscoll_l()` may set `errno`, but no return value is reserved to indicate an error.

Errors

The `wcscoll_l()` function may fail if:

EINVAL

The *`ws1`* or *`ws2`* arguments contain wide-character codes outside the domain of the collating sequence.

EINVAL

`locale` is not a valid locale object handle.

wcsxfrm_l

Name

wcsxfrm_l — wide-character string transformation

Synopsis

```
#include <wchar.h>
int wcsxfrm_l(wchar_t *ws1, const wchar_t *ws2, size_t n,
locale_t locale);
```

Description

The wcsxfrm_l() function shall transform the wide-character string pointed to by *ws2* and place the resulting wide-character string into the array pointed to by *ws1*. The transformation is such that if wcscmp() is applied to two transformed wide strings, it returns a value greater than, equal to, or less than 0, corresponding to the result of wcscoll_l() applied to the same two original wide-character strings and the same locale object *locale*. No more than *n* wide-character codes are placed into the resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0, ws1 is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The wcsxfrm_l() function shall not change the setting of errno if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, then call wcsxfrm_l(), then check errno.

Return Value

The `wcsxfrm_l()` function shall return the length of the transformed wide-character string (not including the terminating null wide-character code). If the value returned is *`n`* or more, the contents of the array pointed to by *`ws1`* are indeterminate.

On error, the `wcsxfrm_l()` function may set `errno`, but no return value is reserved to indicate an error.

Errors

The `wcsxfrm_l()` function may fail if:

EINVAL

> The wide-character string pointed to by *`ws2`* contains wide-character codes outside the domain of the collating sequence.

EINVAL

> *`locale`* is not a valid locale object handle.

3.3.3. Extended Function for Direct Locale Data Access

Functions which do not fit in the two other classes but still need extensions are described in this section.

Possible other candidates for inclusion as extended functions are:

```
strptime_l()
```

nl_langinfo_l

Name

`nl_langinfo_l` — language information

Synopsis

```
#include <langinfo.h>
char *nl_langinfo_l(nl_item item, locale_t locale);
```

Description

The `nl_langinfo_l()` function shall return a pointer to a string containing information relevant to the particular language or cultural area defined in the locale represented by *`locale`* (see `<langinfo.h>`). The manifest constant names and values of item are defined in `<langinfo.h>`. For example:

```
locale_t loc = ...
...
```

```
nl_langinfo_l(ABDAY_1, loc)
```

would return a pointer to the string "Dom" if the language of the locale represented by `locale` was Portuguese, and "Sun" if the identified language was English.

Return Value

In a locale where langinfo data is not defined, `nl_langinfo_l()` shall return a pointer to the corresponding string in the POSIX locale. In all locales, `nl_langinfo_l()` shall return a pointer to an empty string if item contains an invalid setting.

Errors

The `nl_langinfo_l()` function may fail if:

EINVAL

   `locale` is not a valid locale object handle.

strfmon_l

Name

`strfmon_l` — convert monetary value to a string

Synopsis

```
#include <monetary.h>
ssize_t strfmon_l(char *restrict s, size_t maxsize, locale_t
locale, const char *restrict format, ...);
```

Description

The `strfmon_l()` function shall place characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string that contains two types of objects: *plain characters*, which are simply copied to the output stream, and *conversion specifications*, each of which results in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

The application shall ensure that a conversion specification consists of the following sequence:

- A % character
- Optional flags

- Optional field width
- Optional left precision
- Optional right precision
- A required conversion character that determines the conversion to be performed

Flags

One or more of the following optional flags can be specified to control the conversion:

`=`*`f`*

An = followed by a single character f which is used as the numeric fill character. The application shall ensure that the fill character is representable in a single byte in order to work with precision and width counts. The default numeric fill character is the <space> character. This flag does not affect field width filling which always uses the <space> character. This flag is ignored unless a left precision (see below) is specified.

^

Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the locale represented by *`locale`*.

\+ or (

Specify the style of representing positive and negative currency amounts. Only one of + or (may be specified. If + is specified, the *`locale`*'s equivalent of + and - are used (for example, in the U.S., the empty string if positive and - if negative). If (is specified, negative amounts are enclosed within parentheses. If neither flag is specified, the + style is used.

!

Suppress the currency symbol from the output conversion.

-

Specify the alignment. If this flag is present all fields are left-justified (padded to the right) rather than right-justified.

Field Width

w

A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag - is specified). The default is 0.

Left Precision

#*n*

A # followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the `strfmon_l()` function aligned in the same columns. It can also be used to fill unused positions with a special character as in "$***123.45". This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the =*f* flag above).

Right Precision

.*p*

A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision `p` is 0, no radix character appears. If a right precision is not included, a default specified by the locale represented by `locale`is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters

The conversion characters and their meanings are:

i

The double argument is formatted according to international currency format of the locale represented by `locale` (for example, in the U.S.: USD 1,234.56).

n

The double argument is formatted according to the national currency format of the locale represented by `locale` (for example, in the U.S.: $1,234.56).

%

Convert to a %; no argument is converted. The entire conversion specification shall be "%%".

Locale Information

The LC_MONETARY category of the locale represented by `locale` affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the LC_NUMERIC category), the grouping separator, the currency symbols, and formats. The international currency symbol should be conformant with the ISO 4217: 1995 standard.

If the value of maxsize is greater than {SSIZE_MAX}, the result is implementation-defined.

Return Value

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, `strfmon_l()` shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, -1 shall be returned, the contents of the array are indeterminate, and `errno` shall be set to indicate the error.

Errors

The `strfmon_l()` function shall fail if:

E2BIG

Conversion stopped due to lack of space in the buffer.

The `strfmon_l()` function may fail if:

EINVAL

locale is not a valid locale object handle.

strftime_l

Name

`strftime_l` — convert date and time to a string

Synopsis

```
#include <time.h>
ssize_t strftime_l(char *restrict s, size_t maxsize, locale_t
locale, const char *restrict format, const struct *restrict
timeptr);
```

Description

The `strftime_l()` function shall place bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character, possibly followed by an E or O modifier, and a terminating conversion character that determines the conversion specification s behavior. All ordinary characters (including the terminating null byte) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* bytes are placed into the array. Each conversion specification is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the locale represented by *locale* and by the values contained in the structure pointed to by timptr.

Local timezone information is used as though `strftime_l()` called `tzset()`.

%a

Replaced by `locale`'s abbreviated weekday name.

%A

Replaced by `locale`'s full weekday name.

%b

Replaced by `locale`'s abbreviated month name.

%B

Replaced by `locale`'s full month name.

%c

Replaced by `locale`'s appropriate date and time representation.

%C

Replaced by the century number (the year divided by 100 and truncated to an integer) as a decimal number [00-99].

%d

Replaced by the day of the month as a decimal number [01,31].

%D

Same as %m/%d/%y.

%e

Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded by a space.

%F

Equivalent to %Y-%m-%d (the ISO 8601: 1988 standard date format).

%g

Replaced by the last 2 digits of the week-based year (see below) as a decimal number (00-99).

%G

Replaced by the week-based year (see below) as a decimal number (for example, 1977).

%h

Same as %b.

%H

Replaced by the hour (24-hour clock) as a decimal number [00,23].

%I

Replaced by the hour (12-hour clock) as a decimal number [01,12].

%j

Replaced by the day of the year as a decimal number [001,366].

%m

Replaced by the month as a decimal number [01,12].

%M

Replaced by the minute as a decimal number [00,59].

%n

Replaced by a <newline> character.

%p

Replaced by *`locale`*'s equivalent of either a.m. or p.m.

%r

Replaced by the time in a.m. and p.m. notation; in the POSIX locale this is equivalent to %I:%M:%S %p.

%R

Replaced by the time in 24 hour notation (%H:%M).

%S

Replaced by the second as a decimal number [00,61].

%t

Replaced by a <tab> character.

%T

Replaced by the time (%H:%M:%S).

%u

Replaced by the weekday as a decimal number [1,7], with 1 representing Monday.

%U

Replaced by the week number of the year (Sunday as the first day of the week) as a decimal number [00,53].

%V

Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1.

%w

Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday.

%W

Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.

%x

Replaced by `locale`'s appropriate date representation.

%X

Replaced by `locale`'s appropriate time representation.

%y

Replaced by the year without century as a decimal number [00,99].

%Y

Replaced by the year with century as a decimal number.

%z

Replaced by the offset from UTC in the ISO 8601: 1988 standard format -0430 (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no timezone is determinable.

%Z

Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists.

%%

Replaced by %.

If a conversion specification does not correspond to any of the above, the behavior is undefined.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the locale represented by *`locale`*, (see ERA in the Base Definitions volume of IEEE Std. 1003.1-200x, Section 7.3.5, LC_TIME) the behavior shall be as if the unmodified conversion specification were used.

%Ec

Replaced by *`locale`*'s alternative appropriate date and time representation.

%EC

Replaced by the name of the base year (period) in *`locale`*'s alternative representation.

%Ex

Replaced by *`locale`*'s alternative date representation.

%EX

Replaced by *`locale`*'s alternative time representation.

%Ey

Replaced by the offset from %EC (year only) in *`locale`*'s alternative representation.

%EY

Replaced by the full alternative year representation.

%Od

Replaced by the day of the month, using *`locale`*'s alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero; otherwise, with leading spaces.

%Oe

Replaced by the day of the month, using `locale`'s alternative numeric symbols, filled as needed with leading spaces.

%OH

Replaced by the hour (24-hour clock) using `locale`'s alternative numeric symbols.

%OI

Replaced by the hour (12-hour clock) using `locale`'s alternative numeric symbols.

%Om

Replaced by the month using `locale`'s alternative numeric symbols.

%OM

Replaced by the minutes using `locale`'s alternative numeric symbols.

%OS

Replaced by the seconds using `locale`'s alternative numeric symbols.

%Ou

Replaced by the weekday as a number in `locale`'s alternative representation (Monday=1).

%OU

Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using `locale`'s alternative numeric symbols.

%OV

Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to %V) using `locale`'s alternative numeric symbols.

%Ow

> Replaced by the number of the weekday (Sunday=0) using *locale*'s alternative numeric symbols.

%OW

> Replaced by the week number of the year (Monday as the first day of the week) using *locale*'s alternative numeric symbols.

%Oy

> Replaced by the year (offset from %C) using *locale*'s alternative numeric symbols.

%g, %G, and %V give values according to the ISO 8601: 1988 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

If a conversion specifier is not one of the above, the behavior is undefined.

Return Value

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, `strftime_l()` shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array are indeterminate.

Errors

The `strfmon_l()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

3.4. Support for the thread-aware locale model

To complete support for the new thread model there are not many more changes necessary. We have to support setting the locale in a thread-local way and need a new interface for that. Everything else works without interface changes since this is intend. But the description of the functions using locale data has to be modified slightly. Due to the extend of this work (a large percentage of the functions in the C library is effected) we will not list all individual changes here. Instead we show in an example the changes which have to be made. They are not extensive. Compare the man page for `strcoll` given at the end of this section with the one from the Unix standard.

uselocale

Name

`uselocale` — use locale in current thread

Synopsis

```
#include <locale.h>
locale_t uselocale(locale_t newloc);
```

Description

The `uselocale()` function is similar to the `setlocale()` function but it does not effect the global locale. Instead it is selecting the new locale only for the current thread. The locale setting for all other threads remains unchanged.

The value passed for the parameter *`newloc`* must be a value which was returned by `newlocale()` or `duplocale()`, the special locale object descriptor LC_GLOBAL_LOCALE, or zero.

> **Note:** Should we introduce a symbol LC_POSIX_LOCALE and/or LC_C_LOCALE to allow easy selection of the POSIX/C locale?

Unlike `setlocale()`, `uselocale()` does not allow replacing some categories only. If the user wants to install a locale which differs only in a few categories she has to use `newlocale()` to change a locale object equivalent to the currently used locale and install it.

The currently installed locale can be queried by calling `uselocale()` with a null parameter. The object returned is the one passed to `uselocale()` before or LC_GLOBAL_LOCALE in case there was no call to `uselocale()` for this thread before.

If LC_GLOBAL_LOCALE is passed in *`newloc`* the thread will from that point on again use the global locale determined by `setlocale()`.

If a locale object gets modified by a call to `newlocale()` while it is still being used in any thread the behavior of those threads accessing the locale data is unspecified.

The locale object passed to `uselocale()` should not be freed by a call to `freelocale()` unless no thread is using it.

Return Value

The function returns the locale handle previously passed to `uselocale()` for the current thread. If there was no such call before the return value is LC_GLOBAL_LOCALE.

Errors

The `uselocale()` function may fail if:

EINVAL

`locale` is not a valid locale object handle.

newlocale

Name

`newlocale` — create or modify a locale object

Synopsis

```
#include <locale.h>
locale_t newlocale(int category_mask, const char *locale,
locale_t base);
```

Description

> **Note:** This is an extension of the man page earlier in this document.

The special locale object LC_GLOBAL_LOCALE (even when returned by `uselocale()`) must not be passed for the parameter *base*.

freelocale

Name

`freelocale` — Free resources allocated for locale object

Synopsis

```
#include <locale.h>
void freelocale(locale_t locobj);
```

Description

> **Note:** This is an extension of the man page earlier in this document.

If a locale object is passed to `freelocale()` while it is still in use by a thread, initiated by a call to `uselocale()`, the behavior is unspecified.

strcoll

Name

`strcoll` — string comparison using collating information

Synopsis

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

The `strcoll()` function shall compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC_COLLATE category of the locale of the current thread.

The `strcoll()` function shall not change the setting of `errno` if successful.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, then call `strcoll()`, then check `errno`.

Return Value

Upon successful completion, `strcoll()` shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the locale represented by *locale*. On error, `strcoll()` may set `errno`, but no return value is reserved to indicate an error.

Errors

The `strcoll()` function may fail if:

EINVAL

The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

4. Implementation

The implementation of locale objects is by itself a straight forward process. It heavily depends on the way locales are implemented in the current implementation in the C library.

The conversion to the locale object model should not be difficult. The functionality is already there. A little bit more complicated is optimizing the implementation. The implementation should use reference counting to avoid duplicate copies and it should delay unloading of currently unused data since the new model will encourage switching between different locales.

Changing the implementation in the GNU C library to this new model took only a few hours. In glibc 2.1 and 2.2 only the first two sets of functions described in the last section are available. Starting with glibc 2.3 everything will be available. The enabling of thread-local locales requires some changes in the runtime environment even for single-threaded applications. Architectures with dedicated thread registers should have no problem with this. The additional process startup costs are expected to be negligible.

With a thread register available the places in the C library which are using locale data have to be rewritten slightly. The GNU C library organizes all data for a category is in one structure. Therefore all which has to change is they way the address of this data structure is determined. Instead of using a global variable (as it is done in the moment) it will be a memory access through the thread register. The thread descriptor referenced by the register will contain the current pointer for each category.

At program start time the thread-local locale is not set and the global locale is used. This can be imitated by simply initializing the locale category data pointers in the thread descriptor with pointer to the locale data of the global locale (easy if the global locale uses the same format). The same operation will be needed if `uselocale()` is called to LC_GLOBAL_LOCALE. Programs expecting the POSIX locale behavior will not see any difference, only programs using the new `uselocale()` interface will.

5. Application Usage

A few examples will show how the new functions can be used. We will not provide examples for the C++ library implementation here, though.

5.1. Server handling different locales

A server which might handle clients which require different locales might look like this:

```
/* Data structure for each connection.  */
struct conn {
  int fd;
  locale_t loc;                                              ❶
  ...
} *conns;

int add_new_conn (int idx, int fd, const char *name)
{
  conns[idx].fd = fd;
  conns[idx].loc = newlocale (LC_ALL_MASK, name, NULL);     ❷
  return conns[idx].loc != NULL;
}

void del_conn (int idx)
{
  close (conns[idx].fd);
  freelocale (conns[idx].loc);                               ❸
}

{
  ...
  while (1) {
    int n = poll (fds, nfds, 0);

    while (1) {
```

```
      struct conn *thisconn = find_connection (fds);
      if (thisconn == NULL)
        break;

      uselocale (conn->loc);                                    ❹

      handle_conn (conn);
    }
}
```

❶ Each connection structure contains a locale_t object.

❷ Here we create the locale object for each connection. We start all over from a name and do not use `duplocale()` here. Hopefully the implementation will optimize for this and cache previously used locales.

❸ When the connection is not needed anymore we close the file and delete the locale object.

❹ When handling input or output for a connection we switch to the locale which was requested for it. This should be a very fast operation since only a couple of pointers have to be modified.