```
Re-entrancy of Newlib
=====================

* It's not enough to make your own code "thread-safe" by identifying and
  protecting critical sections. You must consider code contributed by
  others, i.e., the Newlib C library. Newlib has been written with
  some support for re-entrancy and thread-safety.

* What is "re-entrancy"? It means that the code can be called "recursively
  and concurrently from multiple processes. To be re-entrant, a function
  must:
    - hold no static data
    - not return a pointer to static data
    - work only on the data provided to it by the caller
    - not call non-reentrant functions" [2]

  Basically, no global or local-static variables. In addition, critical sections
  must be identified and protected with locks.

  A re-entrant function is like a special case of two threads that modify a
  shared resource, except that instead of having two separate pieces of code
  that run as threads, it is the same piece of code instantiated as multiple
  threads (e.g., a web server handler).

* Re-entrancy is important in the C library since two separate threads may end
  up calling the same C function at arbitrary times. If the C library function
  is not re-entrant, one thread can "stomp" on another through the C library
  function.

* Here is a non-reentrant function (from [1]):

    char *strtoupper(char *string) {
      static char buffer[MAX_STRING_SIZE];
      int index;

      for (index = 0; string[index]; index++)
              buffer[index] = toupper(string[index]);
      buffer[index] = 0;

      return buffer;
    }

  Think about what happens if thread 1 is executing this function, it is
  interrupted, then thread 2 calls this function. The local-static area
  'buffer' is IN THE SAME MEMORY LOCATION for both threads (it is NOT on
  the stack, it is in .bss) thus will be corrupted.

* As an example of a re-entrant function, consider the recursive
  implementation of the factorial function:

      int factorial(int n) { return (n<2) ? 1 : n*factorial(n-1); }

  It doesn't matter if this function is executing in thread 1, interrupted
  to switch to thread 2, then thread 2 executes this function too. The
  correct answer will still be given in both threads because the state of
  this function (the variable 'n') is stored on the stack (or in a register)
  which is separate for each thread.

* Newlib provides some support for thread safety through the use of
```

```
  a re-entrancy structure that defines the entire state of the C library
  code including:

    - separate file pointers (stdin, stdout, etc.) for each thread
    - errno, the global variable indicating the last failure in standard
      I/O functions
    - state for the random number generator (so it remains PSEUDO-random
      and does not change with other thread activity)
    - atexit handlers separate to each thread
    - other stuff (see reent.h): tmpnam(), setlocale(), strtod(),
      strtok(), multi-byte characters, signals

* All Newlib functions that require state (e.g., rand, strtok, anything
  that uses errno) do not use a global variable for this state but instead
  use an entry in a "struct _reent" structure that has been previously
  allocated.

  How? The library file libc/reent/impure.c defines the global variable
  "impure_data" of type "struct _reent". This variable is initialized
  using the _REENT_INIT() macro (from "sys/reent.h") that sets all fields
  to NULL values (mostly).
```

```
1  #include <reent.h>
2
3  static struct _reent impure_data = _REENT_INIT (impure_data);
4  struct _reent *_impure_ptr = &impure_data;
5  struct _reent *_CONST _global_impure_ptr = &impure_data;
```

```
/*
 * struct _reent
 *
 * This structure contains *all* globals needed by the library.
 * It's raison d'etre is to facilitate threads by making all library routines
 * reentrant.  IE: All state information is contained here.
 */

struct _reent
{
  int _errno;                   /* local copy of errno */

  /* FILE is a big struct and may change over time.  To try to achieve binary
     compatibility with future versions, put stdin,stdout,stderr here.
     These are pointers into member __sf defined below.  */
  __FILE *_stdin, *_stdout, *_stderr;

  int  _inc;                    /* used by tmpnam */
  char _emergency[_REENT_EMERGENCY_SIZE];

  int _current_category;        /* used by setlocale */
  _CONST char *_current_locale;

  int __sdidinit;               /* 1 means stdio has been init'd */

  void _EXFUN((*__cleanup),(struct _reent *));

  /* used by mprec routines */
  struct _Bigint *_result;
  int _result_k;
  struct _Bigint *_p5s;
  struct _Bigint **_freelist;

  /* used by some fp conversion routines */
  int _cvtlen;                  /* should be size_t */
  char *_cvtbuf;

  union
    {
      struct
        {
          unsigned int _unused_rand;
          char * _strtok_last;
          char _asctime_buf[_REENT_ASCTIME_SIZE];
          struct __tm _localtime_buf;
          int _gamma_signgam;
          __extension__ unsigned long long _rand_next;
          struct _rand48 _r48;
          _mbstate_t _mblen_state;
          _mbstate_t _mbtowc_state;
          _mbstate_t _wctomb_state;
          char _l64a_buf[8];
          char _signal_buf[_REENT_SIGNAL_SIZE];
          int _getdate_err;
          _mbstate_t _mbrlen_state;
          _mbstate_t _mbrtowc_state;
          _mbstate_t _mbsrtowcs_state;
          _mbstate_t _wcrtomb_state;
          _mbstate_t _wcsrtombs_state;
        } _reent;
  /* Two next two fields were once used by malloc.  They are no longer
     used. They are used to preserve the space used before so as to
     allow addition of new reent fields and keep binary compatibility.   */
       struct
         {
```

```
66  #define _N_LISTS 30
67            unsigned char * _nextf[_N_LISTS];
68            unsigned int _nmalloc[_N_LISTS];
69          } _unused;
70      } _new;
71
72    /* atexit stuff */
73    struct _atexit *_atexit;      /* points to head of LIFO stack */
74    struct _atexit _atexit0;      /* one guaranteed table, required by ANSI */
75
76    /* signal info */
77    void (**(_sig_func))(int);
78
79    /* These are here last so that __FILE can grow without changing the offsets
80       of the above members (on the off chance that future binary compatibility
81       would be broken otherwise).  */
82    struct _glue __sglue;         /* root of glue chain */
83    __FILE __sf[3];               /* first three file descriptors */
84  };
85
86  #define _REENT_INIT(var) \
87    { 0, \
88      &var.__sf[0], \
89      &var.__sf[1], \
90      &var.__sf[2], \
91      0, \
92      "", \
93      0, \
94      "C", \
95      0, \
96      _NULL, \
97      _NULL, \
98      0, \
99      _NULL, \
100     _NULL, \
101     0, \
102     _NULL, \
103     { \
104       { \
105         0, \
106         _NULL, \
107         "", \
108         {0, 0, 0, 0, 0, 0, 0, 0, 0}, \
109         0, \
110         1, \
111         { \
112           {_RAND48_SEED_0, _RAND48_SEED_1, _RAND48_SEED_2}, \
113           {_RAND48_MULT_0, _RAND48_MULT_1, _RAND48_MULT_2}, \
114           _RAND48_ADD \
115         }, \
116         {0, {0}}, \
117         {0, {0}}, \
118         {0, {0}}, \
119         "", \
120         "", \
121         0, \
122         {0, {0}}, \
123         {0, {0}}, \
124         {0, {0}}, \
125         {0, {0}}, \
126         {0, {0}} \
127       } \
128     }, \
129     _NULL, \
130     {_NULL, 0, {_NULL}, {{_NULL}, {_NULL}, 0, 0}}, \
```

```
131       _NULL, \
132       {_NULL, 0, _NULL} \
133     }
134
135   #define _REENT_INIT_PTR(var) \
136     { var->_errno = 0; \
137       var->_stdin = &var->__sf[0]; \
138       var->_stdout = &var->__sf[1]; \
139       var->_stderr = &var->__sf[2]; \
140       var->_inc = 0; \
141       memset(&var->_emergency, 0, sizeof(var->_emergency)); \
142       var->_current_category = 0; \
143       var->_current_locale = "C"; \
144       var->__sdidinit = 0; \
145       var->__cleanup = _NULL; \
146       var->_result = _NULL; \
147       var->_result_k = 0; \
148       var->_p5s = _NULL; \
149       var->_freelist = _NULL; \
150       var->_cvtlen = 0; \
151       var->_cvtbuf = _NULL; \
152       var->_new._reent._unused_rand = 0; \
153       var->_new._reent._strtok_last = _NULL; \
154       var->_new._reent._asctime_buf[0] = 0; \
155       memset(&var->_new._reent._localtime_buf, 0, sizeof(var->_new._reent._localtime
    _buf)); \
156       var->_new._reent._gamma_signgam = 0; \
157       var->_new._reent._rand_next = 1; \
158       var->_new._reent._r48._seed[0] = _RAND48_SEED_0; \
159       var->_new._reent._r48._seed[1] = _RAND48_SEED_1; \
160       var->_new._reent._r48._seed[2] = _RAND48_SEED_2; \
161       var->_new._reent._r48._mult[0] = _RAND48_MULT_0; \
162       var->_new._reent._r48._mult[1] = _RAND48_MULT_1; \
163       var->_new._reent._r48._mult[2] = _RAND48_MULT_2; \
164       var->_new._reent._r48._add = _RAND48_ADD; \
165       var->_new._reent._mblen_state.__count = 0; \
166       var->_new._reent._mblen_state.__value.__wch = 0; \
167       var->_new._reent._mbtowc_state.__count = 0; \
168       var->_new._reent._mbtowc_state.__value.__wch = 0; \
169       var->_new._reent._wctomb_state.__count = 0; \
170       var->_new._reent._wctomb_state.__value.__wch = 0; \
171       var->_new._reent._mbrlen_state.__count = 0; \
172       var->_new._reent._mbrlen_state.__value.__wch = 0; \
173       var->_new._reent._mbrtowc_state.__count = 0; \
174       var->_new._reent._mbrtowc_state.__value.__wch = 0; \
175       var->_new._reent._mbsrtowcs_state.__count = 0; \
176       var->_new._reent._mbsrtowcs_state.__value.__wch = 0; \
177       var->_new._reent._wcrtomb_state.__count = 0; \
178       var->_new._reent._wcrtomb_state.__value.__wch = 0; \
179       var->_new._reent._wcsrtombs_state.__count = 0; \
180       var->_new._reent._wcsrtombs_state.__value.__wch = 0; \
181       var->_new._reent._l64a_buf[0] = '\0'; \
182       var->_new._reent._signal_buf[0] = '\0'; \
183       var->_new._reent._getdate_err = 0; \
184       var->_atexit = _NULL; \
185       var->_atexit0._next = _NULL; \
186       var->_atexit0._ind = 0; \
187       var->_atexit0._fns[0] = _NULL; \
188       var->_atexit0._on_exit_args._fntypes = 0; \
189       var->_atexit0._on_exit_args._fnargs[0] = _NULL; \
190       var->_sig_func = _NULL; \
191       var->__sglue._next = _NULL; \
192       var->__sglue._niobs = 0; \
193       var->__sglue._iobs = _NULL; \
194       memset(&var->__sf, 0, sizeof(var->__sf)); \
```

```
195   }
196
197 extern struct _reent *_impure_ptr __ATTRIBUTE_IMPURE_PTR__;
198 extern struct _reent *_CONST _global_impure_ptr __ATTRIBUTE_IMPURE_PTR__;
199
200 #if defined(__DYNAMIC_REENT__) && !defined(__SINGLE_THREAD__)
201 #ifndef __getreent
202   struct _reent * _EXFUN(__getreent, (void));
203 #endif
204 # define _REENT (__getreent())
205 #else /* __SINGLE_THREAD__ || !__DYNAMIC_REENT__ */
206 # define _REENT _impure_ptr
207 #endif /* __SINGLE_THREAD__ || !__DYNAMIC_REENT__ */
```

Newlib Re-entrancy
==========

* For an example of how library functions make use of this state structure, look at rand.c: instead of using a global variable for storing the random seed, it uses impure_data._new._reent._rand_next (through the macro _REENT_RAND_NEXT() defined in "reent.h").

* The macro _REENT defined in "reent.h" is equivalent to "_impure_ptr" which is set to be the address of "impure_data" (in impure.c)

  Thus, in a single-threaded environment, C library functions that have state refer to _REENT->some_field instead of a global variable (like rand.c does).

* If a context switch happens in the middle of rand(), it's OK because the switched-to thread's re-entrancy structure (there's a separate one for each thread!) is used and the switched-from thread's re-entrancy structure is left alone.

```
/*
DESCRIPTION
<<rand>> returns a different integer each time it is called; each
integer is chosen by an algorithm designed to be unpredictable, so
that you can use <<rand>> when you require a random number.
The algorithm depends on a static variable called the ''random seed'';
starting with a given value of the random seed always produces the
same sequence of numbers in successive calls to <<rand>>.

RETURNS
<<rand>> returns the next pseudo-random integer in sequence; it is a
number between <<0>> and <<RAND_MAX>> (inclusive).

NOTES
<<rand>> and <<srand>> are unsafe for multi-threaded applications.
<<rand_r>> is thread-safe and should be used instead.

PORTABILITY
<<rand>> is required by ANSI, but the algorithm for pseudo-random
number generation is not specified; therefore, even if you use
the same random seed, you cannot expect the same sequence of results
on two different systems.
*/

#include <stdlib.h>
#include <reent.h>

void
_DEFUN (srand, (seed), unsigned int seed)
{
  _REENT_CHECK_RAND48(_REENT);
  _REENT_RAND_NEXT(_REENT) = seed;
}

int
_DEFUN_VOID (rand)
{
  /* This multiplier was obtained from Knuth, D.E., "The Art of
     Computer Programming," Vol 2, Seminumerical Algorithms, Third
     Edition, Addison-Wesley, 1998, p. 106 (line 26) & p. 108 */
  _REENT_CHECK_RAND48(_REENT);
  _REENT_RAND_NEXT(_REENT) =
     _REENT_RAND_NEXT(_REENT) * __extension__ 6364136223846793005LL + 1;
  return (int)((_REENT_RAND_NEXT(_REENT) >> 32) & RAND_MAX);
}
```

```
How to Make Newlib Re-Entrant
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

* How does Newlib automatically create a separate "struct _reent" structure
  for each thread? It doesn't...you do! Newlib only initializes ONE
  "struct _reent" structure to hold the C library state for one thread, the
  main program.

* If you want to run multiple threads in your program AND you want your
  program to be thread safe (this isn't a 100% requirement!) then:

  - For every thread you create, you must also dynamically allocate
    a "struct _reent" structure to hold the thread's state

            struct _reent *thread_state =
                   (struct _reent *)malloc(sizeof(struct _reent));
            if (thread_state == 0) .....

  - You must initialize the elements of the structure using the
    _REENT_INIT_PTR() macro defined in "reent.h". For example:

            _REENT_INIT_PTR(thread_state);

  - When the thread terminates, you should clean up all stdio usage by
    the thread and call of its own atexit handlers by calling (see "reent.c"):

       _wrapup_reent(thread_state);

  - Then, reclaim all of the memory used by this structure by calling:

       _reclaim_reent(thread_state);
       free(thread_state);

  - All Newlib functions that have state reference this state through the global
    variable _impure_ptr (which points to a re-entrancy structure -- by
    default, the one named 'impure_data' declared in impure.c). In your
    interrupt handler for context switching, you must point _impure_ptr to the
    "struct _reent" area for the thread you are switching to:

      ....
       _impure_ptr = (threads[currThread].thread_state);
      ....

  - In your kernel, if you want to use C library functions you should
    restore _impure_ptr to point back to the original global reentrancy
    structure:

       _impure_ptr = &impure_data;

* What if you don't do all this? It's not the end of the world:

  - There is only one set of stdio streams (stdin, stdout, stderr) shared by
    all threads, so they could "step on each other's toes" and leave the stdio
    part of the library in an inconsistent state. But if you only have one
    thread that uses stdio, this won't be a problem.

  - Your random number generator, if used by more than one thread, may not
    generate the same sequence every time.
```

```
  - errno might not be correct.

  - You won't have atexit handlers that are separate for each thread.

  - strtok() might give incorrect results if it's used in multiple threads (or
    you can use the inherently thread-safe version strtok_r()).

  - ??? (I didn't write the library and the documentation doesn't spell this
    out explicitly so......who knows?....look at what's inside a _reent
    structure to see which library functions have state)

Problem Solved?
===============

* No. The re-entrancy support of Newlib is not good enough "out of the box" to
  guard against pre-emptive, concurrent calls to the functions. Co-routines
  will be fine (as long as the scheduler moves _impure_ptr around to point to
  each coroutine's re-entrancy structure), but pre-emptive threads won't.

* Why? Take a look at _write_r, the recursive cover routine for the system
  call _write:

   _ssize_t _write_r(struct _reent *ptr, int fd, const void *buf, size_t cnt)
   {
      _ssize_t ret;

      errno = 0;     // THE ONE AND ONLY GLOBAL VARIABLE!!!
      if ((ret=(_ssize_t)_write(fd, buf, cnt)) != -1 && errno != 0) {
        ptr->errno = errno;   // Save a copy of the global variable in
                              // re-entrancy structure
      }
      return ret;
   }

  This function assumes that the system call _write() sets errno to indicate
  why the function failed (if it fails) and then copies this global variable
  into the thread's re-entrancy structure. This won't work if the thread
  is interrupted in between the call to _write() and the saving of errno
  to the re-entrancy structure. If another thread executes and calls _write()
  (or any other function that modifies errno) then when the original thread
  resumes execution, the global errno variable will have changed.

  Solutions:
    - write your own _write_r function and use a lock to protect the
      critical section
    - ignore it (note that the default _write() for ARM doesn't even set errno)

* Anything else? Yes, you should probably investigate the pre-emptive safety
  of all C functions you call (i.e., check for critical sections).

* Here's an important one (from the Newlib documentation for malloc()):

    "If you have multiple threads of execution which may call any of these
    routines, or if any of these routines may be called reentrantly, then you
    must provide implementations of the __malloc_lock and __malloc_unlock
    functions for your system. See the documentation for those functions."

  Since malloc() depends upon a lot of internal state, and it calls _sbrk()
  which carves out memory from the global heap (there is no per-thread heap!),
```

```
  we cannot have malloc() interrupted at random times. The implementation of
  malloc() is "smart" because it is "fine-grained": it only makes calls to
  __malloc_lock() and __malloc_unlock() when necessary to ensure consistency of
  its internal data structures and the heap. The alternative is ugly (poor
  performance):

              lock_acquire(&lock);  // "coarse-grained" locking
              malloc();
              lock_release(&lock);

* You do, however, have to actually write the __malloc_lock() and
  __malloc_unlock() functions because the default implementations in the library

  don't do anything.

* You can use the LDREX/STREX instructions to atomically implement the locking
  needed for __malloc_lock(), but there's an extra little twist:

    "A call to <<malloc>> may call <<__malloc_lock>> recursively; that is, the
    sequence of calls may go <<__malloc_lock>>, <<__malloc_lock>>,
    <<__malloc_unlock>>, <<__malloc_unlock>>.  Any implementation of these
    routines must be careful to avoid causing a thread to wait for a lock that
    it already holds."

  Any ideas on how to implement this?

References
==========

[1]: http://www.unet.univie.ac.at/aix/aixprggd/genprogc/writing_reentrant_thread
_safe_code.htm
[2]: http://en.wikipedia.org/w/index.php?title=Reentrant&oldid=141212078
```

```
1  /* connector for write */
2
3  #include <reent.h>
4  #include <unistd.h>
5
6  _READ_WRITE_RETURN_TYPE
7  _DEFUN (write, (fd, buf, cnt),
8       int fd _AND
9       const void *buf _AND
10      size_t cnt)
11 {
12 #ifdef REENTRANT_SYSCALLS_PROVIDED
13   return _write_r (_REENT, fd, buf, cnt);
14 #else
15   return _write (fd, buf, cnt);
16 #endif
17 }
```

```
1  #include <reent.h>
2  #include <unistd.h>
3  #include <_syslist.h>
4
5  /* We use the errno variable used by the system dependent layer.  */
6  #undef errno
7  extern int errno;
8
9  _ssize_t
10 _DEFUN (_write_r, (ptr, fd, buf, cnt),
11      struct _reent *ptr _AND
12      int fd _AND
13      _CONST _PTR buf _AND
14      size_t cnt)
15 {
16   _ssize_t ret;
17
18   errno = 0;
19   if ((ret = (_ssize_t)_write (fd, buf, cnt)) == -1 && errno != 0)
20     ptr->_errno = errno;
21   return ret;
22 }
```

Thread safety

From Wikipedia, the free encyclopedia

Thread safety is a computer programming concept applicable in the context of multi-threaded programs. A piece of code is **thread-safe** if it functions correctly during simultaneous execution by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.

Thread safety is a key challenge in multi-threaded programming. It was once only a concern of the operating system programmer but since the late 1990s has become a commonplace issue. In a multi-threaded program, several threads execute simultaneously in a shared address space. Every thread has access to virtually all the memory of every other thread. Thus the flow of control and the sequence of accesses to data often have little relation to what would be reasonably expected by looking at the text of the program, violating the principle of least astonishment. Thread safety is a property aimed at minimizing surprising behavior by re-establishing some of the correspondences between the actual flow of control and the text of the program.

Contents

Identification

It is not easy to determine if a piece of code is thread-safe or not. However, there are several indicators that suggest the need for careful examination to see if it is unsafe:

- accessing global variables or the heap
- allocating/reallocating/freeing resources that have global limits (files, sub-processes, etc.)
- indirect accesses through handles or pointers
- any *visible side-effect* (e.g., access to volatile variables in the C programming language)

A subroutine is reentrant, and thus thread-safe, if it only uses variables from the stack, depends only on the arguments passed in, and only calls other subroutines with similar properties. This is sometimes called a "pure function", and is much like a mathematical function.

Implementation

There are a few ways to achieve thread safety:

- *re-entrancy*: Basically, writing code in such a way that it can be interrupted during one task, *reentered* to perform another task, and then resumed on its original task. This usually precludes the saving of state information, such as by using static or global variables.
- *mutual exclusion*: Access to shared data is *serialized* using mechanisms that ensure only one thread is accessing the shared data at any time. Great care is required if a piece of code accesses multiple shared pieces of data - problems include race conditions, deadlocks, livelocks, starvation, and various other ills enumerated in many operating systems textbooks.
- *thread-local storage*: Variables are localized so that each thread has its own private copy. The variables retain their values across subroutine and other code boundaries, and the code which accesses them might not be reentrant, but since they are local to each thread, they are thread-safe.
- *Atomic operations*: Shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library. Since the operations are atomic, the shared data are always kept in a valid state, no matter what other threads access it. Atomic operations form the basis of many thread locking mechanisms.

A commonly used idiom combines these approaches:

- make changes to a private copy of the shared data and then atomically update the shared data from the private copy. Thus, most of the code is concurrent, and little time is spent serialized.

See also

- Control flow analysis
- Priority inversion
- Concurrency control
- exception safety
- Communicating sequential processes - a technique for analyzing concurrency

External links

- Short description of thread-safe (http://whatis.techtarget.com/definition/0,,sid9_gci331590,00.html)
- Writing Reentrant and Thread-Safe Code (http://www.unet.univie.ac.at/aix/aixprggd/genprogc/writing_reentrant_thread_safe_code.htm)
- Thread-safe Tcl Extensions (http://wiki.tcl.tk/3839) (wiki page)
- Article "Thread-safe webapps using Spring (http://javalobby.org/articles/thread-safe/index.jsp) " by Steven Devijver
- Thread-safe design (http://www.javaworld.com/javaworld/javaqa/1999-04/01-threadsafe.html)
- Article "Design for thread safety (http://www.javaworld.com/javaworld/jw-08-1998/jw-08-techniques.html) " by Bill Venners
- Article "Write thread-safe servlets (http://www.javaworld.com/javaworld/jw-07-2004/jw-0712-threadsafe.html) " by Phillip Bridgham
- Article "Smart Pointer Thread Safety (http://jelovic.com/articles/smart_pointer_thread_safety.htm) " by Dejan Jelovic
- A Short Guide to Mastering Thread-Safety (http://www.thinkingparallel.com/2006/10/15/a-short-guide-to-mastering-thread-safety/)

Categories: Articles with unsourced statements since February 2007 | All articles with unsourced statements | Computer programming | Concurrent computing

---

and formatting information; `localeconv` reports on the settings of the current locale.

This is a minimal implementation, supporting only the required `"C"` value for *locale*; strings representing other locales are not honored unless MB_CAPABLE is defined in which case three new extensions are allowed for LC_CTYPE or LC_MESSAGES only: `"C-JIS"`, `"C-EUCJP"`, `"C-SJIS"`, or `"C-ISO-8859-1"`. (`""` is also accepted; it represents the default locale for an implementation, here equivalent to `"C"`.)

If you use `NULL` as the *locale* argument, `setlocale` returns a pointer to the string representing the current locale (always `"C"` in this implementation). The acceptable values for *category* are defined in `` `locale.h' `` as macros beginning with `"LC_"`, but this implementation does not check the values you pass in the *category* argument.

`localeconv` returns a pointer to a structure (also defined in `` `locale.h' ``) describing the locale-specific conventions currently in effect.

`_localeconv_r` and `_setlocale_r` are reentrant versions of `localeconv` and `setlocale` respectively. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
`setlocale` returns either a pointer to a string naming the locale currently in effect (always `"C"` for this implementation, or, if the locale request cannot be honored, `NULL`.

`localeconv` returns a pointer to a structure of type `lconv`, which describes the formatting and collating conventions in effect (in this implementation, always those of the C locale).

Portability
ANSI C requires `setlocale`, but the only locale required across all implementations is the C locale.

No supporting OS subroutines are required.

---

9. Reentrancy

Reentrancy is a characteristic of library functions which allows multiple processes to use the same address space with assurance that the values stored in those spaces will remain constant between calls. Cygnus's implementation of the library functions ensures that whenever possible, these library functions are reentrant. However, there are some functions that can not be trivially made reentrant. Hooks have been provided to allow you to use these functions in a fully reentrant fashion.

These hooks use the structure `_reent` defined in `` `reent.h' ``. A variable defined as `` `struct _reent' `` is called a *reentrancy structure*. All functions which must manipulate global information are available in two versions. The first version has the usual name, and uses a single global instance of the reentrancy structure. The second has a different name, normally formed by prepending `` `_' `` and appending `` `_r' ``, and takes a pointer to the particular reentrancy structure to use.

For example, the function `fopen` takes two arguments, *file* and *mode*, and uses the global reentrancy structure. The function `_fopen_r` takes the arguments, *struct_reent*, which is a pointer to an instance of the reentrancy structure, *file* and *mode*.

There are two versions of `` `struct _reent' ``, a normal one and one for small memory systems, controlled by the `_REENT_SMALL` definition from the (automatically included) `` `<sys/config.h>' ``.

Each function which uses the global reentrancy structure uses the global variable `_impure_ptr`, which points to a reentrancy structure.

This means that you have two ways to achieve reentrancy. Both require that each thread of execution control initialize a unique global variable of type `` `struct _reent' ``:

1. Use the reentrant versions of the library functions, after initializing a global reentrancy structure for each process. Use the pointer to this structure as the extra argument for all library functions.

2. Ensure that each thread of execution control has a pointer to its own unique reentrancy structure in the global variable `_impure_ptr`, and call the standard library subroutines.

The following functions are provided in both reentrant and non-reentrant versions.

```
Equivalent for errno variable:
_errno_r

Locale functions:
_localeconv_r  _setlocale_r

Equivalents for stdio variables:
_stdin_r        _stdout_r       _stderr_r

Stdio functions:
_fdopen_r       _perror_r       _tempnam_r
_fopen_r        _putchar_r      _tmpnam_r
_getchar_r      _puts_r         _tmpfile_r
_gets_r         _remove_r       _vfprintf_r
_iprintf_r      _rename_r       _vsnprintf_r
_mkstemp_r      _snprintf_r     _vsprintf_r
_mktemp_t       _sprintf_r

Signal functions:
_init_signal_r  _signal_r
_kill_r         __sigtramp_r
_raise_r

Stdlib functions:
_calloc_r       _mblen_r        _setenv_r
_dtoa_r         _mbstowcs_r     _srand_r
_free_r         _mbtowc_r       _strtod_r
_getenv_r       _memalign_r     _strtol_r
_mallinfo_r     _mstats_r       _strtoul_r
_malloc_r       _putenv_r       _system_r
_malloc_r       _rand_r         _wcstombs_r
_malloc_stats_r _realloc_r      _wctomb_r

String functions:
_strdup_r       _strtok_r

System functions:
_close_r        _link_r         _unlink_r
_execve_r       _lseek_r        _wait_r
_fcntl_r        _open_r         _write_r
_fork_r         _read_r
_fstat_r        _sbrk_r
_gettimeofday_r _stat_r
_getpid_r       _times_r


Time function:
_asctime_r
```

```
        for (todo = 0; todo < len; todo++) {
            writechar(*ptr++);
        }
        return len;
    }
```

12.2 Reentrant covers for OS subroutines

Since the system subroutines are used by other library routines that require reentrancy, `libc.a' provides cover routines (for example, the reentrant version of `fork` is `_fork_r`). These cover routines are consistent with the other reentrant subroutines in this library, and achieve reentrancy by using a reserved global data block (see section Reentrancy).

`_open_r`
: A reentrant version of `open`. It takes a pointer to the global data block, which holds `errno`.

```
int _open_r(void *reent,
    const char *file, int flags, int mode);
```

`_close_r`
: A reentrant version of `close`. It takes a pointer to the global data block, which holds `errno`.

```
int _close_r(void *reent, int fd);
```

`_lseek_r`
: A reentrant version of `lseek`. It takes a pointer to the global data block, which holds `errno`.

```
off_t _lseek_r(void *reent,
    int fd, off_t pos, int whence);
```

`_read_r`
: A reentrant version of `read`. It takes a pointer to the global data block, which holds `errno`.

```
long _read_r(void *reent,
    int fd, void *buf, size_t cnt);
```

`_write_r`
: A reentrant version of `write`. It takes a pointer to the global data block, which holds `errno`.

```
long _write_r(void *reent,
    int fd, const void *buf, size_t cnt);
```

`_fork_r`
: A reentrant version of `fork`. It takes a pointer to the global data block, which holds `errno`.

```
int _fork_r(void *reent);
```

`_wait_r`

Writing Thread-Safe Stateful Code
=-=

* Wouldn't it be nice to not have to muck with re-entrancy structures? Wouldn't the Newlib code look cleaner without all those REENT_XXX() macros? Can't the toolchain help us by providing an *abstraction* that does the same thing as re-entrancy structures but with more syntactic sugar to make the code cleaner?

* C++11 is now a standard and has built-in threading/concurrency support:

  http://www2.research.att.com/~bs/C++0xFAQ.html#when-compilers

  At least the support is standardized in the *language*. Then there is the implementation...

      "It requires significant support from the linker (ld), dynamic linker (ld.so), and system libraries (libc.so and libpthread.so), so it is not available everywhere."

Thread-local storage

From Wikipedia, the free encyclopedia

In computer programming, **thread-local storage** (TLS) is static or global memory local to a thread.

This is sometimes needed because all threads in a process share the same address space. In other words, data in a static or global variable is normally always located at the same memory location, when referred to by threads from the same process. Variables on the stack however are local to threads, because each thread has its own stack, residing in a different memory location.

Sometimes it is desirable that two threads referring to the same static or global variable are actually referring to different memory locations, thereby making the variable thread local, a canonical example being the C error code variable `errno`.

If it is possible to make at least a memory address sized variable thread local, it is in principle possible to make arbitrarily sized memory blocks thread local, by allocating such a memory block and storing the memory address of that block in a thread local variable.

Contents

Windows implementation

One can use the API function **TlsAlloc** to obtain an unused *TLS slot index*. The *TLS slot index* will then be considered ‘used’.

Using the API function **TlsSetValue** you can write a memory address to a thread local variable identified by the *TLS index*. **TlsSetValue** can only affect the variable for the current thread.

Afterwards you can use the API function **TlsGetValue** to read the address from the variable identified by the *TLS index*.

When you're done with it you can call **TlsFree** to release the *TLS index*. Now it will be considered ‘unused’ so a new call to **TlsAlloc** can return it again.

Pthreads implementation

TLS with Pthreads (Thread Specific Data) is similar to **TlsAlloc** & co. for Windows. **pthread_key_create** creates a *key*, with an optional *destructor*, that can later be associated with thread specific data via **pthread_setspecific**. The data can be retrieved using **pthread_getspecific**. If the thread specific value is not *NULL*, the *destructor* will be called when the thread exits. Additionally, *key* must be destroyed with **pthread_key_delete**.

Language-specific implementation

Apart from relying on programmers to call the appropriate API functions, it is also possible to extend the programming language to support TLS.

Java

In Java thread local variables are implemented by the `ThreadLocal` (`http://java.sun.com/javase/6/docs/api/java/lang/ThreadLocal.html`) class. A `ThreadLocal` object maintains a separate instance of the variable for each thread that calls the object's `get` or `set` method. The following example (for J2SE 5.0 or later version of Java) illustrates using a `ThreadLocal` that holds an `Integer` object:

```
ThreadLocal<Integer> local = new ThreadLocal<Integer>();
```

the preceding code declares and instantiates the `ThreadLocal` object **`local`**. The following code gets the value for the currently executing thread. If the value existed, it is incremented and stored, otherwise the value is set to 1.

```
Integer val = local.get();
if (val == null)
    local.set(1);
else
    local.set(val + 1);
```

`local.get()` returns the current `Integer` object associated with the current thread, or null if no object has been associated with the thread. The code calls **`local.set()`** to set a new (or initial) value associated with the thread. (Note that the above example uses both generics and autoboxing—features added to Java in J2SE 5.0.)

Sun Studio C/C++ (http://developers.sun.com/sunstudio/) , IBM XL C/C++, GNU C & Intel C/C++

The keyword **__thread** is used like this:

```
__thread int number;
```

- **__thread** defines *number* to be a thread local variable.
- **int** defines the type of *number* to be of type **int**.

Visual C++

In Visual C++ the keywords **declspec(thread)** are used like this:

```
__declspec(thread) int number;
```

- **__declspec(thread)** defines *number* to be a thread local variable. (Can also be **__declspec(thread)**.)
- **int** defines the type of *number* to be of type **int**.
- **__declspec(thread)** works in DLLs only when those DLLs are bound to the executable, and will *not* work for those loaded with **LoadLibrary()** (a protection fault will occur)
- There are additional rules: "Rules and Limitations for TLS" (http://msdn2.microsoft.com/en-us/library/2s9wt68x.aspx) in MSDN.

Borland C++ Builder

In Borland C++ Builder the keywords **__declspec(thread)** are used like this:

```
__declspec(thread) int number;
```

the same in a more elegant way:

```
int __thread number;
```

- **__declspec(thread)** defines *number* to be a thread local variable. **__thread** is a synonym for **__declspec(thread)**.
- **int** defines the type of *number* to be of type **int**.

C# and other .NET languages

Static fields can be marked with ThreadStaticAttribute (http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadStaticAttributeClassTopic.asp) :

```
class FooBar
{
  [ThreadStatic] static int foo;
}
```

Also an API (http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadingThreadClassGetNamedDataSlotTopic.asp) is

available for dynamically allocating thread local variables.

Python

In Python version 2.4 or later **local** class in **threading** module can be used to create thread-local storage.

```
import threading
mydata = threading.local()
mydata.x = 1
```

External links

- ELF Handling For Thread-Local Storage (http://people.redhat.com/drepper/tls.pdf) — Document about an implementation in C or C++.
- ACE_TSS< TYPE > Class Template Reference (http://www.dre.vanderbilt.edu/Doxygen/Stable/ace/classACE__TSS.html#_details)
- RWTThreadLocal<Type> Class Template Documentation (http://www.roguewave.com/support/docs/hppdocs/thrref/rwtthreadlocal.html#sec4)
- Article "Use Thread Local Storage to Pass Thread Specific Data (http://www.c-sharpcorner.com/Code/2003/March/UseThreadLocals.asp) " by Doug Doedens
- "Paper (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1966.html) " by Lawrence Crowl

Retrieved from "http://en.wikipedia.org/wiki/Thread-local_storage"

Category: Computer programming
