Measuring the Impact of the Linux Memory Manager

Mel Gorman
Uni. of Limerick, Castletroy, Limerick, Ireland
IBM DSL, Damastown, Dublin, Ireland
melgor@ie.ibm.com

Patrick Healy
CSIS Department
University of Limerick
Limerick
Ireland
patrick.healy@ul.ie

ABSTRACT

A major concern of almost any operating system is the methods and mechanisms it uses to manage its resources. Physical memory and, in many cases, how it is mapped to virtual memory is one resource that almost all aspects of an operating system and its applications require. Hence, the Memory Manager (MM) and Virtual Memory Managers (VMM) play a critical role in the systems performance.

Due to its critical nature, it is important to understand all aspects of how the MM interacts with the operating system and the different types of processes that run on the machine. Our research strives to understand and define metrics measuring the performance of a MM and implement meaningful tools specifically for Linux.

This paper will start by introducing some of the performance concerns of the MM and how they might be measured. At each stage, we will introduce the research we are performing to address the problems.

General Terms

Linux, Virtual Memory Management, Metrics

1. INTRODUCTION

Historically, the memory manager was a core subsystem but not tightly integrated with other portions of the operating system. In recent times, particularly in monolithic kernels, the trend has been to tightly integrate the resource management of IO and filesystems with the memory manager. The integration is especially noticeable in Linux as portions of the IO manager are almost indistinguishable from the memory manager.

This tighter integration implies that the overall performance of the operating system is increasingly dependant on the performance of the memory manager. Despite this, it is difficult to measure performance of just the memory manager. Benchmarks tend to measure a secondary effect and extrapolate the performance of the memory manager based on the results. This makes it very difficult to determine what aspect of a memory manager may be resulting in poor performance. This paper will cover some aspects of the memory manager that impact overall performance of the system and some of the research we are conducting in the area.

The first subsystem examined in the memory manager is the page allocator, discussed in Section 2. Page allocation is a *critical code path* as almost all of an operating system require physical memory. The most important criteria for a page allocator is that it is fast[7] to avoid allocators being built upon the system allocator. The second important criteria is its internal and external fragmentation is kept low. Internal fragmentation is where more memory than necessary is allocated to satisfy a request wasting memory. External fragmentation is where enough memory is free to satisfy a request, but it is split into two or more chunks. Linux uses two standard allocators. The first is a binary buddy allocator for the allocation of pages and the second is a slab allocator for the allocation of small objects.

Section 3 examines how information is copied between userspace and kernel-space. All interaction with the kernel will involve copying data to and from userspace so how the OS handles this is important. On the x86, Linux uses the upper 1GiB of address space for every process table so that the kernel tables are effectively global in nature. This allows the kernel to easily copy data to and from the current process. However, there are other problems with copying to/from other processes or interacting with physical memory over the 1GiB mark.

The process address space and how it is managed will be discussed in Section 4. Modern Linux applications, particularly GUI applications depend on large number of mappings and tracking these effectively is important. For example, **gnome-panel 2.6.2** requires 175 mappings involving 74 separate libraries. Effectively managing this large number of mappings has important performance considerations.

A popular consideration, although sometimes over-emphasised, is the page replacement policy discussed in Section 5. An OS that frequently replaces the wrong page will suffer serious slowdowns so it is important that the correct decisions be made. A related topic is how effectively the OS manages its backing storage.

LIBRE05 '05 Libre Software Meeting, Dijon France

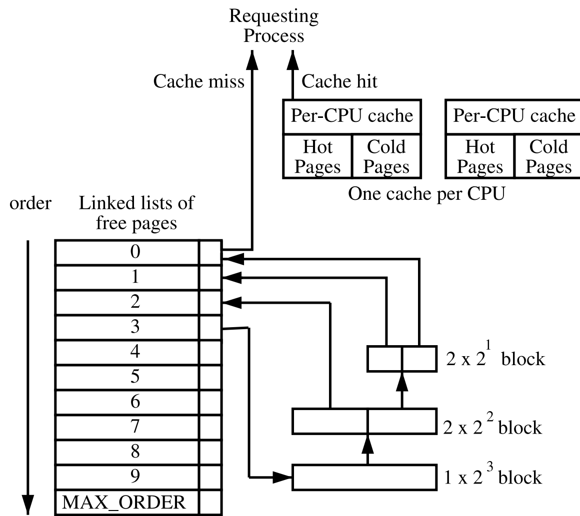

Figure 1: Binary Buddy Allocator in Linux

2. ALLOCATORS

Linux uses a binary buddy allocator with some minor alterations. The most important alternation is the use of a per-cpu cache for order-0 (i.e. 2^0 or 1 page) allocations to avoid locking on multi-processor machines. This arrangement is shown in Figure 1. The aspects of the page allocator that concern performance are as follows;

Raw performance Linux chooses the binary buddy allocator because it has superior time performance[6]. As a result, Linux has very few specialised allocators and they usually exist for emergency pools rather than for avoiding the time overhead of the allocator. Measuring the raw performance of the allocator is achieved by using the Aim9 Benchmark, paying particular attention to page_test and brk_test.

Page coloring Page coloring is a technique that makes sure that physical pages that shared cache lines do not use adjacent areas in virtual memory. As program usually reference memory in localised patterns, it is desirable they do not have poor cache performance between adjacent virtual addresses. This intuitively makes sense, but it is very difficult to measure the effect of adjacent virtual pages having the same "colour". Hence, does not use page coloring in the binary buddy allocator because no benchmark has been devised that show a performance gain for a given workload but the additional complexity to the allocator is noticeable. Coloring is provided within the slab allocator where it can be trivially provided, even though the performance gain is very difficult to measure.

Fragmentation The third metric to use for allocators is its fragmentation properties. The standard binary buddy allocator suffers badly from external fragmentation[9] meaning that subsystems in Linux are carefully written to not require large blocks of physically contiguous memory. Ordinarily, the binary buddy allocator also suffers badly from internal fragmentation but this is heavily mitigated in Linux using the Slab Allocator[2][3].

2.1 Allocators and Metrics

Raw Performance It is relatively simple to measure the raw performance of the allocator using the Aim9 benchmark (*http://sourceforge.net/projects/aimbench*). The Aim9 benchmark runs a number of tests on a wide variety of operating system functions but the ones most important for the performance of the page allocator are listed in Table 2.

Test	Description
page_test	System Allocations AND Pages/second
brk_test	System Memory Allocations/second
exec_test	Program Loads/second
fork_test	Task Creations/second

Figure 2: Aim9 tests reflecting the page allocator

page_test is the best indicator of raw performance. To help run the test and compare results, two utilities come with VMRegress (*http://www.csn.ul.ie/∼mel/projects/vmregress*) called **bench-aim9.sh** and **diff-aim9.sh**. **bench-aim9.sh** runs the AIM9 benchmark non-interactively, stops it after a test specified on the command line and stores the result in a directory based on the kernel version and a user-specified string. A sample report is shown in Figure 3. **diff-aim9.sh** will take two sets of Aim9 results, print the results from each, the difference between them and the percentage differences. A simple report is shown in Figure 4.

Fragmentation There is no single metric for measuring fragmentation all involve some function of free space, the size of free blocks and in some cases, what has already been allocated[5]. The equation we use to measure external fragmentation in Linux is;

$$F_{extfrag} = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}$$

Where 2^n is the largest allocation that can be satisfied, j is the order of the desired allocation and k_i is the number of free page blocks of size 2^i. This yields a number between 0 and 1 where 0 indicates there is no fragmentation for an allocation of 2^j and 1 means the allocation cannot be satisfied. Expressing this as a percentage of fragmentation is a case of multiplying the result by 100.

This necessary information is collected from `/proc/buddyinfo` by the utility **extfrag_stat.pl** provided by VMRegress. A sample report is shown in Figure 6 that expressed fragmentation in terms of percentage.

Linux has no mechanism for showing what the distribution of pages throughout the address space based on either the age of the allocation or its type. The ideal would be that all allocations are grouped together by age but the required accounting makes it impractical. The closets alternative is to group allocations together by type. To measure the distribution of pages by type, patches and tools are provided by VMRegress. Once the patches are applied, the tool **mapfrag_stat.pl** is able to generate a web page showing the distribution of page types in each zone.

We implemented a patch that arranged pages into the types Kernel Unreclaimable, Kernel Reclaimable and User Re-

Test Number	Test Name	Elapsed Time (sec)	Iteration Count	Iteration Rate (loops/sec)	Operation Rate (ops/sec)	
1	creat-clo	60.05	1116	18.58451	18584.51	File Creations and Closes/second
2	page_test	60.01	4414	73.55441	125042.49	System Allocations & Pages/second
3	brk_test	60.04	1608	26.78215	455296.47	System Memory Allocations/second
4	jmp_test	60.00	250917	4181.95000	4181950.00	Non-local gotos/second
5	signal_test	60.01	5448	90.78487	90784.87	Signal Traps/second
6	exec_test	60.03	781	13.01016	65.05	Program Loads/second
7	fork_test	60.05	928	15.45379	1545.38	Task Creations/second
8	link_test	60.01	6102	101.68305	6406.03	Link/Unlink Pairs/second

Figure 3: AIM9 Report

1	creat-clo	18483.33	18584.51	101.18	0.55%	File Creations and Closes/second
2	page_test	121926.35	125042.49	3116.14	2.56%	System Allocations & Pages/second
3	brk_test	441779.11	455296.47	13517.36	3.06%	System Memory Allocations/second
4	jmp_test	4184216.67	4181950.00	-2266.67	-0.05%	Non-local gotos/second
5	signal_test	92051.32	90784.87	-1266.45	-1.38%	Signal Traps/second
6	exec_test	64.87	65.05	0.18	0.28%	Program Loads/second
7	fork_test	1544.49	1545.38	0.89	0.06%	Task Creations/second
8	link_test	6345.14	6406.03	60.89	0.96%	Link/Unlink Pairs/second

Figure 4: AIM9 Difference Report

claimable so that pages that were easily discarded could be grouped together. With this mechanism, it is possible to allocate large blocks of physically contiguous memory by just reclaiming the User-Reclaimable pages.

Figure 5 shows the difference in distribution of page types in the Normal Zone with the standard allocator and an allocator designed to reduce fragmentation. The map was generated after a light run of a kernel-compile-based benchmark called **bench-stresshighalloc.sh**. It is interesting to note that KernelReclaimable pages (value of 2 on the y axis) is barely noticeable in the modified allocator as they are confined to a small portion of the map but scattered throughout the address space in the standard allocator. Under heavy pressure, the modified allocator was able to allocate 63 4MiB pages in comparison to 3 4MiB pages returned by the standard allocator.

3. USER-KERNEL COPYING

Up to one third of kernel time is spent copying between user and kernel space[7] making it a critical operation for the kernel and it is the reason why optimizations like zero-copy exist. There are two traditional ways of copying the data. The first is to use a portion of the each processes address space to map a single view of the kernel address space. This gives the illusion that the kernel space is global in nature and always available. In this configuration, copying is simply a case of reading the userspace portions of the address space either in single bytes or using block-copy operations. The second option is to map the kernel is its own private address space. Information is then copied between the user and kernel spaces using special instructions.

The advantage with sharing the address space is that it is fast and relatively simple to implement. The operations required to copy between address spaces are only half as fast as copying within the same address space on some architectures. Additionally, many architectures will flush the Translation Lookaside Buffer (TLB) when changing address spaces which is a significant penalty for processes that interact heavily with the kernel.

The major disadvantage of sharing the address space is that the entire process address space is not available for use. This means that there is a limit to how much memory may be mapped by a process at a given time. This limit is significantly lower than the addressing capabilities of the hardware. To compound the problem, kernel operations are not able to use all the addressable physical memory and must temporarily map the higher physical addresses into the virtual address space.

Linux provides both usage models for address-space copying. Most architectures share the address space with the split location being architecture dependant. For example, by default the x86 uses the lower 3GiB for the userspace portion and the upper 1GiB for the kernel. The option is provided on the x86 to use a private address space for the kernel with the resulting performance penalties for users that are heavily CPU bound and are required to map the full address space.

It is interesting to note that some architectures like the Sparc64 have hardware support for keeping the kernel in a separate address space without significant performance penalties. Linux does not currently take advantage of this.

3.1 Metrics measuring User-Kernel Copying

Currently, there are no metrics that specifically measure the cost of user-kernel copying. They are indirectly measured by counting how many system calls be be made in a second or how many network operations may be performed. Specifically, web-benchmarks have a heavy dependency on the kernels ability to quickly copy data between different types of

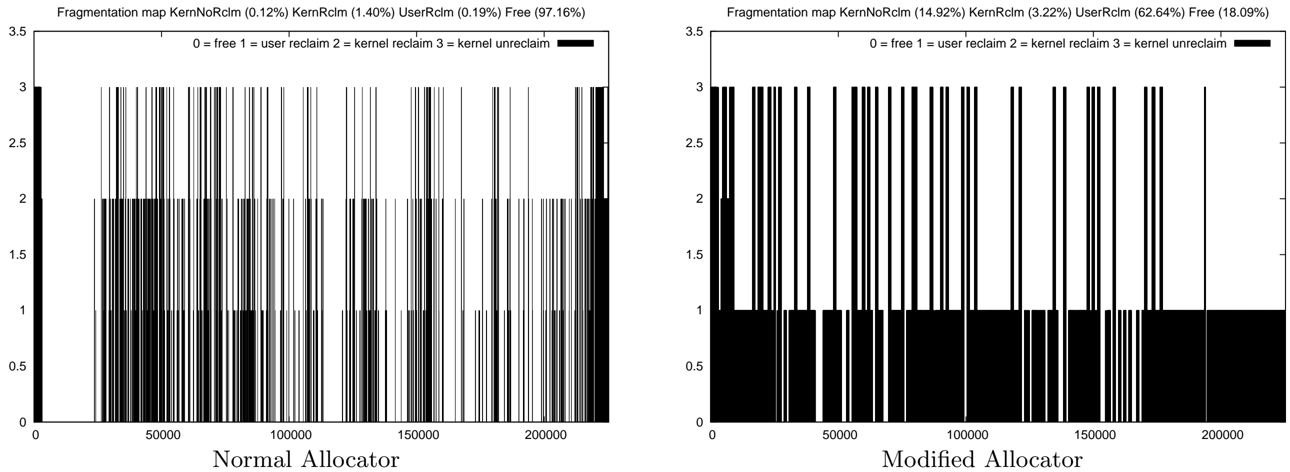

Figure 5: Distribution Map Differences

```
Zone   (% Fragmentation Orders 2^0 -> 2^MAX_ORDER)
DMA    0.000  4.331  6.299  7.087 14.961 18.110 24.409 24.409 49.606 100.000 100.000
Normal 0.000 37.260 37.534 37.534 38.630 38.630 38.630 47.397 64.932 100.000 100.000
```

Figure 6: External fragmentation report

buffers. A targeted metric that did measure the speed of copying would need to measure the following;

- Number of single bytes copied per second (integer parameters)
- Number of pages copied per second (buffers)
- Time taken to fault a user page (read/write on a paged-out page)
- Time taken to setup a zero-copy buffer (networking)

4. PROCESS ADDRESS SPACE

On the x86, 3GiB is available for a process to address by default. As the majority of applications do not use the whole address space, the OS is responsible for tracking what regions are in use and for what purpose. Linux uses a number of structures to track what the portions are being used for.

The first is the `task_struct` of which there are one for every process, thread and kernel thread running in the system. All the tasks in a system are linked together on a doubly linked list that is rarely searched. All tasks except kernel threads will have an associated `mm_struct` where threads within a process share the same `mm_struct`.

Each region in the address space is tracked by a `vm_area_struct` (commonly abbreviated to VMA). The search complexity of the data structure tracking this is performance critical as a wide variety of address space related operations depend on finding the correct `vm_area_struct`, or on some cases, the correct sized hole quickly. To address the requirement to find both VMAs and holes quickly, the VMAs are arranged on both a doubly linked list ($O(n)$ search complexity) and a red-black tree($O(log(n))$ search complexity, both sorted by address.

To further speed the search times of VMAs, a pointer to the last found VMA is stored in `mm_struct`→`mmap_cache`. To help search for holes, the address of a known large hole is stored in `mm_struct`→`free_area_cache`. An interesting property of `free_area_cache` is that it fragments the address space badly although it is not an issue that affects many types of process.

A more interesting problem in the area of process address space management is the use of variable sized pages. Many architectures support more than on page size, including the x86. By default, Linux uses the largest possible pages for the kernel image and a small fixed size page for almost everything else, the exception being files backed by the HugeTLB Filesystem. This can punish applications that use a large portion of their address space such as Java Virtual Machines and Database servers.

The punishment is because of inefficient TLB usage. When a memory reference is made, page tables are searched for the physical page backing the virtual address. The result of this search is cached in a TLB but it is of limited size. A sparse user of the address space will flush the TLB frequently which is a surprisingly penalty, about 900 clock cycles on an Pentium III Xeon. According to the specifications[4], this machine has 128 entries and 64 entries for 4KiB pages in the ICache and DCache respectively. For large pages, it shares the entries in the ICache but each of the 64 entries in the DCache may be used fro either 4KiB pages or 4MiB pages. This means that the 64 entries may be used to store the physical addresses of either 256KiB of data or 256MiB of data in the TLB. For applications using large amounts of

data, this could be a significant saving.

Given such significant differences in how much memory may be addressed by the TLB, Linux has surprisingly poor support for variable sized pages due to a combination of historical reasons. One, many parts of the kernel make assumptions on the size of the page and introducing variable sized pages would break these assumptions. Second, at run-time, large pages are rarely available due to external fragmentation in the page allocator. Despite this problems, an important aim of our research is to measure the performance penalty from inefficient TLB usage and see what may be done to address it.

4.1 Metrics for the Process Address Space

Again, a specific metric does not exist. In the past, small benchmarks that called `mmap()` and `munmap()` were used for different sized regions to determine how fast the appropriate areas could be found. Indirectly, any benchmark that measured the time taken to page fault on a process with many memory-mapped regions would indirectly measure the ability of the operating system to find regions.

5. PAGE REPLACEMENT

Page replacement is one area that has received a lot of attention, both in research academically and within the Linux community. In any demand-paged system, the requirement will exist to select pages for removal from the system or saving to backing storage. Due to the very slow relative speeds of disk accesses, removal is an expensive operation, only matched by the expense of reading the data back in. Hence, the performance of a system that needs to page frequently is heavily dependant on the correct selections from the page replacement policy.

Linux uses a page-replacement policy that defies a specific categorisation. It is called an LRU policy but it is really an LRU-approximation using a combination of WSClock and LRU-2Q. Some heuristics are in place to mitigate pollution of the lists due to processes that scan their address space such as media streaming applications. To avoid some of the traditional locking problems associated with LRU, Linux has a set of lists for each zone in the system, each set protected by its own lock.

The page replacement policy is heavily tuned in Linux but there are no tools for measuring the effectiveness of the replacement policy, partially because of its complexity and partially due to the lack of a suitable benchmark. Hence, reports tend to be the subjective experience of the end-user, not a reliable metric.

For decades, LRU and LRU approximations were the best replacement policy to use as, in some circumstances, they are the optimal replacement policy for workloads that exhibit high locality of reference. Modern applications are showing a much weaker locality of reference, especially with increase in applications based on Java Virtual Machines and a number of media applications. Alternative replacement policies such as ARC[8] and CAR[1] but there is no solid plan in place for the implementation of such policies.

5.1 Metrics and Page Replacement

This area is very difficult to measure the effectiveness for. Traditionally, analysis of page replacement algorithms have been through competitive analysis on the mathematical behavior of the algorithm and through simulations. Collection the necessary data on a live system is very difficult and reproducing the test scenario each time is even harder. Frequently on Linux, benchmarks, such as the Andrew Morton Wiggle Test, have used the subjective experience of the user to measure how much the system appears to "lock up" under memory pressure. Indirectly, other benchmarks such as ConTEST (*http://members.optusnet.com.au/ckolivas/contest/*) measure the page replacement policy by timing how long it takes to finish each part of the test. However, in that case, the number of variables affecting the final result are considerable.

Our research goal will be to develop a tool that measures the replacement policy used by Linux in comparison with Belady's MIN optimal offline algorithm. We intend to measure statistics such as cache misses and page fault rates for arbitrary workloads. To measure the effectiveness of the replacement decisions, we intend to build a number of specific workload types, run them standalone on the system and then force the replacement policy to remove the pages. While the pages are being removed, for known workloads, we will be able to determine if the correct pages were chosen in the correct order within a reasonable confidence.

6. CONCLUSIONS

In this paper, we have listed some of the aspects of the VM that have important performance considerations for the whole system. We discussed how they are implemented in Linux and what our research goals in these areas are.

The intention over the next three years is to develop specific metrics that may be used on each part of the VM and apply those to Linux. We hope to introduce some support for variable sized pages within the operating system and use our metrics to show the performance gain or loss due to such a change.

7. REFERENCES

[1] S. Bansal and D. Modha. Car: Clock with adaptive replacement. In *USENIX Conference on Fileand Storage Technologies (FAST)*, 2004. libre2005: Brief mention.

[2] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[3] J. Bonwick and J. Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX-01)*, pages 15–34, Berkeley, CA, June 25–30 2001. The USENIX Association.

[4] Intel. *IA-32 Architecture Software Developers Manual Volume 3*. Intel, 2002.

[5] K. Kaka and J. Meadows. A technique to evaluate dynamic storage management. In *ANSS '79: Proceedings of the 12th annual symposium on*

Simulation, pages 201–214, Piscataway, NJ, USA, 1979. IEEE Press.

[6] D. G. Korn and K.-P. Bo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

[7] M. K. McKusick. *The design and implementation of the BSD operating system.* Addison-Wesley, 2005. Libre2005: Same as mckusick96 except it's on page 148.

[8] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 03), San Franciso, CA*, 2003. libre2005: Brief mention.

[9] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.