How to Debug Memory Issues with MOCA_DMALLOC

Introduction

This page explains debugging dynamic memory issues in MOCA applications and components.

MOCA provides a conditional wrapper around the malloc( ), calloc( ), realloc( ) and free( ) system calls that can be enabled by setting the MOCA_DMALLOC environment variable. When this variable is set appropriately, MOCA intercepts these system calls and maintains a list of all nodes of memory allocated along with information regarding each node of memory. Using this information, it can detect most issues related to dynamic memory, including leaks, overwrites and double frees.

There are a couple of caveats regarding this, however.

- MOCA can either maintain a fixed-size list of memory nodes allocated, or it can dynamically grow its list. If you choose to use the dynamic allocation option, the memory debugger itself will become susceptible to heap corruption. If you suspect heap corruption, you should use a fixed-size allocation limit. By default, the number of memory allocations that are supported at a single point in time of 16,276 nodes. If an application attempts to allocate more than 16,276 nodes at any one time, it will crash. This number can be overridden if necessary (see below).
- There is a significant amount of overhead involved.

Documentation

There are a number of options available in the MOCA_DMALLOC variable. The options consist of comma-separated values, some of which are mutually exclusive.

- `on` and `off` – The default is `off`, which means that any MOCA_DMALLOC setting must include the `on` option if it is to cause memory debugging to operate.
- `deadbeeflen` – This option specifies the length of the "dead beef" value, which is written at the end of each allocated memory block. The greater this value, the more likely memory overwrites will be detected. A value of 0 will prevent dmalloc from being able to detect any memory overwrites. The default value is 4. The disadvantage of setting this option to a greater value is that it increases the size of every dynamically allocated memory block by that many bytes, however, increasing the value dramatically (such as deadbeeflen=512) can be useful for detecting particularly nasty overwrites (the sort in which the overwriting doesn't occur immediately at the end of a memory block, but rather skips into middle of the next memory block).
- `nomemset` – This option keeps dmalloc from memsetting malloced and freed memory blocks using the "dead beef" string. The disadvantage of using this option is that it prevents dmalloc from interfering with a program that assumes malloc regions are initialized or that use information
  stored in a block after it has been freed. If a program bombs only without this option, it is probably inadvertently using an uninitialized or freed memory block. The advantage of using this
  option is that the application should run faster if it uses dynamic allocation heavily. The complement of this option is `memset`, whichis the default.
- `time` – This option causes dmalloc to track the allocation time of each memory block rather than a call sequence number for the given allocation function. The disadvantage of using this option is that the application will run slower if it uses dynamic allocation heavily. The advantage of specifying this option is only that, while debugging, the time stamps might help to cross-referencing allocations with events specified in a time-stamped log file. The complement of this option is "notime", which is the default.
- `nofreenull` – This option causes dmalloc to complain if the application passes NULL to free. This is different from the ANSI-C defined behavior which is to allow NULL as the argument to free with no effect. The complement of this option is "freenull", which is the default.
- `noabort` – This option causes dmalloc to not abort and create a core dump on UNIX or crash dump on Win32. By default, if a malloc( ), calloc( ) or realloc( ) call fails, the application immediately exits, creating a core/crash dump.
- `maxalloc` – This option sets up the maximum (static) number of allocations that dmalloc will support. Once there have been more than this many memory allocations, it is assumed that the memory has been used up.
- `dynalloc` – This option tells dmalloc to support dynamic memory allocation. The size of the allocation list to track will grow dynamically as memory is allocated. This may cause problems if significant heap corruption occurs (it uses the heap). If problems occur with this option enabled, run in static (maxalloc) allocation mode, instead.
- `trackmalloc`, `trackrealloc`, `trackcalloc` – These options cause dmalloc to perform an abort on the n'th call to malloc, calloc or realloc. This can aid in debugging memory overwrites by forcing a core dump on the exact call to malloc, calloc, realloc that you know causes an overwrite.

The most complete documentation currently exists within the source code the implements the debugging malloc functionality. The following is a copy of that documentation:

```
/***************************************************************************
****
 *
 *                          D O C U M E N T A T I O N
 *
 * Depending on the runtime options specified in the MOCA_DMALLOC
environment
 * variable, this function will report each of the currently allocated
memory
 * blocks including their addresses, sizes, times of allocation, and
whether or
 * not they have been overwritten.
 *
 * OPTIONS
 *
 *     Turning Options On and Off
 *
 *         Runtime options to the dmalloc package should be specified in an
 *         environment variable called MOCA_DMALLOC.
 *         The options should be comma seperated.
 *
 *         Examples:
 *
 *             unix> export MOCA_DMALLOC=off
 *             unix> export MOCA_DMALLOC=dynalloc
 *             unix> export MOCA_DMALLOC=nomemset
 *             unix> export MOCA_DMALLOC=nomemset,deadbeeflen=100
 *             unix> export MOCA_DMALLOC=trackmalloc=100
 *
 *     off
 *
 *         This option turns off dmalloc at runtime so that it need not be
 *         compiled out.  Basically, the dmalloc package will do nothing
more
 *         than call the native allocation routines.  With this option, the
 *         application should run nearly as fast as with dmalloc compiled
out.
 *         The complement of this option is "on", which is the default.
 *
 *     nomemset
 *
 *         This option keeps dmalloc from memsetting malloced and freed
memory
 *         blocks using the "dead beef" string.  The disadvantage of using
this
 *         option is that it prevents dmalloc from interfering with a
program
 *         that assumes malloc regions are initialized or that use
information
 *         stored in a block after it has been freed.  If a program bombs
only
 *         without this option, it is probably inadvertently using an
 *         uninitialized or freed memory block.  The advantage of using
this
 *         option is that the application should run faster if it uses
dynamic
 *         allocation heavily.  The complement of this option is "memset",
which
 *          is the default.
 *
 *      deadbeeflen
 *
 *          This option specifies the length of the "dead beef" value,
which is
 *          written at the end of each allocated memory block.  The greater
this
 *          value, the more likely memory overwrites will be detected.  A
value
 *          of 0 will prevent dmalloc from being able to detect any memory
 *          overwrites. The default value is 4.  The disadvantage of
setting
 *          this option to a greater value is that it increases the size of
 *          every dynamically allocated memory block by that many bytes,
however,
 *          increasing the value dramatically (such as deadbeeflen=512) can
be
 *          useful for detecting particularly nasty overwrites (the sort in
 *          which the overwriting doesn't occur immediately at the end of a
 *          memory block, but rather skips into middle of the next memory
block).
 *
 *      time
 *
 *          This option causes dmalloc to track the allocation time of each
 *          memory block rather than a call sequence number for the given
 *          allocation function.  The disadvantage of using this option is
that
 *          the application will run slower if it uses dynamic allocation
 *          heavily.  The advantage of specifying this option is only that,
 *          while debugging, the time stamps might help to cross-
referencing
 *          allocations with events specified in a time-stamped log file.
The
 *          complement of this option is "notime", which is the default.
 *
 *      nofreenull
 *
 *          This option causes dmalloc to complain if the application
passes
 *          NULL to free.  This is different from the ANSI-C defined
behavior
 *          which is to allow NULL as the argument to free with no effect.
The
 *          complement of this option is "freenull", which is the default.
 *
 *      noabort
 *
 *          This option causes dmalloc to not abort and create a core dump
on
 *          UNIX or crash dump on Win32.  By default, if a malloc( ),
 *         calloc( ) or realloc( ) call fails, the application immediately
 *         exits, creating a core/crash dump.
 *
 *     maxalloc
 *
 *         This option sets up the maximum (static) number of allocations
 *         that dmalloc will support.  Once there have been more than this
 *         many memory allocations, it is assumed that the memory has been
 *         used up.
 *
 *     dynalloc
 *
 *         This option tells dmalloc to support dynamic memory allocation.
 *         The size of the allocation list to track will grow dynamically
 *         as memory is allocated.  This may cause problems if significant
 *         heap corruption occurs (it uses the heap).  If problems occur
 *         with this option enabled, run in static (maxalloc) allocation
 *         mode, instead.
 *
 *
 *     trackmalloc
 *     trackcalloc
 *     trackrealloc
 *
 *         These options cause dmalloc to perform an abort on the n'th call
 *         to malloc, calloc or realloc.  This can aid in debugging memory
 *         overwrites by forcing a core dump on the exact call to malloc,
 *         calloc, realloc that you know causes an overwrite.
 *
 * DEBUGGER
 *
 *     The function misDebugReport is usually invoked automatically at
exit.
 *     However, it may be useful to call it from within within a source
level
 *     debugger (such as dbx) in this way: p misDebugReport("", 0)
 *
 * WARNING
 *
 *     To accurately report the status of dynamic allocations, all code
 *     that calls calloc, malloc, realloc or free should be compiled with
 *     MOCA_DEBUG_MALLOC defined in mocaconfig.h.
 *

***************************************************************************
***/
```

Finding Memory Issues

The following steps can be used to determine if any memory issues exist in a MOCA component. The general idea here is simply to execute the component in question with the debugging malloc functionality turned on and look for messages on stderr regarding and leaks, overwrites or bad addresses. **NOTE**: These examples are given for Windows environments, but the same basic steps are involved on other platforms.

All memory leaks will begin with the message "MEMORY LEAK". All memory overwrites will begin with the message "MEMORY OVERWRITE". All bad addresses will begin with the message "BAD ADDRESS?". Bad addresses are usually associated with two calls to the free( ) system call with the same address.

1. Set up a development environment.

```
c:\> env.bat
```

2. Set the MOCA_DMALLOC environment variable.

```
c:> set MOCA_DMALLOC=on
```

3. Start a standalone server process.

```
c:\> mocasrvprc -p8000
```

4. Start a MSQL session and execute the component in question.

```
c:\> msql -p8000
MSQL> run my component
MSQL> /
```

5. Stop the server process.
6. Any memory leaks, overwrites or bad addresses will be displayed in a message to stderr.

Resolving Memory Issues

Bad addresses and memory overwrites usually involve finding the piece of memory that was overwritten, determining what it was overwritten by, and fixing the offending code.

Memory leaks require a different approach. The general idea is to force a crash on the malloc( ), calloc( ) or realloc( ) system call based off of the leak list provided in the steps above. After the crash occurs, the call stack can be examined to find who requested the memory and, from that point, the developer should be able to determine what needs to be done to resolve the issue.

1. Set up a development environment.

```
c:\> env.bat
```

2. Set the MOCA_DMALLOC environment variable with the correct "track" argument.

```
c:> set MOCA_DMALLOC=on,trackmalloc=1012
```

3. Start a standalone server process.

```
c:\> mocasrvprc -p8000
```

4. Start a MSQL session and execute the component in question.

```
c:\> msql -p8000
MSQL> run my component
MSQL> /
```

5. When the server process crashes, choose to debug it.
6. Fix the issue in the code.

Dynamic Allocation and Performance

In more recent versions of MOCA-based products (2005.1 and newer), there is a new option, *dynalloc*. This option no longer requires that you configure the maximum number of allocations to track. As described above, though, it has the caveat that it is more susceptible to heap corruption, while the static allocation option (maxalloc=x) is more or less immune to it.

Also, in recent versions, the malloc debugger has much better performance than in past versions. As such, it is possible to run a full instance of WM, for instance, with MOCA_DMALLOC turned on and only notice a slight performance degradation. In older versions, the performance degradation would be too great and it would be impossible to do any long-term debugging (e.g. in production) to find memory leaks.