Reprinted from the

Proceedings of the Linux Symposium

Volume One

July 21th–24th, 2004
Ottawa, Ontario
Canada

Hotplug Memory and the Linux VM

Dave Hansen, Mike Kravetz, with Brad Christiansen
IBM Linux Technology Center
haveblue@us.ibm.com, kravetz@us.ibm.com, bradc1@us.ibm.com

Matt Tolentino
Intel
matthew.e.tolentino@intel.com

Abstract

This paper will describe the changes needed to the Linux memory management system to cope with adding or removing RAM from a running system. In addition to support for physically adding or removing DIMMs, there is an ever-increasing number of virtualized environments such as UML or the IBM pSeries™ Hypervisor which can transition RAM between virtual system images, based on need. This paper will describe techniques common to all supported platforms, as well as challenges for specific architectures.

1 Introduction

As Free Software Operating Systems continue to expand their scope of use, so do the demands placed upon them. One area of continuing growth for Linux is the adaptation to incessantly changing hardware configurations at runtime. While initially confined to commonly removed devices such as keyboards, digital cameras or hard disks, Linux has recently begun to grow to include the capability to hot-plug integral system components. This paper describes the changes necessary to enable Linux to adapt to dynamic changes in one of the most critical system resource—system RAM.

2 Motivation

The underlying reason for wanting to change the amount of RAM is very simple: availability. The systems that support memory hot-plug operations are designed to fulfill mission critical roles; significant enough that the cost of a reboot cycle for the sole purpose of adding or replacing system RAM is simply too expensive. For example, some large ppc64 machines have been reported to take well over thirty minutes for a simple reboot. Therefore, the downtime necessary for an upgrade may compromise the five nine uptime requirement critical to high-end system customers [1].

However, memory hotplug is not just important for big-iron. The availability of high speed, commodity hardware has prompted a resurgence of research into virtual machine monitors—layers of software such as Xen [2], VMWare [3], and conceptually even User Mode Linux that allow for multiple operating system instances to be run in isolated, virtual domains. As computing hardware density has increased, so has the possibility of splitting up that computing power into more manageable pieces. The capability for an operating system to expand or contract the range of physical memory resources available presents the possibility for virtual machine implementations to balance memory requirements and improve the management of memory availability between domains[1]. This author currently leases a small User Mode Linux partition for small Internet tasks such as DNS and low-traffic web serving. Similar configurations with an approximately 100 MHz processor and 64 MB of RAM are not uncommon. Imagine, in the case of an accidental Slashdotting, how useful radically growing such a machine could be.

3 Linux's Hotplug Shortcomings

Before being able to handle the full wrath of Slashdot. we have to consider Linux's current design. Linux only has two data structures that absolutely limit the amount of RAM that Linux can handle: the page allocator bitmaps, and `mem_map[]` (on contiguous memory systems). The page allocator bitmaps are very simple in concept, have a bit set one way when a page is available, and the opposite when it has been allocated. Since there needs to be one bit available for each page, it obviously has to scale with the size of the system's total RAM. The bitmap memory consumption is approximately 1 bit of memory for each page of system RAM.

4 Resizing `mem_map[]`

The `mem_map[]` structure is a bit more complicated. Conceptually, it is an array, with one `struct page` for each physical page which the system contains. These structures contain bookkeeping information such as flags indicating page usage and locking structures. The complexity with the `struct page`s is associated when their size. They have a size of 40 bytes each on i386 (in the 2.6.5 kernel). On a system with 4096 byte hardware pages, this implies that about 1% of the total system memory will be consumed by `struct page`s alone. This use of 1% of the system memory is not a problem in and of itself. But, it does other problems.

[1]err, I could write a lot about this, so I won't go any further

The Linux page allocator has a limitation on the maximum amounts of memory that it can allocate to a single request. On i386, this is 4MB, while on ppc64, it is 16MB. It is easy to calculate that anything larger than a 4GB i386 system will be unable to allocate its `mem_map[]` with the normal page allocator. Normally, this problem with `mem_map` is avoided by using a boot-time allocator which does not have the same restrictions as the allocator used at runtime. However, memory hotplug requires the ability to grow the amount of `mem_map[]` used at runtime. It is not feasible to use the same approach as the page allocator bitmaps because, in contrast, they are kept to small-enough sizes to not impinge on the maximum size allocation limits.

4.1 `mem_map[]` preallocation

A very simple way around the runtime allocator limitations might be to allocate sufficient memory form `mem_map[]` at boot-time to account for any amount of RAM that could possibly be added to the system. But, this approach quickly breaks down in at least one important case. The `mem_map[]` must be allocated in low memory, an area on i386 which is approximately 896MB in total size. This is very important memory which is commonly exhausted [4],[5],[6]. Consider an 8GB system which could be expanded to 64GB in the future. Its normal `mem_map[]` use would be around 84MB, an acceptable 10% use of low memory. However, had `mem_map[]` been preallocated to handle a total capacity of 64GB of system memory, it would use an astounding 71% of low memory, giving any 8GB system all of the low memory problems associated with much larger systems.

Preallocation also has the disadvantage of imposing limitations possibly making the user decide how large they expect the system to be, either when the kernel is compiled, or when it is booted. Perhaps the administrator of the above 8GB machine knows that it will never get any larger than 16GB. Does that make the low memory usage more acceptable? It would likely solve the immediate problem, however, such limitations and user intervention are becoming increasingly unacceptable to Linux vendors, as they drastically increase possible user configurations, and support costs along with it.

4.2 Breaking `mem_map[]` up

Instead of preallocation, another solution is to break up `mem_map[]`. Instead of needing massive amounts of memory, smaller ones could be used to piece together `mem_map[]` from more manageable allocations Interestingly, there is already precedent in the Linux kernel for such an approach. The discontiguous memory support code tries to solve a different problem (large holes in the physical address space), but a similar solution was needed. In fact, there has been code released to use the current discontigmem support in Linux to implement memory hotplug. But, this has several disadvantages. Most importantly, it requires hijacking the NUMA code for use with memory hotplug. This would exclude the use of NUMA and memory hotplug on the same system, which is likely an unacceptable compromise due to the vast performance benefits demonstrated from using the Linux NUMA code for its intended use [6].

Using the NUMA code for memory hotplug is a very tempting proposition because in addition to splitting up `mem_map[]` the NUMA support also handles discontiguous memory. Discontiguous memory simply means that the system does not lay out all of its physical memory in a single block, rather there are holes. Handling these holes with memory hotplug is very important, otherwise the only memory that could be added or removed would be on the end.

Although an approch similar to this "node hotplug" approach will be needed when adding or removing entire NUMA nodes, using it on a regular SMP hotplug system could be disastrous. Each discontiguous area is represented by several data structures but each has at least one `structzone`. This structure is the basic unit which Linux uses to pool memory. When the amounts of memory reach certain low levels, Linux will respond by trying to free or swap memory. Artificially creating too many zones causes these events to be triggered much too early, degrading system performance and under-utilizing available RAM.

5 CONFIG_NONLINEAR

The solution to both the `mem_map[]` and discontiguous memory problems comes in a single package: nonlinear memory. First implemented by Daniel Phillips in April of 2002 as an alternative to discontiguous memory, nonlinear solves a similar set of problems.

Laying out `mem_map[]` as an array has several advantages. One of the most important is the ability to quickly determine the physical address of any arbitrary `struct page`. Since `mem_map[N]` represents the Nth page of physical memory, the physical address of the memory represented by that `struct page` can be determined by simple pointer arithmetic:

Once `mem_map[]` is broken up these simple

```
physical_address = (&mem_map[N] - &mem_map[0]) * sizeof(struct page)

struct page N = mem_map[(physical_address / sizeof(struct page)]
```

Figure 1: Physical Address Calculations

calculations are no longer possible, thus another approach is required. The nonlinear approach is to use a set of two lookup tables, each one complementing the above operations: one for converting `struct page` to physical addresses, the other for doing the opposite. While it would be possible to have a table with an entry for every single page, that approach wastes far too much memory. As a result, nonlinear handles pages in uniformly sized sections, each of which has its own `mem_map[]` and an associated physical address range. Linux has some interesting conventions about how addresses are represented, and this has serious implications for how the nonlinear code functions.

5.1 Physical Address Representations

There are, in fact, at least three different ways to represent a physical address in Linux: a physical address, a `struct page`, and a page frame number (pfn). A pfn is traditionally just the physical address divided by the size of a physical page (the *N* in the above in Figure 1). Many parts of the kernel prefer to use a pfn as opposed to a `struct page` pointer to keep track of pages because pfn's are easier to work with, being conceptually just array indexes. The page allocator bitmaps discussed above are just such a part of the kernel. To allocate or free a page, the page allocator toggles a bit at an index in one of the bitmaps. That index is based on a pfn, not a `struct page` or a physical address.

Being so easily transposed, that decision does not seem horribly important. But it does cause a serious problem for memory hotplug. Consider a system with 100 1GB DIMM slots that support hotplug. When the system is first booted, only one of these DIMM slots is populated. Later on, the owner decides to hotplug another DIMM, but puts it in slot 100 instead of slot 2. Now, nonlinear has a bit of a problem: the new DIMM happens to appear at a physical address 100 times higher address than the first DIMM. The `mem_map[]` for the new DIMM is split up properly, but the allocator bitmap's length is directly tied to the pfn, and thus the physical address of the memory.

Having already stated that the allocator bitmap stays at manageable sizes, this still does not seem like much of an issue. However, the physical address of that new memory *could* have an even greater range than 100 GB; it has the capability to have many, many terabytes of range, based on the hardware. Keeping allocator bitmaps for terabytes of memory could conceivably consume all system memory on a small machine, which is quite unacceptable. Nonlinear offers a solution to this by introducing a new way to represent a physical address: a fourth addressing scheme. With three addressing schemes already existing, a fourth seems almost comical, until its small scope is considered. The new scheme is isolated to use inside of a small set of core allocator functions a single place in the memory hotplug code itself. A simple lookup table converts these new "linear" pfns into the more familiar physical pfns.

5.2 Issues with `CONFIG_NONLINEAR`

Although it greatly simplifies several issues, nonlinear is not without its problems. Firstly, it does require the consultation of a small number of lookup tables during critical sections of code. Random access of these tables is likely to cause cache overhead. The more finely grained the units of hotplug, the larger these tables will grow, and the worse the cache effects.

Another concern arises with the size of the nonlinear tables themselves. While they allow pfns and `mem_map[]` to have nonlinear relationships, the nonlinear structures themselves remain normal, everyday, linear arrays. If hardware is encountered with sufficiently small hotplug units, and sufficiently large ranges of physical addresses, an alternate scheme to the arrays may be required. However, it is the authors' desire to keep the implementation simple, until such a need is actually demonstrated.

6 Memory Removal

While memory addition is a relatively black-and-white problem, memory removal has many more shades of gray. There are many different ways to use memory, and each of them has specific challenges for *un*using it. We will first discuss the kinds of memory that Linux has which are relevant to memory removal, along with strategies to go about unusing them.

6.1 "Easy" User Memory

Unusing memory is a matter of either moving data or simply throwing it away. The easiest, most straightforward kind of memory to remove is that whose contents can just be discarded. The two most common manifestations of this are clean page cache pages and swapped pages. Page cache pages are either dirty (containing information which has not been written to disk) or clean pages, which are simply a copy of something that *is* present on the disk. Memory removal logic that encounters a clean page cache page is free to have it discarded, just as the low memory reclaim code does today. The same is true of swapped pages; a page of RAM which has been written to disk is safe to discard. (Note: there is usually a brief period between when a page is written to disk, and when it is actually removed from memory.) Any page that *can* be swapped is also an easy candidate for memory removal, because it can easily be turned into a swapped page with existing code.

6.2 Swappable User Memory

Another type of memory which is very similar to the two types above is something which is only used by user programs, but is for some reason not a candidate for swapping. This at least includes pages which have been `mlock()`'d (which is a system call to prevent swapping). Instead of discarding these pages out of RAM, they must instead be moved. The algorithm to accomplish this should be very similar to the algorithm for a complete page swapping: freeze writes to the page, move the page's contents to another place in memory, change all references to the page, and re-enable writing. Notice that this is the same process as a complete swap cycle except that the writes to the disk are removed.

6.3 Kernel Memory

Now comes the hard part. Up until now, we have discussed memory which is being used by user programs. There is also memory that Linux sets aside for its own use and this comes in many more varieties than that used by user programs. The techniques for dealing with this memory are largely still theoretical, and do not have existing implementations.

Remember how the Linux page allocator can only keep track of pages in powers of two? The Linux slab cache was designed to make up for that [6], [7]. It has the ability to take those powers of two pages, and chop them up into smaller pieces. There are some fixed-size groups for common allocations like 1024, 1532, or 8192 bytes, but there are also caches for certain kinds of data structures. Some of these caches have the ability to attempt to shrink themselves when the system needs some memory back, but even that is relatively worthless for memory hotplug.

6.4 Removing Slab Cache Pages

The problem is that the slab cache's shrinking mechanism does not concentrate on shrinking any particular memory, it just concentrates on shrinking, period. Plus, there's currently no mechanism to tell *which* slab a particular page belongs to. It could just as easily be a simply discarded dcache entry as it could be a completely immovable entry like a `pte_chain`. Linux will need mechanisms to allow the slab cache shrinking to be much more surgical.

However, there will always be slab cache memory which is not covered by any of the shrinking code, like for generic `kmalloc()` allocations. The slab cache could also make efforts to keep these "mystery" allocations away from those for which it knows how to handle.

While the record-keeping for some slab-cache pages is sparse, there is memory with even more mysterious origins. Some is allocated early in the boot process, while other uses pull pages directly out of the allocator never to be seen again. If hot-removal of these areas is required, then a different approach must be employed: direct replacement. Instead of simply reducing the usage of an area of memory until it is unused, a one-to-one replacement of this memory is required. With the judicious use of page tables, the best that can be done is to preserve the virtual address of these areas. While this is acceptable for most use, it is not without its pitfalls.

6.5 Removing DMA Memory

One unacceptable place to change the physical address of some data is for a device's DMA buffer. Modern disk controllers and network devices can transfer their data directly into the system's memory without the CPU's direct involvement. However, since the CPU is not involved, the devices lack access to the CPU's virtual memory architecture. For this reason, all DMA-capable devices' transfers are based on the physical address of the memory to which they are transferring. Every user of DMA in Linux will either need to be guaranteed to not be affected by memory replacement, or to be notified of such a replacement so that it can take corrective action. It should be noted, however, that the virtualization layer on ppc64 can properly handle this remapping in its IOMMU. Other architectures with IOMMUs should be able to employ similar techniques.

6.6 Removal and the Page Allocator

The Linux page allocator works by keeping lists of groups of pages in sizes that are powers of two times the size of a page. It keeps a list of groups that are available for each power of two. However, when a request for a page is made, the only real information provided is for the *size* required, there is no component for specifically specifying which particular memory is required.

The first thing to consider before removing memory is to make sure that no other part of the system is using that piece of memory. Thankfully, that's exactly what a normal allocation does: make sure that it is alone in its use of the page. So, making the page allocator support memory removal will simply involve walking the same lists that store the page groups. But, instead of simply taking the first available pages, it will be more finicky, only "allocating" pages that are among those about to be removed. In addition, the allocator should have checks in the `free_pages()` path to look for pages which were selected for removal.

1. Inform allocator to catch any pages in the area being removed.
2. Go into allocator, and remove any pages in that area.
3. Trigger page reclaim mechanisms to trigger `free()`s, and hopefully unuse all target pages.
4. If not complete, goto 3.

6.7 Page Groupings

As described above, the page allocator is the basis for all memory allocations. However, when it comes time to remove memory a fixed size block of memory is what is removed. These blocks correspond to the sections defined in the the implementation of nonlinear memory. When removing a section of memory, the code performing the remove operation will first try to essentially allocate all the pages in the section. To remove the section, all pages within the section must be made free of use by some mechanism as described above. However, it should be noted that some pages will not be able to be made available for removal. For example, pages in use for kernel allocations, DMA or via the slab-cache. Since the page allocator makes no attempt to group pages based on usage, it is possible in a worst case situation that every section contains one in-use page that can not be removed. Ideally, we would like to group pages based on their usage to allow the maximum number of sections to be removed.

Currently, the definition of zones provides some level of grouping on specific architectures. For example, on i386, three zones are defined: DMA, NORMAL and HIGHMEM. With such definitions, one would expect most non-removable pages to be allocated out of the DMA and NORMAL zones. In addition, one would expect most HIGHMEM allocations to be associated with userspace pages and thus removable. Of course, when the page allocator is under memory pressure it is possible that zone preferences will be ignored and allocations may come from an alternate zone. It should also be noted that on some architectures, such as ppc64, only one zone (DMA) is defined. Hence, zones can not provide grouping of pages on every architecture. It appears that zones do provide some level of page grouping, but possibly not sufficient for memory hotplug.

Ideally, we would like to experiment with teaching the page allocator about the use of pages it is handing out. A simple thought would be to introduce the concept of sections to the allocator. Allocations of a specific type are directed to a section that is primarily used for allocations of that same type. For example, when allocations for use within the kernel are needed the allocator will attempt to allocate the page from a section that contains other internal kernel allocations. If no such pages can be found, then a new section is marked for internal kernel allocations. In this way pages which can not be easily freed are grouped together rather than spread throughout the system. In this way the page allocator's use of sections would be analogous to the slab caches use of pages.

7 Conclusion

The prevalence of hotplug-capable Linux systems is only expanding. Support for these systems will make Linux more flexible and will make additional capabilities available to other parts of the system.

Legal Statement

References

[1] *Five Nine at the IP Edge* `http://www.iec.org/online/tutorials/five-nines`

[2] Barham, Paul, et al. *Xen and the Art of Virtualization* Proceedings of the ACM Symposium on Operating System Principles (SOSP), October 2003.

[3] Waldspurger, Carl *Memory Resource Management in VMware ESX Server* Proceedings of the USENIX Association Symposium on Operating System Design and Implementation, 2002. pp 181–194.

[4] Dobson, Matthew and Gaughen, Patricia and Hohnbaum, Michael. *Linux Support for NUMA Hardware* Proceedings of the Ottawa Linux Symposium. July 2003. pp 181–196.

[5] Gorman, Mel *Understanding the Linux Virtual Memory Manager* Prentice Hall, NJ. 2004.

[6] Martin Bligh and Dave Hansen *Linux Memory Management on Larger Machines* Proceeedings of the Ottawa Linux Symposium 2003. pp 53–88.

[7] Bonwick, Jeff *The Slab Allocator: An Object-Caching Kernel Memory Allocator* Proceedings of USENIX Summer 1994 Technical Conference `http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html`