shalloc: Private-by-Default Threads

Abstract

Using a heap that is shared amongst threads has safety and security implications. shalloc tackles this problem by dividing the heap accessible to each thread into a private and a shared part. It takes the form of a library that a programmer can leverage to easily implement this paradigm.

Introduction

The concurrent execution paradigm that is most commonly used at the moment is POSIX threads. Threads are easy to use and reason about, and have very little overhead. However, they end up sharing too much. The process's entire heap is visible to all the threads. Any change by any one thread to a part of the heap is immediately visible to all the other threads. While this behavior makes it easy to share information amongst threads, it becomes tough to isolate problems. An off-by-one write in one thread could potentially affect all the other threads. A segmentation fault caused by one thread causes the entire program to crash.

Further, it is difficult to isolate the different parts of the program. A thread which handles sensitive information writes to the same heap that a thread that interfaces with the world can read from. This creates the opportunity for security vulnerabilities.

The obvious solution then is to blow threads up into separate processes. Each process has a separate heap. The problem then is that there is no elegant way of communicating between processes. The usual IPC mechanisms are too cumbersome and require careful exchange of set up information amongst processes before they can be used.

Let us take shared memory [1] as an example. Processes that use shared memory to communicate have to negotiate a key for each shared memory segment they use. Further, there is no unified model that processes can use to determine where each object is stored in the shared memory segment. Dealing with all this does not help the application developer focus on the problem.

This paper discusses an elegant solution for this problem.

The Solution

Each process has access to 2 different heaps: a private heap that no other process has access to, and a shared heap that is accessible to all the processes in the group. By default, new objects are allocated memory from the private heap. However, it is possible for the programmer to specify that an object should be shared amongst other processes in the group. The object is then allocated memory from the shared heap and is immediately accessible to all processes in the group.

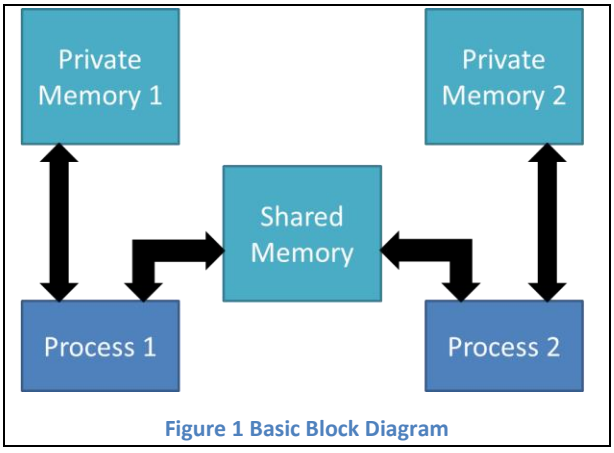

Figure 1 Basic Block Diagram

Implementation Details

A library called shalloclib is defined that hides the details of the implementation from the programmer. It comprises of two classes called SharedClass and MemPool, as well as utility functions that help the programmer mirror a thread based workflow. The programmer writes code as if it is a multi-threaded program. The importatant factor the programmer has to keep in mind is that only the portions of the heap

that need to be shared should be marked as shared. This marking is done by extending the SharedClass.

shalloclib::MemPool::MemPool

shalloclib has MemPool objects, each of which represent a memory pool of a different block size. Currently, three block sizes - 64, 128 and 512 bytes, are specified by default. For each MemPool object, the constructor creates a shared memory segment and logically divides it into blocks. The size of each block and the number of such blocks in each pool is configurable statically. The shared memory segment is then mapped using mmap [2]. While doing this, the MAP_SHARED [2] flag is set, so that it is visible to child processes. Further, as the MemPool objects themselves are created at the beginning of execution, they are visible to all the child processes.

If necessary, it is trivial to extend the library to support more such memory pools of different sizes. At present, the user needs to have a rough idea of what the application requires, and all these decisions have to be taken at compile time.

To manage each shared memory segment, an array of booleans is used. This helps to keep track of which blocks are allocated and which are free. A bitmap could have been used here, but that is just a speed vs. space tradeoff. The array itself is located at the beginning of the shared memory segment, and thus can be used by any of the processes that have access to the shared memory segment.

shalloclib::MemPool::getBlock

This gets a free block from this memory pool and also sets the status flag corresponding to this block as used. If there are no free blocks, it displays an error message and exits.

shalloclib::MemPool::destroyBlock

This sets the status flag corresponding to this block as free. It does not need to do anything else as the next allocation will automatically use this block if necessary.

shalloclib::MemPool::addrWithinRange

This is a utility function that checks to ensure that the given address falls within the purview of the memory pool under consideration. It also makes sure that the address is the beginning address of a block.

shalloclib::SharedClass

An object which wants to be allocated shared memory extends this class. The new and delete operators of this class are overridden. SharedClass also has an element called lock, which gives the programmer a locking option for the whole class.

In order for locks to work across processes, the PTHREAD_PROCESS_SHARED [3] attribute should be associated with them. The lock that comes with the SharedClass object automatically has this, and thus, can be used to get exclusive access to the object across processes. The programmer, of course, is free to devise other locks that suit the need at hand, as long as they work across processes.

shalloclib::SharedClass::operator new

The new operator is overridden so that memory is allocated from shared memory. This function tries to find the optimal block to allocate. It compares the size of the required object and tries to allocate from that memory pool whose block size is just able to satisfy the requirement. If that pool is exhausted, it tries the memory pool with the next greater block size and so on. It fails only if there is no block free that can satisfy the requirement.

shalloclib::SharedClass::operator delete

This function is relatively straightforward. From the block address, it first finds out the actual memory pool from which this block was allocated, finds out the entry in the array of booleans corresponding to the memory pool, and modifies it to indicate a free location.

The address supplied must be the beginning address of a block. While it is straightforward to allow an address that points to the middle of the block to delete the block, more often than

not, such a scenario indicates a bug in the program, and hence, the library does not permit such deletions.

shalloclib::sthread_create

This function is used to create a new "thread." It accepts a function pointer and an argument pointer and spawns a new process using the fork system call. In the child, the function is called with the given argument.

The shared memory segment is visible to the child process too, because the MAP_SHARED flag was set when the shared memory segment was 'mmap'ed. Thus, any changes to the shared memory segment made by the child process are immediately visible to the parent process and vice versa.

As the array of booleans for storing the free status of blocks is also stored in the shared memory segment, the child process is able to allocate new objects, and as long as a reference to the new object is stored in some place in the shared memory that the parent knows about, the parent will be able to use the new object.

shalloclib::sthread_join

sthread_join allows a "thread" to wait for another "thread." Instead of pthread_t [4] objects, all that is needed is the process ID of the process. Internally, sthread_join just calls wait_pid [5] on the process.

shalloclib::sthread_cancel

sthread_cancel allows a "thread" to be killed. It accepts the process ID of the target process as a parameter. Internally, it sends SIGTERM [6] to the process.

Discussion

Performance is one of the most significant factors to look at. The replacement of threads with processes involves the creation of a separate address space for the child processes. This involves a definite space overhead. However, one thing to note is that in UNIX, the fork system call [7] has copy-on-write semantics by default. Thus, when a child process is formed, "the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child." [7]

The second factor to consider is that by partitioning memory, false sharing is eliminated in the private segment of the heap. "False sharing occurs when processors in a shared-memory parallel system make references to different data objects within the same coherence block (cache line or page), thereby inducing unnecessary coherence operations." [8] Previously, on the heap, objects which were shared and objects which were not meant to be shared were interleaved, and there was nothing preventing them from being on the same cache line. However, with the private segment of the heap separated out, other than at the segment boundaries, there will be no false sharing. False sharing does remain a problem in the shared memory segment.

The library achieves the goal of sharing only what the programmer wants to share. Further, because each spawned process handles only the signals received by them, a segmentation fault affects only the process that caused it and the other processes continue to execute.

Results

Performance results have not been obtained yet, but will be published in the full version of this paper.

There are test cases for correctness of the various features present along with the code.

Related Work

Grace [9] and DThreads [10] use the threads as processes paradigm, but while Grace does not support communication between threads, DThreads "merges" the heap together at synchronization points. shalloc offers a middle ground – it is possible to communicate between threads and the programmer is also able to specify which parts are shared.

Future Work

Objects like strings and vectors allocate their own memory. So, if a string or a vector is part of a shared object, modifications to them are opaque to the other processes. Custom memory allocators should be designed for every such container in order for them to work properly in this approach so that all memory allocated for them come from shared memory.

The shared memory segments used in this approach are finite. Future versions of the library should allow for the dynamic expansion of the shared memory space when it is exhausted. Instead of allocating one of the bigger blocks (which is terribly inefficient), the library should instead create a new shared memory segment and make it visible to all the processes. An approach used in the UNIX file system, with multiple layers of indirect blocks, can be used. When the control information itself is present in the shared memory segment, it is easy to chain together additional shared memory segments as needed. Existing tools like Hoard [11] and Heap Layers [12] can be modified to achieve this goal.

Threads allow a lot of fine tuning, especially via the use of thread attributes. The aim of the library should be to replicate all those features while using the process model.

Performance results are needed before any claims can be made in that area, and will be the top priority in the full version of this paper.

Conclusion

shalloc allows the programmer to concentrate on the actual algorithms, rather than be distracted by peripheral concerns such as how the sharing of memory can be implemented between processes. It allows the programmer to control exactly what objects are visible to other threads and what objects are not.

References

[1] Michael Kerrisk, http://man7.org/linux/man-pages/man3/shm_unlink.3.html, 2013.

[2] Michael Kerrisk, http://man7.org/linux/man-pages/man2/mmap.2.html, 2013.

[3] Michael Kerrick, http://man7.org/linux/man-pages/man3/sem_init.3.html, 2012.

[4] Michael Kerrick, http://man7.org/linux/man-pages/man7/pthreads.7.html, 2010.

[5] Michael Kerrick, http://man7.org/linux/man-pages/man2/wait.2.html, 2012.

[6] Michael Kerrick, http://man7.org/linux/man-pages/man7/signal.7.html, 2012.

[7] Michael Kerrick, http://man7.org/linux/man-pages/man2/fork.2.html, 2013.

[8] William J. Bolosky and Michael L. Scott, False Sharing and Its Effect on Shared Memory Performance, 1993.

[9] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark, Grace: Safe Multithreaded Programming for C/C++, 2009.

[10] Tongping Liu, Charlie Curtsinger, and Emery D. Berger, DTHREADS: Efficient and Deterministic Multithreading, 2010.

[11] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson, Hoard: A Scalable Memory Allocator for Multithreaded Applications, 2000.

[12] Emery D. Berger, Benjamin D. Zorn, and Kathryn S. McKinley, Composing High-Performance Memory Allocators, 2001.