TotalView

WHITE PAPER

Finding Memory Leaks and Errors in Parallel Applications

Introduction

Parallel applications are common everywhere. Nowhere is it more noticeable then for companies streaming video or using web-based applications that are always presenting different content based on user requests. In these industries, applications get requests from many users and each one is serviced by a process or thread. The number of threads and processes is based on the number of users on the system at any given time. Many times, threads get created and destroyed, however in some cases the memory stays allocated. Other times applications have their threads and processes stick around. These long running processes can cause major memory leaks which multiply by the size of the data feed. Memory bugs, a mistake in the management of heap memory, lead to rapid slowdowns and even crashes of the system.

Memory bugs are hard to track down because a variety of different situations cause them including, overwriting the stack, not freeing the heap, or freeing it at the wrong time. It's difficult to trace back to a specific line of code because it can take a long time to manifest. Overall, memory bugs can have a disastrous effect on an applications usability. Having to boot everyone off servers periodically and restart doesn't suffice when people rely on these servers. Luckily, there are solutions and we explain the root cause and how to resolve these bugs in this paper.

Factors Leading to Memory Bugs

Memory bugs can occur in any program and are caused by several factors:

- Failure to check for error conditions.
- Relying on nonstandard behavior
- Failure to free memory.
- Dangling references.
- Array bounds violations
- Memory corruption.

These cause programs to crash and generate incorrect random results, or lurk in the code base for long periods of time — only to manifest themselves at the worst possible moment.

Leaks are difficult to track down even on a single application running on a desktop. They are much more vexing when encountered on a distributed, parallel system. Developers write parallel programs for situations with large problem sets, so the program naturally ends up loading a significant amount of data and using a lot of memory.

Classifying Memory Errors

Programmers need to pay attention to heap memory because programs explicitly manage heap memory rather than implicitly at compile or run time. There are several ways that a program fails to properly use heap memory. We'll be describing this in terms of the C malloc() API. However, similar errors can be made using the C++ new and Fortran90 allocate statements.

MALLOC ERRORS

Malloc errors occur when a program passes an invalid value to one of the operations in the Heap Manager API. This could happen by copying the value of a pointer into another pointer, and then at a later time, both pointers are passed to free().

DANGLING POINTERS

A dangling pointer references previously deallocated memory. Any memory access through a dangling pointer leads to undefined behavior. Programs with dangling pointer bugs may appear to function without any obvious errors.

MEMORY BOUNDS VIOLATIONS

Individual memory allocations returned by malloc() represent discrete blocks of memory with defined sizes. Any access to memory immediately before the lowest address, or after the highest address in the block of memory results in undefined behavior.

READ-BEFORE-WRITE ERRORS

Reading memory before initialization is a common error. Many compilers identify reads before initialization for local variables. Detecting reads before initialization of memory through a pointer is much more difficult. Dynamic memory is affected, since it is always accessed through a pointer.

Detecting Memory Leaks

Leaks occur when a program finishes using a block of memory and discards all references to the block but fails to call free. The impact of the leak depends on the nature of the application. In some cases, the effects are minor; in others, where the rate of leakage is high enough or the runtime of the program is long enough, leaks significantly change the performance of the program. This makes leaks all that much more annoying, since they often linger in otherwise well understood code. It is challenging to manage dynamic memory in complex applications and ensure that allocations are released exactly once so that leak errors do not occur.

It's hard to define "ceasing to use a memory block" but an advanced memory debugger executes leak detection by seeing if the program retains a reference to specific memory locations. If it can't be referenced it should have been freed. To detect memory leaks, you have to watch the malloc() call to the operating system. A common method for tracking the malloc() is by adding instrumentation code into the application that replaces the malloc(). The main disadvantage of using this approach is it means you need to recompile your application which may lead to differences in behaviors or time spent fixing compilation issues. The other issue is that the instrumentation code may lead to applications slowdowns.

Interposition is another method to watch malloc(). An interposition library engages the application at runtime to insert itself between the user's application code and the malloc() subsystem. The interposition library defines functions for each of the memory allocation functions. These functions get called by the program whenever it allocates or frees a block of memory. It then passes the call to the underlying operating system. Using this method, the malloc() call still works normally and the application doesn't need to be re-compiled.

DETECTING HEAP BOUNDS VIOLATIONS

Blocks are often contiguous with other blocks of program data. Therefore, if the program writes past the end of an array, it usually overwrites the contents of some other unrelated allocation. When the program is re-run, the ordering of allocations may differ and the overwriting occurs in a different way. This leads to extremely frustrating "racy" bugs that manifest in different ways. Sometimes they cause the program to crash, sometimes result in bad data, and sometimes turn out to be completely harmless.

MemoryScape Debugger

A feature with in TotalView, MemoryScape is an interactive, dynamic memory analysis, and debugging tool that reduces time spent on memory debugging. It has a lightweight architecture that requires no recompilation and modest impact on the runtime performance of the program.

MemoryScape is designed for use with parallel applications, providing both detailed information about individual processes, and the high-level memory usage statistics across all the processes. It includes support for launching and attaching to all processes of a parallel job, the ability to memory debug many processes from within one GUI and do script-based debugging to use batch environments.

ARCHITECTURE

MemoryScape uses the interposition method to detect memory issues. Its library is called the Heap Interposition Agent (HIA). The primary reason to choose the interposition method is because it provides lightweight memory debugging. The runtime performance of a program being debugged performs similarly to when the HIA is absent. This is critical for many applications, in which a heavyweight approach might make the runtime

of programs exceed the patience of developers or even change the effects because of a major lag effect in using the data.

PARALLEL ARCHITECTURE

MemoryScape uses a behind-the-scenes, distributed parallel architecture to manage runtime interaction with the user's parallel program. MemoryScape starts lightweight debugging agent processes, which run on the nodes of the cluster where the user's code is executing. These processes are responsible for the low-level interactions with the individual local processes and the HIA module that is loaded into the process being debugged. The processes communicate directly with the MemoryScape front-end.

COMPARE MEMORY STATISTICS

Many applications have expected memory usage behaviors. They may be structured so that all the nodes allocate the same amount of memory or some similar pattern. If such a pattern is expected or if the user wishes to examine the set of processes to look for patterns, MemoryScape has a memory statistics window that provides memory usage statistics in several graphical forms (line, bar, and pie charts) for all or a subset of the processes. Users can select the set of processes that they wish to see statistical information about. The view represents the state of the program at that point in time. The debugger process controls drive the program to a new point in execution and then update the view to look for changes. If any processes look out of line, the user can look at the detailed status of the heap memory.

LOOK AT HEAP STATUS

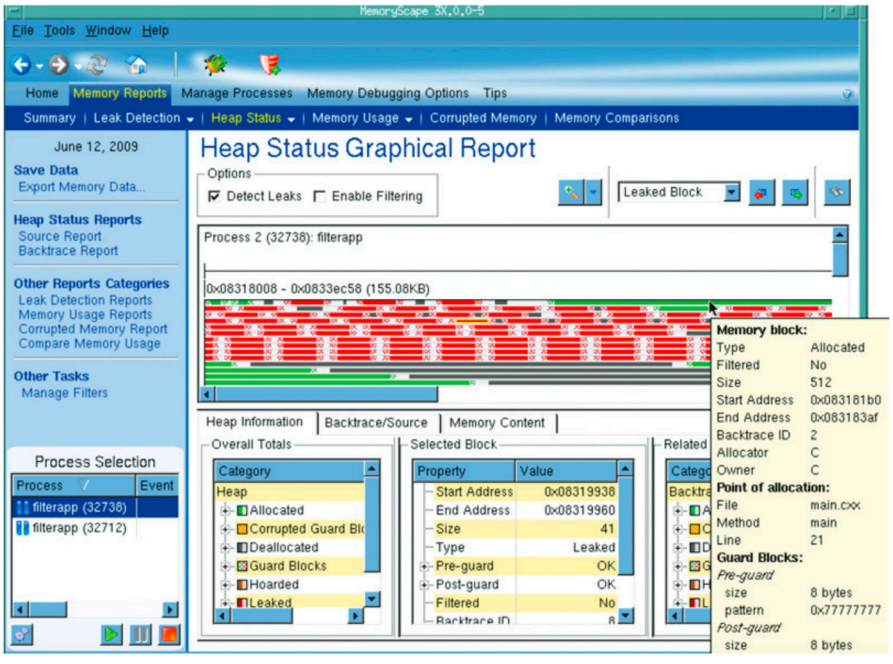
MemoryScape provides a wide range of heap status reports. Whenever a process has been stopped, a user can obtain a view of the heap. This gives the user a great way to see the composition of the program's heap memory at a glance. The view is interactive; selecting a block highlights related allocations and presents the user with detailed information about both the selected block and the full set of related blocks. Users can filter the display to dim allocations based on properties such as size or the shared object they were allocated in.

DETECT LEAKS

MemoryScape performs heap memory leak detection by generating a leak report. The resulting report lists all the heap allocations in the program for which there are not any valid references. A block of memory, that the program is not storing a reference to anywhere, is a leak. Users can observe leaks in the heap graphical display.

DETECT HEAP BOUNDS VIOLATIONS

MemoryScape provides supports for guard blocks, which get allocated before and after heap memory. Since this bit of memory is not part of the allocation, the program



MemoryScape GUI provides an interactive view of the heap. Colors indicate the status of memory allocations.

should never read or write to that location. The HIA can initialize the guard blocks with a pattern and check the guard blocks for changes. Changes mean that the program wrote past the bounds of the allocation.

USING MEMORYSCAPE INSIDE TOTALVIEW

Memory debugging data files can be loaded by the memory module of the TotalView source code debugger. This allows user to share powerful debugging techniques, taking advantage of having memory debugging and access to all the variables and state data.

In Summary

Memory bugs can occur in any program. These types of bugs are often a source of great frustration for developers because they can be introduced at any time and are caused by several factors. They lurk in a code base for long periods of time and tend to manifest in several ways.

This makes memory debugging a challenging task. Commonly used development tools and techniques are not specifically designed to solve memory problems and can make the process of finding and fixing memory bugs an even more complex process.

TotalView's MemoryScape feature is an easy-to-use memory debugging tool that helps developers identify and resolve memory bugs. Its specialized features, including the ability to compare memory statistics, look at heap status, and detect memory leaks, make it uniquely well-suited for debugging these parallel and distributed applications.

See for yourself how TotalView can identify and resolve your runtime issues faster.

REQUEST A FREE TRIAL

totalview.io/free-trial

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle. Our portfolio including solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static & dynamic code analysis, version control, and more. With over 20,000 customers, Perforce is trusted by the world's leading brands to drive their business critical technology development. For more information, visit www.perforce.com.