


newlib-nano-rtx

Page last updated 30 Oct 2013, by  [Adam Green](#). [1 reply](#)

Newlib-Nano Multithreaded Support for RTX

These are my rough notes on what I figure needs to be done in my [gcc4mbed project](#) to make newlib-nano work better with RTX and its multithread support. While gcc4mbed also supports the use of the standard newlib library, I just plan to support RTX with newlib-nano at first since the nano version is the default and more popular option. I will try to also include notes here on what I see different for newlib. [Some notes on newlib-nano multithread support](#).

Syscall Reentrancy Support

Single and Multithreaded Syscalls

When retargeting newlib to different environments there are low level system calls, syscalls, such as `_open()`, `_close()`, `_read()`, `_write()`, etc. which provide the appropriate feature for each function. When building binaries with gcc4mbed, these syscalls are provided by the mbed SDK via [common/retarget.cpp](#) and from GCC_ARM's `libnosys.a`. The syscalls provided by these libraries are however the single threaded versions. For example when they set `errno`, they are setting the global `errno` variable and not a per thread version. It turns out that newlib-nano (and newlib) calls these single threaded versions of the routines through thread-safe routines which ends with a `_r` suffix. For example `_open_r` calls `_open`. These reentrant versions of the syscall routines take a pointer to a per-thread reentrant structure which contains amongst other things, a per thread `errno` field. The bulk of the newlib-nano (and newlib) code use the per-thread reentrant structure so the `_r` version of the functions just act as a thunk to call simpler single threaded versions of the routines which only know how to set the global variables (such as `errno`) and then copy that global state into the per-thread structure. This doesn't work in a multi-threaded environment since multiple of these syscalls could be made at the same time from multiple threads and the setting of the global state by one thread could be overwritten by another before it gets copied into the correct per-thread structure. It would therefore be better to implement the `_r` versions of syscall routines which currently modify global state and have them set the appropriate field in the per-thread reentrant structure instead. The following table walks through the syscalls I know get linked into gcc4mbed projects:

Syscall	Implementor	Notes
<code>_open</code>	mbed SDK	Should set <code>errno</code> .
<code>_close</code>	mbed SDK	Should set <code>errno</code> .

<code>_write</code>	mbed SDK	Should set <code>errno</code> .
<code>_read</code>	mbed SDK	Should set <code>errno</code> .
<code>_isatty</code>	mbed SDK	Should set <code>errno</code> .
<code>_lseek</code>	mbed SDK	Should set <code>errno</code> .
<code>_fstat</code>	mbed SDK	Uses <code>errno</code> .
<code>_sbrk</code>	mbed SDK	Uses <code>errno</code> .
<code>_chown</code>	libnosys	Ignore. Not used.
<code>_execve</code>	libnosys	Uses <code>errno</code> .
<code>_fork</code>	libnosys	Uses <code>errno</code> .
<code>_getpid</code>	libnosys	Shouldn't set <code>errno</code> at all.
<code>_gettimeofday</code>	libnosys	Uses <code>errno</code> .
<code>_kill</code>	libnosys	Uses <code>errno</code> .
<code>_link</code>	libnosys	Uses <code>errno</code> .
<code>_readlink</code>	libnosys	Ignore. Not used.
<code>_stat</code>	libnosys	Uses <code>errno</code> .
<code>_symlink</code>	libnosys	Ignore. Not used.
<code>_times</code>	libnosys	Uses <code>errno</code> .
<code>_unlink</code>	libnosys	Uses <code>errno</code> .
<code>_wait</code>	libnosys	Uses <code>errno</code> .

Some syscalls such as `_chown()` don't have a `_r` thunk but I also see nothing which calls these private routines from within newlib-nano and an application isn't really supposed to make calls to functions prefixed with an underscore as they are private. Another thing to note about the libnosys syscalls is that most of the libnosys.a syscall implementations just set `errno` to 0x58 (ENOSYS) and return -1.

I plan to copy the syscall implementations from the mbed SDK and libnosys into a gcc4mbed specific file, **retarget_gcc.c**, and modify them to use the reentrancy structure with the `_r` suffix.

newlib: *This same approach should work equally well for the standard newlib. It should also work to use this approach no matter if the code is single or multithreaded.*

RTX Support for Reentrant Calls

The reentrant functions described above will only work if each thread has its own reentrancy structure. By default newlib-nano just uses the single global `impure_data` structure defined in newlib's **impure.c** source file. For the ARM compiler, the `__user_perthread_libspace()` function exists in the RTX operating system to be called by the ARM C standard library whenever one of its functions needs access to its equivalent reentrant data. This function returns the structure which is appropriate for the currently running thread. The newlib libraries have a dynamic mode which can be enabled by the `__DYNAMIC_REENT__` macro to call a `__getreent()` function for a similar purpose but it isn't enabled in the versions which ship with the GCC_ARM compiler. Instead the newlib libraries expect to have the

global **_impure_ptr** always pointing to the reentrant data structure which is appropriate for the currently running thread. This means that I will need to add a struct **_reent** to each OS_TCB thread object and modify the GCC_ARM version of RTX's context switching code to point **_impure_ptr** to the correct **_reent** structure when switching between threads.

As I started implementing this change, I noticed that the reentrancy structure being allocated for each thread was much larger than the 100 bytes I expected for new lib-nano. There are two things I have found that appear to be related to this size increase:

- When building RTX code, the **_REENT_SMALL** macro should be defined so that the larger newlib definition isn't used. Turning this macro on from the compiler command line did manage to shrink the size down to 240 bytes.
- The public **sys/reent.h** header file installed by GCC_ARM is the one for newlib and not the one which has been customized for newlib-nano to further reduce the size. The following diff shows the differences between the two versions of the header. *I did talk to Joey Ye at the MBED summit and he indicated that the next major update of GCC_ARM will use the mainline version of newlib which includes most of the newlib-nano changes so this issue will go away at that time. I suspect that I won't get these changes ready to release before this update becomes available anyway.*

```
/depots/gcc-arm-none-eabi/src/newlib-nano-1.0/newlib/libc/include/sys$ diff /depots/gcc-arm-none-eabi/src/newlib/newlib/libc/include/sys/reent.h reent.h
```

```
81d80
< #ifdef _REENT_SMALL
83,90d81
< struct _atexit *_next;          /* next in list */
< int _ind;                      /* next index in this table */
< void (*_fns[_ATEXIT_SIZE])(void); /* the table itself */
< struct _on_exit_args *_on_exit_args_ptr;
< };
< #else
< struct _atexit {
< struct _atexit *_next;          /* next in list */
96d86
< #endif
214a205,206
> #if 1
> /* do not support wide-oriented stream, though _mbstate is kept. */
215a208
> #endif
268a262,263
> #if 1
> /* do not support wide-oriented stream, though _mbstate is kept. */
269a265
> #endif
397d392
< struct _atexit _atexit0;
429d423
< { _NULL, 0, { _NULL}, _NULL}, \
```

```

456,459d449
< (var)->_atexit0._next = _NULL; \
< (var)->_atexit0._ind = 0; \
< (var)->_atexit0._fns[0] = _NULL; \
< (var)->_atexit0._on_exit_args_ptr = _NULL; \
646d635
< struct _atexit _atexit0; /* one guaranteed table, required by ANSI */
702d690
< { _NULL, 0, { _NULL }, { { _NULL }, { _NULL }, 0, 0 } }, \
757,761d744
< (var)->_atexit0._next = _NULL; \
< (var)->_atexit0._ind = 0; \
< (var)->_atexit0._fns[0] = _NULL; \
< (var)->_atexit0._on_exit_args._fntypes = 0; \
< (var)->_atexit0._on_exit_args._fnargs[0] = _NULL; \

```

newlib: *This same approach should work equally well for the standard newlib.*

Synchronizing Heap Accesses (malloc/free)

The memory allocation routines implementation in newlib-nano include references to **MALLOC_LOCK** and **MALLOC_UNLOCK** macros. In the standard newlib library, these macros would be weak references to `__malloc_lock()` and `__malloc_unlock()` routines. A project like gcc4mbed could then provide implementation of these routines to use a mutex to serialize access to the heap. However, in newlib-nano these macros are `#define`'ed to nothing so this isn't possible.

To properly serialize access to the newlib-nano heap in gcc4mbed I will use the linker to wrap the **malloc_r()** and **free_r()** routines. The wraps will wait on the heap mutex, call the real heap routine, and then release the mutex before returning to the caller. This mutex can be initialized from within the `_start()` routine before any other library routines are called.

newlib: *Implement `__malloc_lock()` and `__malloc_unlock()` to utilize a mutex for synchronization since more than just `malloc_r()` and `free_r()` access the heap directly in that version of that library.*

Synchronizing File I/O Operations

I see 2 sets of locking routines used with respect to file operations in newlib-nano:

1. `__sfp_lock_acquire()` / `__sfp_lock_release()`
2. `_flockfile()` / `_funlockfile()`

Calls to the second set of routines are compiled **OUT** of newlib-nano while the state of the first set is a little bit more complicated. `__sfp_lock_acquire()` / `__sfp_lock_release()` are defined in **findfp.c** to be routines that just return without doing anything. From looking at the disassembly, I see routines like `fopen_r()` and `fclose_r()` actually making calls to these stub routines. However there are some important routines also defined in **findfp.c** such as `__sfp()` and `__sinit()` which should make similar

calls to these locking routines but they are not found in the disassembly. I am guessing that this happens because the compiler can optimize within this module to know that those locking routines have no impact.

To me it looks like there are a couple of issues with the `__sfp_lock_acquire()` / `__sfp_lock_release()` locking routines in newlib-nano (and probably standard newlib as well):

- I can't just use the linker to wrap the implementations of these routines since some have been optimized away. This could lead to corruption in the **_GLOBAL_REENT** structure where the global list of open FILE* structures is maintained since its protection in `__sfp` has been optimized away.
- These are large grain locks. It looks like all file I/O goes through a single lock when using these functions. The `_flockfile()` / `_funlockfile()` routines provide finer grain serialization since they just lock a particular FILE* object. So in theory if they hadn't been `#define`'ed away, they could be used instead of the `__sfp_lock_acquire()` / `__sfp_lock_release()` locks to allow parallel file access as long as it was to different files. However, I don't think that would work either because the `__sfp_lock_acquire()` / `__sfp_lock_release()` locks are the only thing serializing the open/close access to the **_GLOBAL_REENT** structure.

In general, newlib looks like it really only works with the big serialization hammer and using the finer grain serialization (per file) approach would still lead to potential concurrency problems.

Since I am more worried about robustness and less about performance when it comes to supporting these file handling routines, I think I will take the large grain approach. The easiest way to do this is to probably take advantage of the `__sfp_lock_acquire()` / `__sfp_lock_release()` stubs that are already called by most file routines in newlib-nano. I can wrap these routines with the linker and have them make use of a mutex for serialization. I would then wrap `fopen_r()` as well with the linker so that its whole execution is serialized by the same mutex. Without this wrap, `fopen_r()` makes the `_open_r()` syscall outside of the lock and this could lead to corruption of the file handle table maintained in the retargeting layer. To be safe, I would wrap `__sinit()` and `__sfp` and have them serialize on the same mutex as well. *This large grain approach also has the advantage that the low level file system drivers don't have to worry about concurrency issues.*

newlib: *I think the same approach should work with the standard newlib as well.*