Memory Leak Detection in Embedded Systems

Cal discusses mtrace, dmalloc and memwatch--three easy-to-use tools that find common memory handling errors.

by Cal Erickson

One of the problems with developing embedded systems is the detection of memory leaks; I've found three tools that are useful for this. These tools are used to detect application program errors, not kernel memory leaks. Two of these tools (mtrace and dmalloc) are part of the MontaVista Linux Professional Edition 2.1 product. The other (memwatch) is available from the Web (see Resources).

C and C++ programmers control dynamic memory allocation. Reckless use of this control can lead to memory management problems, which cause performance degradation, unpredictable execution or crashes.

Some of the problems that cause memory leaks are writing or reading beyond an allocated memory segment or trying to free memory that has already been freed. A memory leak occurs when memory is allocated and not freed after use, or when the pointer to a memory allocation is deleted, rendering the memory no longer usable. Memory leaks degrade performance due to increased paging, and over time, cause a program to run out of memory and crash. Access errors lead to data corruption, which causes a program to behave incorrectly or crash. When a program runs out of memory it also can cause the Linux kernel to crash.

Designing and programming an embedded application requires great care. The application must be robust enough to handle every possible error that can occur; care should be taken to anticipate these errors and handle them accordingly--especially in the area of memory. Often an application can run for some time before it mysteriously crashes itself or the system as a result of a memory allocation that is never freed. Finding these errors can be done through use of memory leak detectors.

These tools work by replacing malloc, free and other memory management calls. Each tool has code that intercepts calls to malloc (and other functions) and sets up tracking information for each memory request. Some tools implement memory protection fences to catch errant memory accesses.

Some of the leak detection programs are very large and require a virtual memory image of the program being searched. This requirement makes it very difficult to use on embedded systems. However, mtrace, memwatch and dmalloc are simple programs that find most errors.

All three tools were run on one example C program containing common memory handling errors. This program, together with Makefiles for building it with the three tools, is available as a downloadable file at ftp.ssc.com/pub/lj/listings/issue101/6059.tgz. All of these tools have been used in several different target architectures. The example code will work whether compiled natively or cross-compiled.

mtrace

The simplest of the three tools is mtrace. A feature of the GNU C library, mtrace allows detection of memory leaks caused by unbalanced malloc/free calls. It is implemented as a function call, mtrace(), which turns on tracing and creates a log file of addresses malloc'd and freed. A Perl script, also called mtrace, displays the log file, listing only the unbalanced combinations and--if the source file is available-- the line number of the source where the malloc occurred. The tool can be used to check both C and C++ programs under Linux. One of the features that makes mtrace desirable is the fact that it is scalable. It can be used to do overall program debugging but can be scaled to work on a module basis as well.

Key to the use of the mtrace feature are three items: include mcheck.h, set the MALLOC_TRACE environment variable and call the mtrace() function call. If the MALLOC_TRACE variable is not set, mtrace() does nothing.

The output of mtrace includes messages such as:

```
- 0x0804a0f8 Free 13 was never alloc'd
/memory_leak/memory_leaks/mtrace/my_test.c:193
```

to indicate memory that was freed but never malloc'd and a ``Memory not freed" section that includes the address, size and line number of calls to malloc for which no free occurred.

memwatch

memwatch is a program that not only detects malloc and free errors but also fencepost conditions. Fencepost conditions occur when writing data into an allocated chunk of memory (allocated by malloc) and the data goes beyond the end of the allocated area. Some things that memwatch does not catch are writing to an address that has been freed and reading data from outside the allocated memory.

The heart of memwatch is the memwatch.c file. It implements the wrappers and code for the address checking. To use memwatch the file memwatch.h must be included in the source. The variables MEMWATCH and MW_STDIO must be defined on the compile command line (-DMEMWATCH and -DMW_STDIO). The memwatch.c file must be used with the application as well. The object module from the compile of memwatch.c must be included in the link of the application. Upon execution of the application, a message will appear on stdout if memwatch found any abnormalities. The file memwatch.log is created that contains the information about the errors encountered. Each error message contains the line number and source-code filename where the error occurred.

Comparing memwatch.log with the log from mtrace, the same errors are reported. The memwatch tool also found a fencepost condition where the memory addresses were changed to overwrite the start and end of an allocated area, showing the expanded capability of memwatch in this case. The disadvantage is that memwatch is not scalable. It has to run on the whole application.

dmalloc

The third tool is a library that is designed as a drop-in substitute for malloc, realloc, calloc, free and other memory management functions. It provides runtime configurability. The features of the tool provide memory leak tracing and fencepost write detection. It reports its errors by filename and line number and logs some general statistics. This library, created and maintained by Gray Watson, has been ported to many operating systems other than Linux.

The package is configurable to include thread support and C++ support. It can be built both as shared and static libraries. All of these options are selected when building the tool. The result is a set of libraries that are used when linking the application program. There is an include file (dmalloc.h) that needs to be included in the source of the application to be checked. In addition to the library and include file, it is necessary to have an environment variable set up that dmalloc reads to configure how it checks and where it puts the logging information. The following line is the setup used with the test program for dmalloc:

```
export \
DMALLOC_OPTIONS=debug=0x44a40503,inter=1,log=logfile
```

What this means is 1) log is a file named logfile in the current directory, 2) inter is the frequency for the library to check itself and 3) debug is a hex number whose bits select the types of checking to do. This

example tests for just about every possible error. The following is a list of the tests and the corresponding bits to set in ``debug'':

- none (nil): no functionality (0)
- log-stats (lst): log general statistics (0x1)
- log-non-free (lnf): log non-freed pointers (0x2)
- log-known (lkn): log only known non-freed (0x4)
- log-trans (ltr): log memory transactions (0x8)
- log-admin (lad): log administrative info (0x20)
- log-blocks (lbl): log blocks when heap-map (0x40)
- log-bad-space (lbs): dump space from bad pointers (0x100)
- log-nonfree-space (lns): dump space from non-freed pointers (0x200)
- log-elapsed-time (let): log elapsed time for allocated pointer (0x40000)
- log-current-time (lct): log current time for allocated pointer (0x80000)
- check-fence (cfe): check fencepost errors (0x400)
- check-heap (che): check heap adm structs (0x800)
- check-lists (cli): check free lists (0x1000)
- check-blank (cbl): check mem overwritten by alloc-blank, free-blank (0x2000)
- check-funcs (cfu): check functions (0x4000)
- force-linear (fli): force heap-space to be linear (0x10000)
- catch-signals (csi): shut down program on SIGHUP, SIGINT, SIGTERM (0x20000)
- realloc-copy (rco): copy all re-allocations (0x100000)
- free-blank (fbl): overwrite freed memory space with BLANK_CHAR (0x200000)
- error-abort (eab): abort immediately on error (0x400000)
- alloc-blank (abl): overwrite newly alloced memory with BLANK_CHAR (0x800000)
- heap-check-map (hcm): log heap-map on heap-check (0x1000000)
- print-messages (pme): write messages to stderr (0x2000000)
- catch-null (cnu): abort if no memory available (0x4000000)
- never-reuse (nre): never reuse freed memory (0x8000000)

- allow-free-null (afn): allow the frees of NULL pointers (0x20000000)
- error-dump (edu): dump core on error and then continue (0x40000000)

If the library needs to check C++ programs, a source file named dmalloc.cc is needed with the application. This module provides wrapper functions for new to malloc and delete to free. The GNU debugger GDB can be used with dmalloc. A file is included with the product that can be used as part of a .gdbinit file so that GDB is set up automatically to know about dmalloc.

Along with the library is a utility named dmalloc that will programmatically set up the DMALLOC_OPTIONS variable. I've made a setup script that is sourced prior to running the program to be debugged. This way the setup is repeatable without errors.

This article only covers the general use of the tool, but the documentation for it is extensive and details more available features. The test program used with the earlier tools was run using DMALLOC. The results can be lengthy and optionally include the bytes present at critical areas of memory such as the ``fence-top", where a pointer overruns a malloc'd area. The end of the log file includes statistics, addresses, block sizes and line numbers for occurrences of malloc without free.

The three tools provide varying support for memory leak detection and reporting. Each of these tools has been used on a Linux workstation as well as cross-compiled and executed on several different target architectures. In one application, developers used all three tools. **mtrace** was used to find a memory leak in a third-party C++ library where an exception throw/catch block caused a major leak. The dmalloc tool was used to find memory leaks in the execution of Linux pthreaded applications. The memwatch tool was used to catch a buffer pool mechanism that was not properly defragmenting itself. These tools are small and easy to implement and remove when debugging is completed.

The example program consists of one source file, my_test.c. There are three separate directories that contain a README, Makefile and a script to run the test [available at ftp.ssc.com/pub/lj/listings/issue101/6059.tgz]. In the dmalloc test, the environment setup script is also available. Each of the tests has been built on Red Hat and SuSE Linux releases, as well as the MontaVista Linux cross-development environments.

Resources

Cal Erickson *(cal_erickson@mvista.com) currently works for MontaVista Software as a senior Linux consultant. Prior to joining MontaVista, he was a senior support engineer at Mentor Graphics Embedded Software Division. Cal has been in the computing industry for over 30 years with experience at computer manufacturers and end-user development environments.*