Bionic Buffalo Tech Note #102

Using the Ivory Coast Memory Management Routines

last revised Friday 2003.05.23

Introduction

The Ivory Coast library is a collection of C language routines to make it easier to write portable code, and to provide some useful, general purpose functions and definitions for application development. Among other things, the library includes various memory management routines which are wrappers for `malloc()`, `free()`, and related other standard memory management functions. This Tech Note describes the ivory coast memory management routines, their design rationale, and explains how they are used.

The Ivory Coast library is available at *http://www.tatanka.com/prod/info/ivoire.html*.

Background and Design Rationale

The C language has long relied on `malloc()`, `free()` and related other routines for memory allocation, release, and ancillary functions. Some routines in the standard libraries, such as `strdup()`, return memory allocated using `malloc()`.

Although `malloc()` and `free()` are almost universally available, there are occasional environments (especially some embedded operating systems) which do not offer them. When they do exist, the behaviour isn't always the same from one platform to the next. (For example, calling `free(NULL)` sometimes works, and sometimes doesn't.) Some environments which do have `malloc()`, `free()`, and the other procedures defined in the standard, also offer other procedures which are not in the standard and are not always available on other platforms. Some of these are wrappers around standard routines, offering additional functionality (such as tracing or process failure in case sufficient memory isn't available).

Bionic Buffalo wanted memory allocation functions that offered some of these extended functions, plus others, in a portable way, and could not find an existing API that met its requirements. It developed an API, which evolved into the current Ivory Coast memory management API. Older forms of the API were found in the Morocco and Mongolia libraries, but with the A-series revisions the API is now found in the Ivory Coast library.

The Ivory Coast memory management API has the following features:

> It is built as a wrapper around the standard `malloc()` and `free()` routines, or alternatively can be built on top of other, proprietary memory allocation functions.
>
> It allows for multiple regions from which to allocate buffers. This can be useful to distinguish among heap, shared, adapter, kernel, and other distinct areas in some environments.
>
> It will, if asked, clear memory when allocated (for consistent initialization) or when released (for security). Also, it will optionally terminate the application in case of insufficient memory or other errors.
>
> It provides various debugging and trace features, including canaries for overflow checking.
>
> It allows the allocator to associate a small value with each buffer, which can be used by the application or the developer.
>
> It provides a function to interrogate the size of any memory buffer, and a function to resize (re-allocate) a buffer.

The API is now used in a wide variety of environments, including network protocol engines, object brokers, compilers, utilities, and end-user applications.

In the beginning, the API design changed as understanding of the problems evolved. Now, however, it is stable, and is a candidate for optimization by use of in-line code.

The Memory Block Prefix

To implement the API, it was necessary to associate information with each allocated block of memory. This could have been done at least two ways: keep a database of blocks with the additional values, or add a prefix or suffix to each block. The prefix mechanism was chosen.

Before each allocated block is a prefix, which should be considered opaque to the application. When the caller requests memory, the library allocates a larger block than requested. The additional memory is used to contain a prefix (along with possible pad bytes for alignment), which is a fixed length from the beginning of the user buffer area. The pointer returned to the user is offset from the actual beginning of the block by the size of the prefix, and the application need not be aware of the size of the prefix. In fact, the application should not depend on any specific prefix size, since it may vary from platform to platform, or from version to version.

When an application passes back its pointer to the library, the library works backward to the beginning of the prefix from the application's pointer.

When canaries are requested, then the library also allocates a suffix after the user area. In the current implementation, the suffix consists only of the canary. (In future implementations, planned for typed or tagged memory, the suffix may also contain type data or other varying length information.)

Allocating and Releasing Memory

To obtain memory, the application calls

```
civ_status_T    civ_memory_allocate
                        ( size_t            size,
                        uint8_t             region,
                        void             ** pointer,
                        civ_flags_T         flags ) ;
```

The `size` is the amount of memory requested by the application. It does not include any prefix or suffix size.

The `region` is the pool from which memory is to be allocated. In the default implementation, the only meaningful region is `CIV_MEMORY_REGION_HEAP`, which obtains the buffer using `malloc()`. In other versions, special regions may be defined. For example, a driver sharing a memory buffer with a network card might define `CIV_MEMORY_REGION_ADAPTER`.

Currently, there are eight user flags and five library flags defined.

The user flags are the least significant eight bits of the `flags`, and may be set in any combination entirely at the discretion of the application. The user flags are kept with the block (in the prefix), and may be retrieved at any later time by `civ_memory_flags_get()`. The application might (for example) use them to remember the purpose or type of data in a block, the task or program which allocated the block, or whether the block was locked or otherwise put to some special use.

The library flags modify the behaviour of the memory allocation API.

> `CIV_MEMORY_FLAG_CLEAR_ALLOCATED` causes the library to zero the new block before returning it to the caller. This is similar to the effect of using `calloc()` instead of `malloc()`.
>
> `CIV_MEMORY_FLAG_CLEAR_RELEASED` will cause the library to zero the block when (and if) it is finally released. The main intended purpose for this is security: if the buffer will be used to contain confidential information, then zeroing it will make it more difficult for an attacker to acquired the contents.
>
> `CIV_MEMORY_FLAG_PANIC_NO_MEMORY` will cause the library to terminate the process by calling `exit()` in case insufficient memory is available to satisfy the request.

Before such a termination, an appropriate error message will be printed. This eliminates the need to check returned pointers for `NULL`: when this flag is set, the routine will never return a `NULL` pointer.

`CIV_MEMORY_FLAG_PANIC_OTHER` will cause the library to terminate the application in case it detects corruption in the memory allocation data structures. When corruption is detected, then there probably is a programming bug, and the error probably is not recoverable. Termination is usually the only reasonable option under normal circumstances. The only reasons an application might set this flag are to perform some special cleanup operations before terminating itself, or during development to track down the cause of the error.

`CIV_MEMORY_FLAG_USE_CANARY` will cause the library to place a canary immediately following the user buffer. No software should ever write to the canary location. When the block is released, or at some other time as requested by special subroutine call (`civ_memory_validate_*()`), the canary value will be checked. If the canary value has changed since allocation, then there is a strong likelihood that the application has written beyond the end of its buffer, and memory is presumed corrupted.

Once allocated, a block may be reallocated using

```
civ_status_T    civ_memory_allocate
                        ( void          ** pointer,
                        size_t             new_size,
                        civ_flags_T        flags ) ;
```

This causes a new block to be allocated in the same region as the original block, and the contents of the old block will be copied to the new block. The old pointer is overwritten by the new pointer, and the old block is released. This is similar to calling `realloc()`, except for the features defined by the flags. The new block may be larger or smaller than the old block. If the block is expanded and `CIV_MEMORY_FLAG_CLEAR_ALLOCATED` is used, then the additional area is zeroed.

Memory is returned using

```
civ_status_T    civ_memory_release
                        ( void           * pointer,
                        civ_flags_T        flags ) ;
```

In all of the above calls, the success or failure of the call is indicated by the `civ_status_T` return value. Normally, the only reasons for failure are corrupted memory, bad parameters, or insufficient memory.

Obtaining and Modifying User Flags

An application can determine the values of the user flags associated with a specific block by calling the procedure

```
civ_status_T    civ_memory_flags_get
                        ( void            * pointer,
                        civ_flags_T       * flags ) ;
```

The flags value returned will contain the user flags as the least significant eight bits. The values of other bits are not defined. The returned value can be masked by ANDing it with `CIV_MEMORY_FLAG_USER_VALUE_MASK`. If desired, the user flags can be modified and rewritten to the block by calling

```
civ_status_T    civ_memory_flags_set
                        ( void            * pointer,
                        civ_flags_T         flags ) ;
```

When `civ_memory_flags_set()` is called, none of the `flags` bits above the least significant eight should be set. Otherwise, the consequences are unpredictable.

Checking Memory

An application can check a single block of memory for integrity using

```
civ_status_T    civ_memory_validate_user_block
                        ( void            * pointer,
                        civ_flags_T         flags ) ;
```

This will verify the correct value of the canary (if any), and do some consistency checking on the prefix.

By default, there is no way to check all of the blocks together, because the library doesn't keep a master list of allocated blocks. However, if the library is built with the `CIV_MEMORY_KEEP_CHAIN` flag set, then the library adds extra fields to the prefix to keep a linked list of all allocated blocks. In that case,

```
civ_status_T    civ_memory_validate_all
                        ( civ_flags_T       flags ) ;
```

can be called to check every allocated block. (The value of `CIV_MEMORY_KEEP_CHAIN` is defined in the ivory coast header files.)

There is an additional routine defined by the library, `civ_memory_validate_system_block()`, which is used internally and should not be called by applications.

In the current version of the library, the use of `civ_memory_validate_all()` and `CIV_MEMORY_FLAG_KEEP_CHAIN` is not thread safe.

Using the Memory Routines in Applications

The Ivory Coast library, and other libraries, use the memory allocation routines internally. Often, there is no explicit way to set the memory allocation flags for calls such as `civ_string_duplicate()`. However, the memory allocation routines refer to the global variable

```
civ_flags_T    civ_memory_flags_default ;
```

during their operation. If any of the five library flags or eight user flags are set in this variable, then they are applied to the calls made by the library or the application. For example, an application which executes the statement

```
civ_memory_flags_default |= CIV_MEMORY_FLAG_CLEAR_ALLOCATED ;
```

will cause all subsequent memory allocations to zero the user blocks before returning to the caller.

Remember that the pointers returned or used by the `civ_memory_*()` routines are incompatible with the pointers returned or used by `malloc()`, `free()`, and other standard library routines. This includes standard routines such as `strdup()`, which allocates a buffer using `malloc()`. An application using the Ivory Coast memory procedures should avoid such libary routines; instead, it should use compatible procedures such as `civ_string_duplicate()`. , which allocate their buffers using the `civ_memory_*()` procedures.
