Mitosis: A High Performance, Scalable Virtual Memory System

Juan E. Navarro* and Alan Cox
Department of Computer Science
Rice University
{jnavarro,alc}@cs.rice.edu

Abstract

Many modern applications use virtual memory APIs introduced in the 1980's in unforeseen ways, stressing the underlying data structures and exposing the old designs to a variety of performance and scalability problems. The two-decade-old data structures show their age when, for instance, a Web server maps thousands of files or a garbage collector plays memory protection tricks.

Observing how today's applications use the VM facilities, we came up with a set of requirements that any VM implementation should follow in order to efficiently support modern workloads. Current VM systems completely neglect one of these requirements, and only partially fulfill a second one. In this paper we propose a design that meet all of the requirements, and present preliminary performance results.

We also describe the future second stage of this project: the use of persistent data structures, that is, structures that are shared on a copy-on-write way. Current VM systems use copy-on-write techniques on physical memory to reduce the overhead of forking, but the semantics of fork suggest a more aggressive approach: use copy-on-write to share the data structures as well. Persistence presents a number of advantages and solves in a uniform way additional problems that current systems have solved only partially and in an ad-hoc manner. We describe how we plan to extend our implementation to include persistence.

*Also affiliated to Universidad Católica de Chile

1 Introduction

Current Unix VM systems are still based on non-scalable data structures that were designed in the 80's for the environment of that time. Today, not only do systems have more memory and run more (and larger) processes, but also processes have changed their behavior, e.g., page-protection techniques are often used to do such things as garbage collection, checkpointing, or software distributed shared memory. While the designs could have been the right ones for the time, they are no longer appropriate: we have passed the point where lack of scalability becomes a problem.

To describe a process's address space, most VM systems use an ordered list of *map entries*[1] where each map entry represents a contiguous region of memory that is mapped by the process. A simple linked list was the right structure for processes with only a handful of regions (text, data, bss and stack), but today's processes rely heavily on shared libraries, memory mapped files and anonymous memory, which add at best dozens of entries, and thousands at worst (we observed up to 7400 entries in real-world applications running in a small-sized system). In FreeBSD we found that under typical conditions, the Flash web server [4] spent more than 10% of its execution time in kernel routines that scan the map entry list. Using a balanced (binary search) tree instead of a list, as Linux does, solves the problem only partially, because the tree does not improve the time required to find unused address space.

[1] Map entries in BSD, memory segments in SVR4, memory areas in Linux.

The problem is exacerbated in systems that require regions to be uniform, that is, that all pages in the region have the same protection attributes. As a consequence, if a process changes the protection of a page in the middle of a region, the region must be split into three smaller pieces. Systems that do not have this problem represent protection attributes as a bitmap, making operations on page ranges more expensive, and paying the cost of inefficient allocation, since the bitmap has no fixed size and, hence, cannot be zone-allocated.

Another problem comes from the way copy-on-write (COW) is implemented. COW was adopted as a means to reduce fork's overhead, especially when a process is forked just to exec another program. Nevertheless, there is still considerable overhead because all VM data structures are copied from parent to child (and then discarded if the child execs). The `vfork` system call still exists in COW systems as a "dangerous" [7] alternative to avoid this overhead. In addition, current COW implementations suffer from either long object chains and swap memory leak, or inefficient allocation and additional fork overhead, as explained below.

As the previous examples show, current designs have performance and scalability problems that become evident under conditions that are not uncommon nowadays. This is a consequence of an old design that has been extended to solve particular problems in a narrowly focused manner. We propose a new design, Mitosis, that looks at all such problems from a global perspective, and solves them in a uniform way, leading to a solution that is superior to any combination of previous approaches.

Mitosis is being implemented in two stages: the first one is related to the management of map entries, and the second one deals with COW and forking. This paper shows preliminary results from the implementation of the first part of Mitosis, and motivate the need of the second part, which is in progress.

2 Requirements for performance and scalability

The management of the ordered list of map entries that describe a process's address space is a key issue in a VM system. While every-day use programs such as Netscape Navigator and Emacs may not require more than 100 entries, it is not uncommon to find programs that generate several thousands.

We studied several such cases, including a Web server, a DSM library, a Scheme implementation with a VM-based garbage collector, and a VM-based `malloc` debugger.

We have found that the data structure used for map entry management must support efficient and scalable: (1) ordered set operations such as lookup, insert, delete, predecessor, and successor; (2) operations for finding available address space of a given length; and (3) per-page attributes (such as inheritance and current and maximum protection) within each map entry, as a *separate* concept. Efficient operations should be provided to change attributes of either one page or a range of pages without affecting the map entry itself.

2.1 Current systems

Current VM systems fall roughly into two categories: Mach-like [3] and SVR4-like [2]. Both kind of systems use a simple linked lists to manage map entries; as a consequence, none of the above requirements are met. Linux superimposes an AVL tree when the list grows too large, but such approach only satisfies the first requirement above.

In Mach-like systems, map entries have fields to describe page attributes, which apply uniformly to all pages in the entry. As a consequence, attribute changes force entry splitting. The splitting increases the number of map entries, making the cost of subsequent operations on map entries higher; incurs overhead (for instance, references to the backing object must be added); and produces redundancy, since the two or three entries resulting from the split will contain essentially the same information, except for the changed page attribute.

SVR4-like systems, in contrast, use bitmaps associated to each map entry, thus managing page attributes *below* map entries. This approach does not present the aforementioned problems (note that this does not mean that SVR4 is free from long map entry lists: for instance, any file or Web server that uses `mmap` to cache files may end up with thousands of entries). Nevertheless, since the bitmap can have an arbitrary size, it must be allocated with a general-purpose memory allocator, which is much less efficient—both in time and space—than an alternative zone allocator. Also, changing the attributes of a range of pages may be more costly than in Mach-like systems. Finally, a bitmap is not the most efficient way of representing homogeneous or close to homogeneous attributes, and forks require allocation of new bitmaps to copy the forking process's ones.

2.2 A Christmas tree

We propose a hierarchical search structure decorated with free space information; each node in the structure keeps track of the largest chunk of space available right after any of its descendants. We call such a structure a Christmas tree or X-tree for short, although it does not need to be a tree proper (in fact our implementation is based on a skip list [5]). Insertions and deletions in an X-tree require some extra overhead to keep space information up-to-date, but their complexity remains $O(\log n)$. Searching for an unused address segment of a given size is also $O(\log n)$.

For per-page attributes, we propose a combination of the current two approaches that takes the best of each: manage attributes below map entries, but using a list of *attribute entries* that describe ranges of pages with uniform attributes. In addition, we propose the use of hierarchical search structure to manage the attribute entries. It may seem like overkill, but this ensures scalability and facilitates data structure persistence as described in section 4.

This data structure meets all of the stated requirements of efficiency and scalability, using only fixed-size structures and with no redundancies.

3 Preliminary results

We compared the performance of several alternative implementations of a VM system:

Base. Unmodified FreeBSD 4.1, i.e., a linked list for map entries, and page attributes at the map entry level. This implementation has a small optimization in the form of two hints. One points to the last entry hit and is used to improve lookups since it is frequently the one being looked up. The other points at the first entry after which there is unused space, and is supposed to improve the search for free space.

Bare. Same as Base, but without the hints. The intention was to assess how much the hints are actually helping.

Tree. Linux-like: a non-augmented skip list for map entries, page attributes at the map entry level.

Tree+bitmap. Same as Tree, but with a bitmap for page attributes.

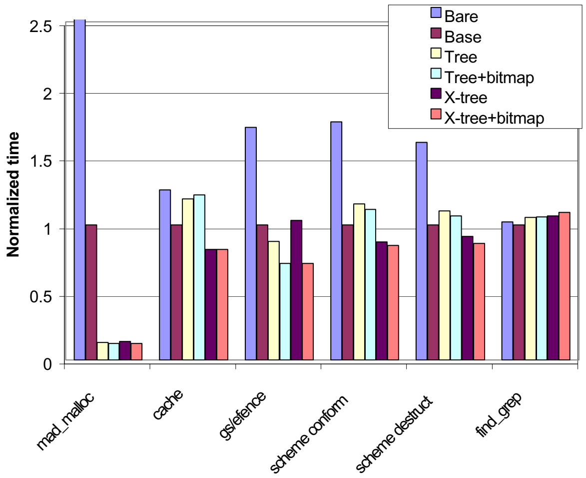
X-Tree. An X-Tree for map entries, page attributes at the map entry level.

X-Tree+bitmap. Same as X-Tree, but with a bitmap for page attributes.

Our test bed was a Pentium III machine at 550MHz, with 128 MB of memory. The graph shows the running times of 8 benchmark applications normalized to the Base implementation. Each bar is the average of 4 runs after a warm-up run. Each benchmark stresses at least one of the requirements of section 2.

The hit rate for the lookup hint in the Base implementation varies from 25% to 94%, while the free space hint ended up being completely useless because of a broken implementation (we believe that even a correct implementation would not help much, though). Therefore, the difference between Bare and Base is due only to the lookup hint. The tree and X-tree implementations do not take advantage of hints.

Mad_malloc is a benchmark that stresses the VM system by allocation and deallocation of memory under *Electric Fence*, a VM-based `malloc` debugger.



Because of its intensive use of `mprotect`, it generates over 40000 map entries in the unmodified OS. The use of a tree reduces execution time 7.5 times, while the use of bitmaps improves performance about 8% further, although the tree does not have much incidence when bitmaps are used, since only 167 entries are generated in that case. This test had the highest hint hit rate (94%), which explains why the Bare implementation did not finish in a reasonable time.

Cache simulates the behavior of the Flash Web server. Flash caches files by keeping them `mmap`ped. When a request for a new file arrives and there are too many files already mapped, some are unmapped. Therefore, insertion, deletion and finding free space are frequent operations. Interestingly, the simple tree does not help; on the contrary, it makes performance about 20% worse than the original system, because the overhead of insertions and deletions is paid with no benefits for finding space. In contrast, the X-tree improves performance by 18% with respect to the base case.

Gs/efence is GhostScript compiled with Electric Fence and used to interpret a 12-page document. The simple tree improves performance by 12.5%, and the use of bitmaps by another 16%, but the X-tree without the bitmaps degrades performance in 3%, because each allocation creates two new map entries, whose insertion forces to update the space information, which is rarely used.

Conform and destruct are part of a battery of programs being used at Rice University to benchmark Scheme implementations. We run them under one with a VM-based garbage collector. These cases, as the cache one, show the advantage of an augmented tree over a simple one.

Find_grep is just `find . -exec grep xxx` over a directory with 2000 files. The intention of this test is to show that, while more sophisticated structures to manage VM do improve scalability, they are indeed heavier weight, and the cost of copying and destroying them is higher. Therefore, `fork` and `exec` pay a penalty shown in this case: from 5.5% for the simplest implementation to 9% for the most complex. The second stage of our project tackles this problem.

4 Persistent structures

Current systems use copy-on-write (COW) techniques to reduce fork overhead, but both the Mach- and the SVR4-like COW implementations present problems.

Mach [6] uses *memory object chains* to handle COW pages. As processes fork and modify COW pages, potentially long object chains can be created, which must be traversed at each page fault. Under this approach only objects—not pages—are reference-counted. As a consequence, a page can be completely *shadowed* (thus, inaccessible by any process) yet the system will not realize this and will not free the page until all processes that are referencing the object containing the page terminate. This problem is known as the *swap memory leak*, because the page will eventually be evicted from memory but will still use swap space.

In contrast, SVR4-like systems are based on *anons*. A map entry points to an *anon reference array* which contains one slot per page in the region. Each slot contains a pointer to an anon, which is reference counted and is used to locate the page. While this approach does not have the problems of object chains, it has two drawbacks: an efficient zone allocator cannot be used for anon reference arrays since they are arbitrarily sized, and the array must be traversed both at fork and exit to update the anons's reference counts.

Our solution to these problems is inspired by the same fork semantics that led to COW of memory pages: apply the same technique not just to pages, but to the VM data structures as well. Right after forking, both parent and child will point to the same structure; as the processes change the address space in any way, only those parts of the structure that need to be different for both processes are copied before being modified. The graphical representation of the process that makes the shared structure progressively disjoined is remarkably similar to cell division by mitosis; hence the name of the project.

We call a structure shared in a COW way a persistent structure [1], because several versions (one for each process) of the same structure are kept at a time. There must be only one access point to each version of the structure and items within the structure must be reference counted. Each time an item is modified, the path that leads from the entry point to the item must be copied, starting from the first item whose reference count is greater than one. If the path from the entry point to any item is short, then copying is reduced and sharing increases; this suggests a tree-like structure.

Our design decisions for section 2 were guided by the requirements of persistent structures. In particular we chose a (slightly modified) skip list because rotations required by most search trees kill sharing of nodes.

Fork overhead is substantially reduced, because the VM data structures will be copied only as necessary, i.e., never if the child execs immediately. Even if the child does not exec, it is likely that nodes corresponding to read-only portions of the address space remain shared, thus saving the time and memory overhead of copying.

5 Conclusions and future work

Observing the behavior of applications we have defined a set of requirements that a VM implementation should follow to efficiently support modern workloads. We proposed and evaluated a structure that meets all of the requirements. In particular, we showed that a hierarchical search structure to manage map entries is not sufficient for scalability; the structure must be augmented with information to locate unused space logarithmically rather than linearly. In addition, per-page attributes should be managed *below* (as opposed to *at the level off*) map entries.

The use of persistent data structures solves the remaining problems of current designs, while eliminating the potential disadvantages of having heavier, more sophisticated structures.

In short, Mitosis is superior to any combination of previous approaches, since it puts together all their advantages (and even more) without any of the drawbacks. Mitosis's advantages include a simple, unified design based on efficient, sophisticated data structures with no pathological cases possible; scalability to large, complex address spaces; low cost of forking; absence of object chains and memory leaks; efficient zone allocation due to the use of fixed-size structures only; and fewer memory requirements.

Besides finishing Mitosis's implementation, future work includes studying mechanisms to share in a COW fashion portions of the physical page table, and investigating ways of sharing map entries between unrelated processes in order to reduce start-up time of processes (currently dominated by the loading of shared libraries).

6 Acknowledgments

This work is being supported by a USENIX Student Research Grant. We would also like to thank Sitaram Iyer for his contribution with countless (good, bad and ugly) hacking tricks, and John Clements for his help with the Scheme benchmarks.

References

[1] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, Berkeley, California, 28–30 May 1986.

[2] J. Mauro and R. McDougall. *Solaris Internals.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 2001.

[3] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, Reading, Massachusetts, 1996.

[4] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, June 1999.

[5] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[6] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Bar, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor operating system. *IEEE Transactions on Computers*, C-37:896–908, 1988.

[7] U. Vahalia. *UNIX Internals.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1996.