AUTOMATYKA/ AUTOMATICS • 2013 • Vol. 17 • No. 1

http://dx.doi.org/10.7494/automat.2013.17.1.29

Andon Coleman*, Janusz Zalewski**

A Study of Real-time Memory Management: Evaluating Operating System's Performance

1. Introduction

1.1. Traditional Run-time Memory Allocation

Early operating systems used to measure the run-time memory usage of a process by adding the length of its code, data, and other segments. The very first implementations of the C libraries kept track of all blocks of memory allocated by growing and shrinking the data segment at run-time. While no requirement is placed on how the standard library must allocate storage, it is generally understood to use a heap structure.

Traditional memory allocation mechanisms are for multiple reasons inadequate. Due to fragmentation, it is possible that the data segment contains enough free space to satisfy a request, but not enough contiguous space. Then, the data segment grows since the new memory will reside at the end of the heap. The C standard library [1] does not include any mechanism for garbage collection or compaction. Freeing memory does not necessarily shrink the data segment, as it addresses the need to stay the same to maintain pointer consistency.

The problem with the flat addressing approach is that the amount of allocated memory is directly related to the highest address a program references. In other words, without compaction, empty spaces sandwiched between reserved blocks of memory in a program's data segment are unusable by other programs. Simple flat addressing implementations may give malicious processes the ability to modify data belonging to another process or kernel.

To solve the problems associated with flat addressing, modern multi-tasking operating systems tend to use protected virtual memory (for example [2]). A program reserves its memory in fixed-size blocks (pages) and maps them to the addresses in such a way that one process' address space is completely isolated from another. Because memory is not

* Student of Computer Science, Florida Gulf Coast University, Ft. Myers, FL 33965, USA
** Professor, Computer Science and Software Engineering, Florida Gulf Coast University, Ft. Myers, FL 33965, USA

addressed relative to the data segment's base address, the run-time memory requirements are not related to the largest address in use. When memory is freed, empty pages are released and the kernel can give the empty pages to other processes without requiring compaction.

1.2. Related Work

It must be noticed that virtual memory does not come cheap. Special hardware and software is required to implement translation from logical to physical memory addresses, protected-mode operating systems must perform mode switches whenever they allocate or release pages, and memory is allocated in blocks of one or more page at a time. In addition, sharing memory between processes or accessing DMA-mapped regions requires the use of special APIs such as Posix or Win32 to alter the memory map when using virtual memory.

Modern operating systems allocate virtual memory by reserving enough pages to fill a user-requested amount of space. Under this scheme, the kernel is not required to map these pages to physical memory until the first page fault (reference to an unmapped address) occurs. This concept, better known as demand paging, improves allocation performance by deferring complicated kernel bookkeeping tasks until reserved regions of memory are actually used. Page faults require a special hardware interrupt to handle, and will be discussed later.

Demand paging leads to multiple measures of process memory consumption. The amount of reserved memory (also known as working set or virtual size) refers to the sum of all allocated pages, while the amount of resident memory (also known as resident set or physical size) refers to the number of pages actually mapped to a system memory. Allocated memory belonging to unreferenced pages, or pages swapped out of system memory, is not included in the tally of resident memory. Likewise, on systems that implement demand paging, the kernel also keeps track of the number of page faults on a per-process basis.

As a result, each call to `malloc()` may result in multiple system calls and/or interrupts on i386/Linux (a protected-mode virtual memory-based platform), depending on whether or not the requested allocation will fit into one or more existing heap blocks (pages). Additionally, a standard library does not place strict requirements on the alignment or slack space of allocated memory blocks. The only alignment requirement `malloc()` stipulates is that it will return an address aligned to the host architecture's word size [3]. Memory fragmentation and paging also differ from one implementation to another, making it difficult to predict the expense of calling `realloc()`. In the worst case, `realloc()` may have to physically relocate memory by reading chunks of memory from one location and writing to another.

In addition to practical work on memory allocation, there have been multiple academic papers published in this area, for example [4–7], discussing allocation schemes and memory models for real-time Java [4], bare-bones machine memory allocation and scheduling [5], allocation across space, time and a characterization of memory usage [6], and attempting defragmentation [7]. They have high research value on their own, but none of them addresses actual implementation in commercial operating systems that are used or can be used in real-time applications.

1.3. Addressing the Problems

In general, all platforms provide more advanced techniques for managing memory than the standard library offers. Clearly, the C standard library allocates memory with portability, rather than performance in mind. High performance applications often allocate buffers with slack space (e.g., if a buffer must store 13 elements, to allocate enough space to store 16 before requiring re-allocation) to satisfy data alignment requirements and reduce fragmentation. Given the added expense of system calls and protected mode switches, some applications even avoid the standard library's memory management system altogether, in favor of more sophisticated libraries and lower-level routines.

Rather than finding a way to make the standard library more efficient, the focus should be on the development of new and/or the modification of existing algorithms to improve memory management performance and creating the software necessary to evaluate performance. Principally, the algorithms should address the key hardware and software concepts discussed above; this could involve directly interfacing with kernel memory functions, reducing the number of required system calls, and researching placement schemes more advanced than the fragmentation prone "first-fit."

The objective of this paper is to conduct experiments to study selected currently used memory allocation mechanisms in the most common operating systems, and compare them with a real-time kernel. The rest of the paper is structured as follows. First, the research methodology is explained briefly, which is followed by the description of actual experiments and the analysis of results. The paper ends with a conclusion.

2. Methodology

Successful real-time memory management requires algorithms that have consistent timing properties and that ensure determinism. The goal of any real-time software is to meet time deadlines; in the context of memory management, this is achievable by focusing on two key concepts. The first is how well the algorithm deals with simultaneous allocation requests, measured in terms of wait time and synchronization overhead. Measuring thread efficiency is straightforward, as it only requires thread-level timing.

The second factor affecting real-time memory management relates to the amount of run-time and storage an allocator uses establishing optimal address space distribution. Measuring location-based performance requires the spatial analysis of allocated blocks of memory, the empty spaces in-between and the size and structure of record keeping data.

2.1. General Timing Considerations

Timing is a fundamental part of any performance analysis software, however, the most general purpose high-precision timers measure the wrong thing. While the timers provide adequate resolution, their drawback is that they relate to the system clock or the CPU clock rate. This makes them inappropriate for measuring the performance of a multi-tasking program; the ideal solution involves communicating with the scheduler.

In operating systems, usually scheduling has to do with the distribution of CPU and I/O resources on a per-process basis, with threads using the allocated CPU time for concurrent operations. However, in the real-time OS, the usual process/thread paradigm is modified. VxWorks schedules tasks and threads rather than processes. Thus, it makes more sense to measure performance on a per-thread basis. The benefits are two-fold: the amount of CPU time a thread receives is measurable in most OS's, and modern hardware architectures are shifting from powerful single-core processors to lightweight multi-core processors.

Traditionally, `malloc()` and alternatives were developed before parallel computing became the norm, and thread-safety was never a major factor. Whether or not `malloc()` is safe in a multi-threaded application varies and some platforms even require compiling and linking a program against a special thread-safe version of the C standard library [1]. In most thread-safe implementations, locking is used as a safeguard mechanism. However, there are special algorithms designed to eliminate the contention for memory addresses, such that locking is not required. Timing at the thread-level will ensure fair performance evaluation of thread-optimized algorithms vs. traditional lock-based algorithms.

2.2. Evaluating Address Space Distribution

The efficiency of a memory manager depends on more than just the amount of time to acquire and release memory resources. Other factors include fragmentation and unusable space, and the frequency with which resizing a block of memory requires moving the entire block. Depending on the application, CPU or I/O time may be less important than the effective utilization of all memory. Even when performance is of major concern, reducing the number of required relocations should directly translate to a reduction in CPU runtime.

To evaluate these performance criteria, the test suite should perform record keeping of its own on a per-memory block basis including extra information pertinent to performance evaluation. While memory allocators already perform internal record keeping, implementations such as ANSI C do so non-extensibly. The only reasonable way to achieve the goals above is to duplicate said functionality.

2.3. Design of a Benchmark

The performance evaluation software must be able to identify holes in the allocated address space, and calculate the number of times a block of memory moved to satisfy allocation requests. Because many virtual memory architectures use paging, the reallocation of blocks with sizes that are not multiples of the system's page size(s) will be an important part of the test procedure. Last, it must also test the performance of memory allocation algorithms in threaded and non-threaded access patterns, using various block sizes. Optional performance tests could include address space compaction and garbage collection.

Many micro-benchmarking approaches focus on the number of transactions per--second using random sequences of allocation. To get a reasonable approximation of real-world performance using a random sequence of tests requires a very large number of allocations. Worse, an algorithm cannot be tuned for specific applications when the input is random. To solve this problem, memory manager projects often instrument applications and analyze run-time behavior after termination using a profiler to validate performance.

To address embedded systems and make testing multiple memory manager implementations easier, a tool with a special translation layer has been developed, which mimics the traditional ANSI C interface for functions such as `malloc()`, `free()` and `realloc()`. While the interface an application developer sees is nearly identical, the internal translation layer converts memory requests to the necessary formats for proprietary memory managers. The developed API is referred to as TLM; TLM is not officially an acronym, but it could stand for *Transaction Logged Malloc*, *TransLated Malloc*, etc.

2.4. CPU Timing in Posix

Adopting the Posix API allows for the unification of measurements [8]. Rather than measuring the time that has elapsed since some arbitrary instant, Posix provides special timers known as CPU timers. They represent the total time the OS has spent running a process or thread. The biggest difference between the elapsed vs. run time approach is that CPU timers do not include the time a process or thread spends sleeping or waiting for I/O.

Depending on the way thread-safety is implemented in a memory manager, the software may have to block (wait) until other threads finish their memory operations. Done properly, the waiting thread should utilize the lowest overhead locking mechanism possible. Using Posix CPU timers, that is, measuring the amount of CPU time a memory manager requires vs. the amount of time an operation takes to complete, gives a good indication of its suitability in multi-threaded environments.

Table 1
CPU Timers used by TLM

Operating System	CPU Timer	Resolution
Windows	`GetThreadTimes()`	0.1 μs

Linux and VxWorks	`pthread_getcpuclockid()`
	`clock_gettime()`
		1 Hz*
		*Variable,
		see: `clock_getres()`

Table 1 lists the interfaces TLM uses for thread-level CPU timing. While POSIX timers are standard in Linux and VxWorks, not all Posix-compliant platforms guarantee their existence. MacOS X supports only a Posix subset; to perform thread-level timing on MacOS X requires directly interfacing with the Mach kernel.

3. Timing Experiments

TLM was designed to measure more than raw throughput and execution time, it also measures memory consumption, thread-level memory activity and can analyze the layout of allocated memory. The following sections present the results of tests necessary to measure and analyze the real-time performance characteristics discussed above.

Thread contention arises when multiple threads attempt to modify the state of memory allocation concurrently. Thus, it is necessary to measure the amount of time a thread spends sleeping (waiting) using basic per-thread CPU timers. TLM defines the amount of wait time as follows, where T represents a single thread, T_{start} is the time a thread began, and T_{user} and T_{kernel} measure the amounts of time spent in user- and kernel-mode, respectively:

$$T_{lifetime} = T_{current} - T_{start}$$

$$T_{wait} = T_{lifetime} - (T_{user} + T_{kernel})$$

In a multi-tasking OS, the amount of time a process/thread spends sleeping varies depending on factors such as the number of processes and priorities. While processor load may cause a sleeping thread to sleep longer, the ratio of time spent executing vs. waiting over multiple test runs is adequate to analyze a memory allocator's lock and wait behavior.

In general, a real-time memory allocator must have low wait-time variability and must bias user-mode time over kernel-mode time. There is no prize for finishing a job early in real-time applications, but finishing late has serious consequences – real-time algorithms need to reduce the worst-case expense and aim to stabilize the average case. Likewise, switching between user-mode and kernel-mode in a real-time application adds significant overhead, and must be kept to a minimum. Therefore, average wait time and user/kernel-mode switch frequency are the two most important timing metrics TLM measures.

Formally, an allocator x is more suitable in real-time applications than another allocator y, if it has consistently shorter wait times and spends less time on kernel-mode operations. However, many dedicated real-time OS's have highly customizable schedulers. Thus, depending on operating system configuration, wait time consistency may play a smaller role in determining real-time performance than context switch frequency.

The following subsections provide a comprehensive summary of results for some combinations of platforms and algorithms supported by TLM. For reasons of space allocation, all grapahs are placed at the end of this section, after the narrative.

With the exception of VxWorks, all of the graphs below measure four aspects of runtime behavior. The left graph on the top represents the ratio of kernel- to user-mode execution time (smaller is better). The right graph on the top row represents the amount of time the system's scheduler suspends each thread – the series is always non-decreasing, apparent jitter in this graph is the result of kernel-level inaccuracies on some platforms. From left to right on the second row, the graphs measure user-mode and kernel-mode times respectively.

3.1. Hoard on Mac OS X and Linux

Hoard was one of the first thread-optimized memory allocators on the market, its development is seriously lagging behind newer alternative memory allocators and thus its performance is consistently several orders of magnitude worse on all platforms [9]. Hoard performs considerably better on Linux than it does on Mac OS X, in large part due to its lower kernel overhead (top left graph in Fig. 1 and 2). Sleep time looks very sporadic on Linux (top right graph in Fig. 2), but comes down to jitter in the Linux kernel's thread run-time reporting.

3.2. Jemalloc on Mac OS X, Linux and Windows

Results of running jemalloc [10] for Mac OS X (Fig. 3) exemplify what makes a memory allocator real-time. Extremely low kernel overhead is attributed to leaving allocated memory empty, which allows demand-paging kernels to defer some operations until memory is first used. User-mode time is linear and wait times are very consistent. Overall jemalloc is very well suited for real-time applications on this OS.

Jemalloc behaves consistently on Linux (Fig. 4). Jitter in sleep time is due to kernel-level issues. No official port of jemalloc exists for Windows, so TLM was derived from a port of jemalloc for use in the Mozilla Firefox browser. Behavior is similar to Mac OS X and Linux, the step-function like growth in runtime is attributed to granularity and the unusual way that Windows records scheduler history (Fig. 5).

3.3. System Default Allocator: Linux, Windows and VxWorks

Interestingly, the default memory allocators in GNU/Linux and Microsoft Windows both incur greater kernel overhead allocating 4 KiB blocks than 16 KiB blocks (Fig. 6 and 7). The Visual C++ 2008 allocator (Fig. 7) on Windows does a poor job allocating empty blocks of memory, and locking behavior (top right graph in Fig. 7) also hinders concurrency.

TLM is unable to distinguish user- from kernel-mode time on VxWorks, and the amount memory on the test hardware is less than 256 MiB. As such, the test had to be significantly scaled down. The results on the left side of Figure 8 show that VxWorks' default allocator achieves better than O(n) overhead when locating free blocks. The right side of Figure 8 shows that kernel-heavy threads in a user-mode real-time process can trip up the real-time scheduler (eliminating all concurrency). Presently, the only solution for memory allocation on VxWorks is the default allocator – the results of testing it suggest room for improvement, and porting `jemalloc` and other allocators to VxWorks would make sense.

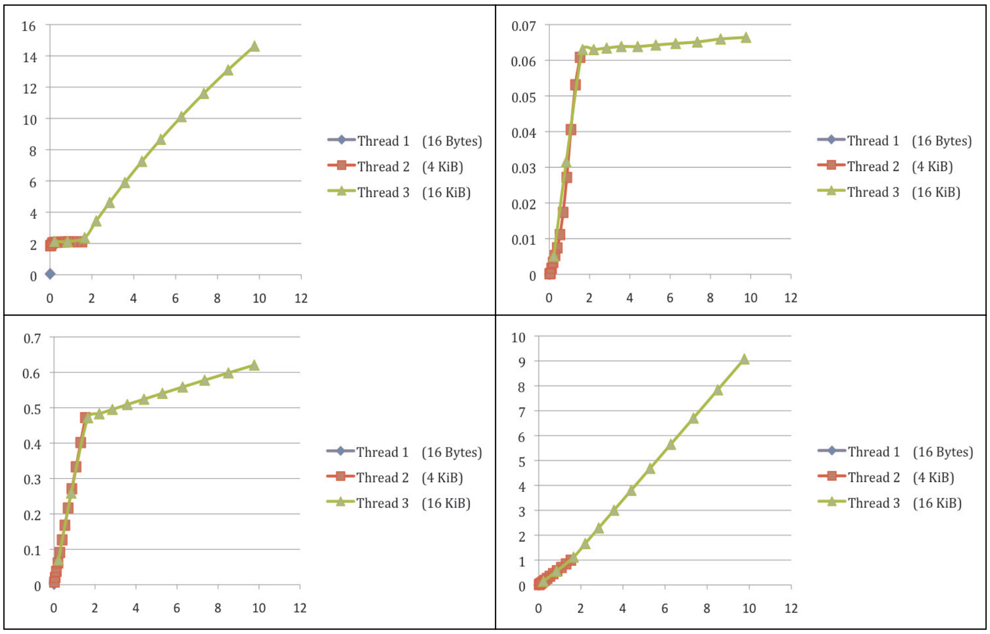

Fig. 1. Experimental results using Hoard on Mac OS X 10.6.8

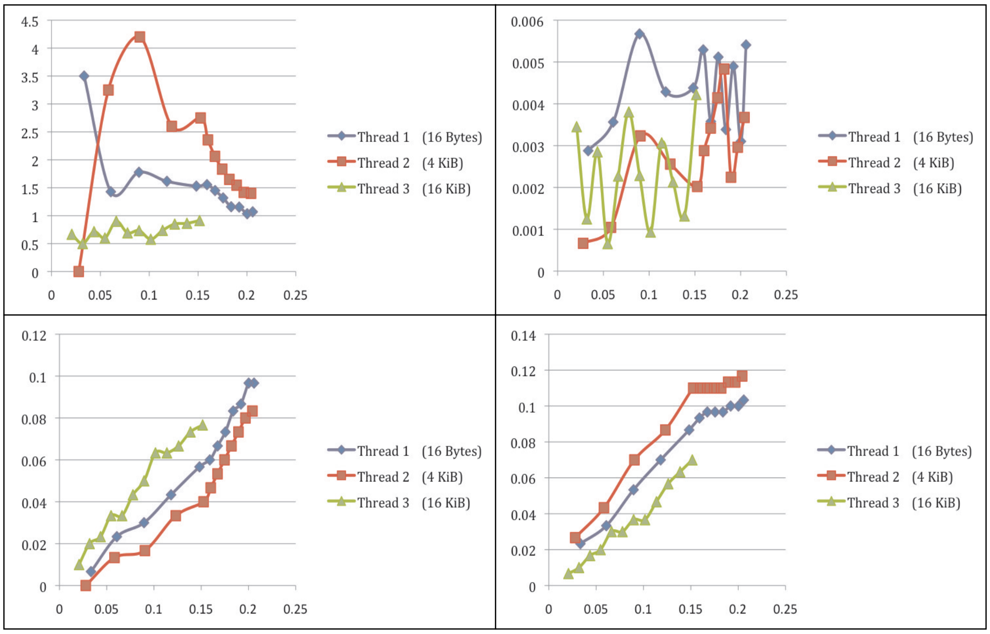

Fig. 2. Experimental results using Hoard on Linux 3.2.16

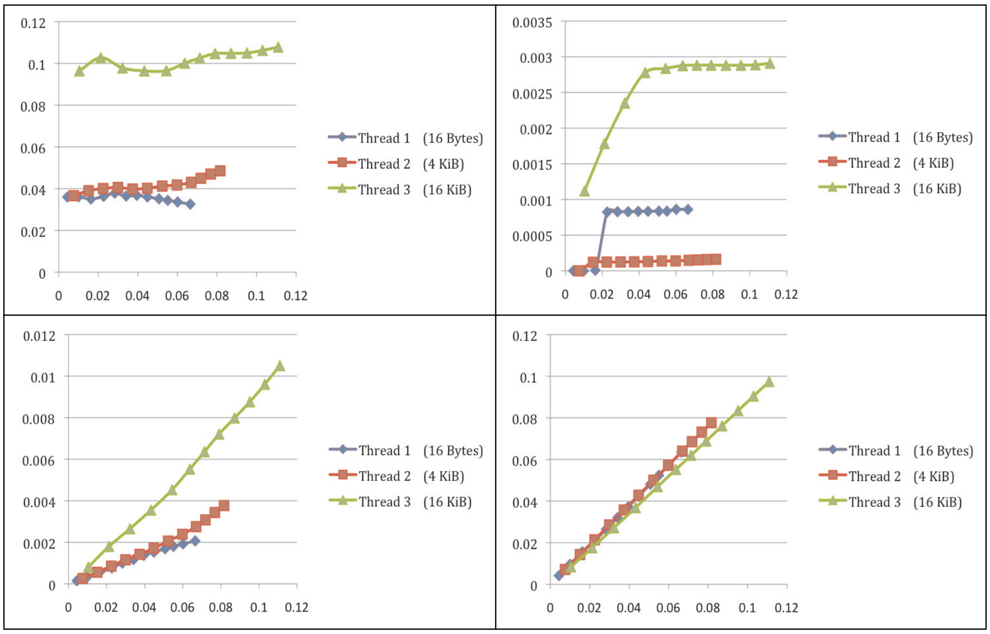

Fig. 3. Experimental results using jemalloc on Mac OS X 10.6.8

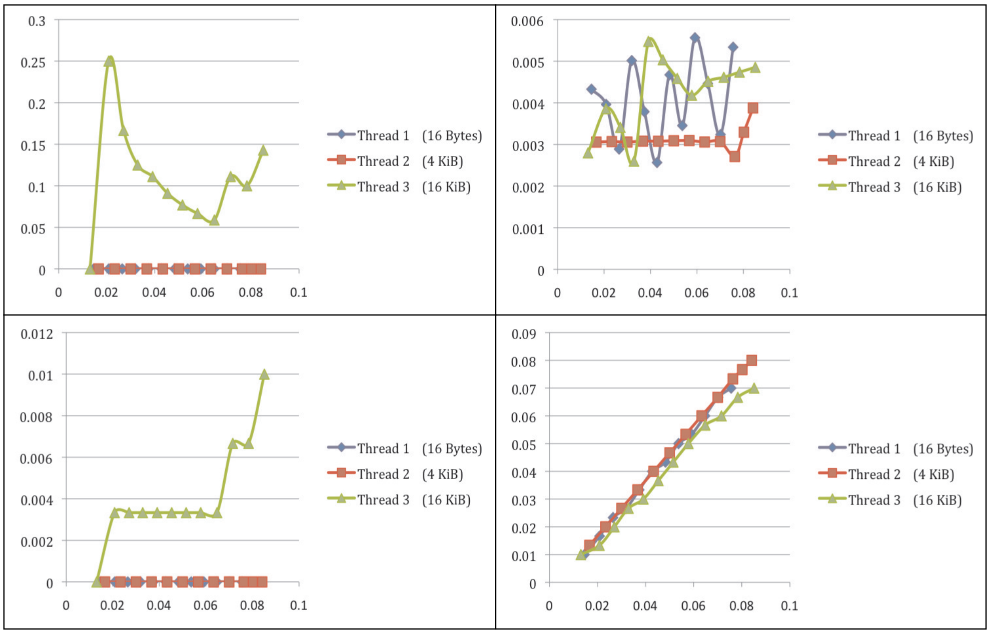

Fig. 4. Experimental results using jemalloc on Linux 3.2.16

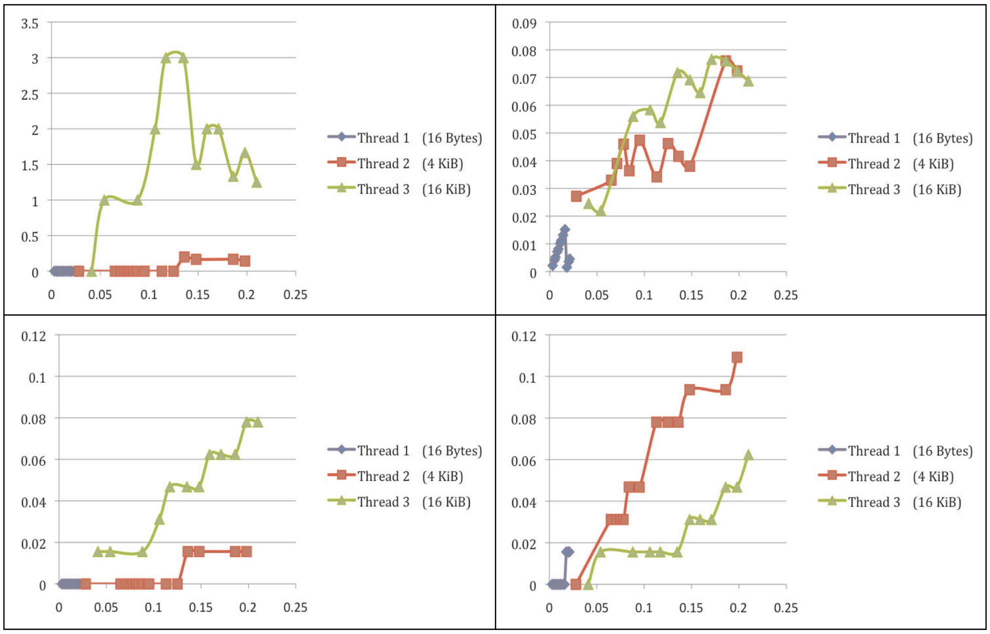

Fig. 5. Experimental results using jemalloc on Windows 6.1.7200

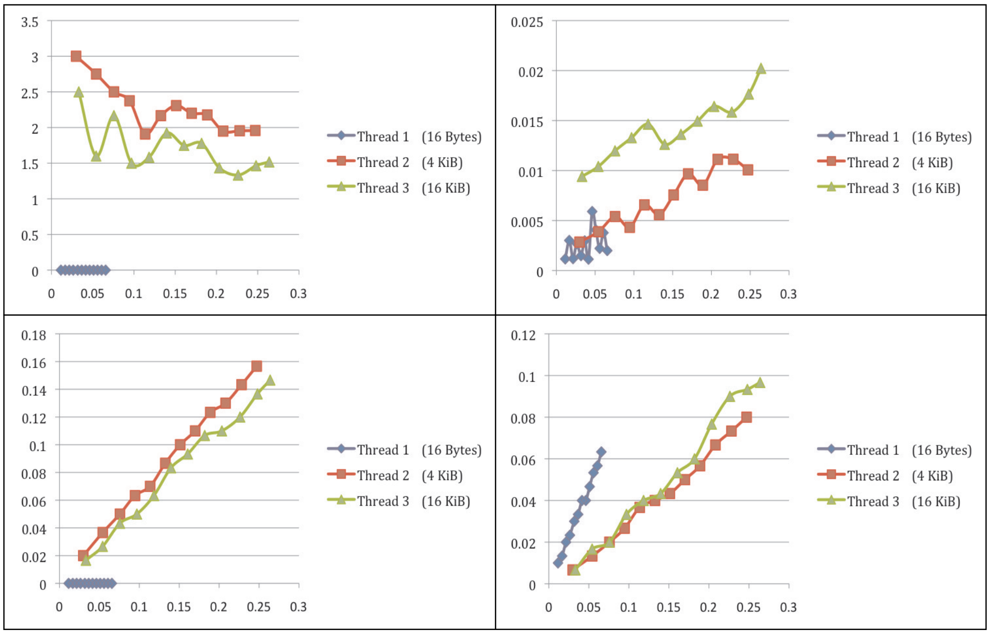

Fig. 6. Experimental results using glibc 2.5 on Linux 3.2.16

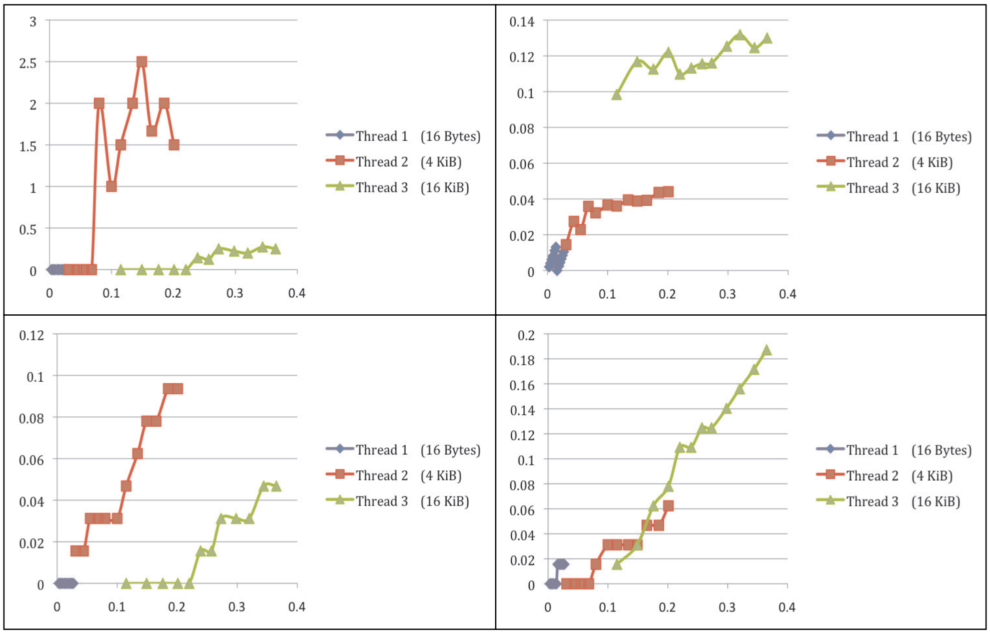

Fig. 7. Experimental results using Visual C++ 2008 on Windows 6.1.7200

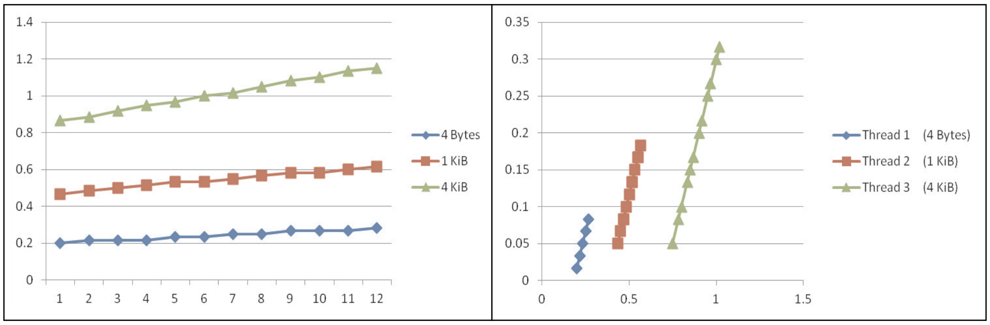

Fig. 8. Experimental results using the default allocator on VxWorks 6.7

4. Discussion of Results for All Platforms

Mac OS X is a desktop operating system built on top of Darwin, an open-source kernel derived from BSD. The Darwin kernel exhibits many characteristics desirable for real-time computing and its microkernel-like architecture sets it apart from the only true real--time operating system (VxWorks) tested. Mac OS X is also the only platform tested that includes a thread-optimized memory allocator as part of its shipped C standard library.

Tests show that the default allocator is consistently the best choice on Mac OS X, and suggest that Apple has adopted jemalloc as its default allocator, just as NetBSD and FreeBSD have. Additionally, it was the only fully 64-bit platform (kernel, C standard library, TLM binary) tested.

Just as Mac OS X is a separate entity from its kernel, the Linux kernel should not be mistaken with GNU/Linux. Many different platforms based on the Linux kernel exist, and its many variations exist to satisfy different computing requirements. TLM focused testing on the desktop/server platform known as GNU/Linux; it used an SMP variant of the Linux kernel built with pre-emptive scheduling. Results are not necessarily indicative of a Linux platform with real-time emphasis, such as Wind River Linux.

In implementing TLM on GNU/Linux, one of TLM's key components – the ability to distinguish kernel-mode time from user-mode time at the thread-level – is only possible on systems running kernel 2.6.26 or newer. For this project, we had to change test hardware late in the development cycle because the original host could only support a 2.6.25 kernel, a poor TLM test candidate. Another quirk on the GNU/Linux platform is the interface required to measure per-process memory consumption. Posix defines `getrusage()`, to measure CPU runtime and memory consumption per-process. While none of the platforms tested fully implement `getrusage()`, GNU/Linux implementation was by far the worst.

Implementing TLM on Microsoft Windows required writing separate code paths for proprietary APIs; every instance of standard Posix/BSD API calls must be replaced with their appropriate Win32 analog. Like GNU/Linux, Windows ships with a less-than-ideal memory allocator implementation. The unofficial port by Mozilla of jemalloc to Windows provides twice the runtime throughput and a fraction of the memory consumption.

Win32 has simplest interface for querying thread run-time `GetThreadTimes()`. Sadly, its returned values are very sporadic, often failing to update the reported user/kernel run-time for periods as great as 100 ms. On the other hand, Windows keeps track of thread execution by measuring the number of scheduler intervals a thread spends in user-mode, kernel-mode or asleep since its creation. Returned values are nothing more than multiplying the number of said quanta by the scheduler's time slice, which makes the granularity of thread execution timing on Win32 many times coarser than other operating systems tested.

VxWorks was the only true real-time platform tested, and the available hardware configuration was the most restrictive. Traditionally, VxWorks only schedules tasks, but offers an implementation of Posix threads given a specially configured kernel. It was later realized that VxWorks only provides Posix threads to ease porting existing software to the platform – WindRiver only exposes its most advanced scheduling features, and presumably only optimizes its memory allocation algorithms for its native task-based APIs.

The unusual results of testing on VxWorks may be attributed to a combination of unique hardware and software limitations (particularly the number of processors) and non-native implementation of concurrent job scheduling. It would appear that the default memory allocator that ships with VxWorks is kernel-bound, and locks acquired by the kernel can affect the system's ability to concurrently schedule threads created by user-mode processes. VxWorks will almost certainly benefit from a working port of a proper thread-optimized memory allocator such as jemalloc.

5. Conclusion

The goal of this project was to evaluate memory allocation performance with application to real-time systems. This was meant to be accomplished by producing and/or modifying existing memory allocation algorithms. It quickly became apparent that theory and practice were two different things. In researching existing allocation solutions, it was determined that traditional test methodology (given the vast majority of published results) was inadequate to solve many of the most important questions related to real-time viability.

Had anyone analyzed in detail the user and kernel-mode penalties for synchronizing memory allocation requests? Had the behavior of threads within concurrent systems been modeled? Had the differences between real-time kernel design and desktop kernels been considered? Overwhelmingly, the answer to each of these questions kept coming back: no.

How could anyone realistically hope to improve upon years of memory allocation wisdom if the questions asked were fundamentally wrong – thus, the focus shifted from adapting memory allocation algorithms based on common wisdom to developing new tools necessary to test the extreme limits of accepted wisdom.

The poor kernel-level implementation of thread time measurement on Windows means that the test methodology used for VxWorks, Linux and Mac OS X must be re-evaluated to produce useful results on Windows. To accommodate the limited memory space on the VxWorks test hardware, some of the tests had to be scaled back. VxWorks testing also brought up the issue of the number of processors a system has to work with.

Last, the issue of an application's compiled architecture vs. the kernel's native architecture arose in tests on Windows and Linux. The transition from ×86 to ×86-64 is an ongoing process, and consequently all major operating systems currently allow 32-bit ×86 instruction set applications to run on top of an ×86-64 kernel. Win64 requires a translation layer that converts Win32 API calls to Win64 to accomplish this, which has a measurable impact on performance.

Less is known about how Linux accomplishes this, but in both cases, using a 64-bit native word size inevitably increases memory consumption over 32-bit. To get the most accurate measure of performance in terms of runtime and memory consumption, TLM should test user-space applications that use the same architecture as the underlying kernel. With this in mind, considering the prevalence of 32-bit software running on 64-bit kernels, the results of mixed-mode execution are still very relevant.

In summary, rather than benchmarking raw throughput, or time to complete a set of tasks, the focus of this research has been on measuring allocator behavior at the absolute lowest-level possible. In effect, TLM has evolved from a simple benchmark into a full-fledged memory allocation profiler and has answered many of the questions listed above, while creating several new questions for future research. With continued work, TLM will undoubtedly lead to improved user-space memory allocation for real-time platforms.

References

[1] Plauger P.J., *The Draft C++ Standard.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[2] *Kernel Programming Guide: Kernel Architecture Overview*. Apple Inc., Mac OS X Developer Library, Cupertino, Calif., 2012.

[3] Kalev D., Understanding What Memory Alignment Means. *The Know-How Behind Application Development Website*, August 13, 1999. URL: http://www.devx.com/tips/Tip/13265

[4] Kalibera T. *et al.*, Scheduling Real-Time Garbage Collection on Uniprocessors. *ACM Trans. Computer Systems*, Vol. 29, No. 3, Article 8, August 2011.

[5] Craciunas S.S. *et al.*, Programmable Temporal Isolation in Real-Time and Embedded Execution Environments, *Proc. IIES'09, Second Workshop on Isolation and Integration in Embedded Systems*, Nürnberg, Germany, March 31, 2009.

[6] Borg A. *et al.*, Real-Time Memory Management: Life and Times, *Proc. ECRTS'06, 18th Euromicro Conference on Real-Time Systems*, Dresden, Germany, July 5–7, 2006, pp. 237–250.

[7] Bacon D., Cheng P., Rajan V.T., A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. *Proc. POPL'03, 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 15–17, 2003, pp. 285–298.

[8] *IEEE Std 1003.1b POSIX.1b, Real-time Extensions.* IEEE, Washington, DC, 1993.

[9] Berger E.D. *et al.*, Hoard: A Scalable Memory Allocator for Multithreaded Applications. *ACM SIGPLAN Notices*, Vol. 35, No. 11, pp. 117–128, 2000.

[10] Evans J., A Scalable Concurrent malloc(3) Implementation for FreeBSD. April 2006. URL: http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf.