Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Efficient Use of the Page Cache with 64 KB Pages

Dave Kleikamp
IBM Linux Technology Center
shaggy@austin.ibm.com

Badari Pulavarty
IBM Linux Technology Center
pbadari@us.ibm.com

Abstract

In order for 64-bit processors to efficiently use large address spaces while maintaining lower TLB miss rates, the Linux® kernel can be configured with base page sizes up to 64 KB. While this benefits access to large memory segments and files, it greatly reduces the number of smaller files that can be resident in memory at one time. This paper proposes a change to the Linux kernel to allow file data to be more efficiently stored in memory when the size of the file, or the data at the end of a file, is significantly smaller than the page size.

1 Introduction

While 64 KB page support is not the primary topic of discussion in this paper, it does introduce the problem we are trying to address. We will take a quick look at rationale for using a larger page size.

1.1 Why use 64 KB pages?

Many processors use a fixed-size Translation Lookaside Buffer (TLB) to translate from virtual to physical addresses. This is a cache containing information from the kernel's page tables. When the needed TLB entry is not present for a memory translation, a TLB Miss occurs and the processor must go through an expensive operation of traversing the page tables and load the entry into the TLB [2]. While the amount of physical memory supported in recent systems has increased significantly, the TLB sizes remain relatively small. TLB coverage, the amount of memory accessible through cached mappings without incurring TLB misses, is becoming an important factor for applications with large working sets [1].

The use of larger page sizes is a well-known technique to reduce TLB misses. Linux's huge page support (`hugetlbfs`) is explicitly developed for this purpose. Unfortunately, huge pages require special handling and are too big for many uses.

Besides translation, the efficiency of page fault handling can be improved with larger page sizes. Due to a larger page size, applications end up requiring fewer page faults. A larger page size could also benefit hardware prefetching.

Performance analysis of various industry standard benchmarks showed significant gains (8-20%) with 64 KB page support.

1.2 Page Cache Fragmentation

An unfortunate side effect of a larger page size is internal fragmentation in the page cache. The

page cache will allocate a minimum of one page to cache the contents of a small file. The memory between the logical end of file and the end of the last page needed to cache the file is lost to fragmentation. When the page size is 4 KB, the fragmentation cost cannot exceed $4K-1$ bytes for any given file. With a page size of 64 KB, the fragmentation cost of a single file may be as great as $64K-1$ bytes.

This paper discusses changes to the page cache to allocate storage for file tails from a memory pool, allowing more efficient use of memory.

As of this writing, this project is at an early stage of development. There is no working prototype yet, but we expect to have a reasonable implementation and results in time for the presentation.

2 Alternate Approaches

Our initial goal was to separate the page cache from the page size. We considered making the page cache aware of multiple page sizes, the base page size and some smaller *fragment* size. One problem with this approach is how to represent the fragment. The simplest solution is to use the `page` structure. For a normal page, the kernel typically uses the `page` struct's position in the page table in order to determine the physical address of the page data. If we were to use the `page` struct to represent the fragment, we would have to add at least one more field into the structure to point to the backing storage. Every effort is made to keep the `page` struct as small as possible. Using a new structure to represent the fragment is also problematic. A lot of code within the Virtual File System (VFS) layer and the file systems themselves operate on the `page` struct. Any change to use another structure would prove to be very intrusive.

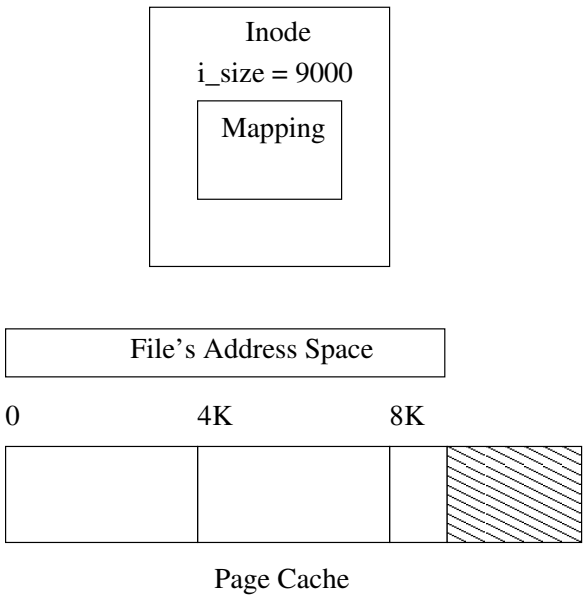

Figure 1: File Read into Page Cache

3 File Tails

Any file that has data resident in memory is represented by an `inode`, which in turn contains a data structure called a `mapping`. The `mapping` describes the address space of the file. Conceptually, the address space is a linear representation of a file bounded by the limits of the file; between offset zero and the size of the file (found in `i_size` in the inode).

For the majority of file systems in the Linux kernel, the data for the file is buffered in the page cache. The pages within the page cache are aligned to the address space of the file, and I/O is typically performed at the page level. When some data is read from disk, the kernel reads all the data in the pages that contain the data. There may be holes within the page, where no data is allocated on disk. In this case, the part of the page corresponding to the holes is zeroed. Likewise, when writing to disk, all dirty data within the pages containing the written data are written at one time.

Depending on the size of the file, the last page

within the address space of the file is usually only partially filled. (The remainder of the page is zero-filled, in case the file is extended.) This part of the file is what we call the File Tail. In the case of a file smaller than the base page size, the entire contents of the file will be in the tail.

When the page size is 4 KB, there is relatively little wasted memory in the page cache. For each cached file, less than 4 KB will be wasted between the end of the file and the end of the page containing the tail. When we switch to a 64 KB page size, each non-empty file will still require a minimum of one page to store the file data, but the space wasted in the page cache for each file may approach 64 KB. This will result in fewer files being able to be cached in the same amount of memory.

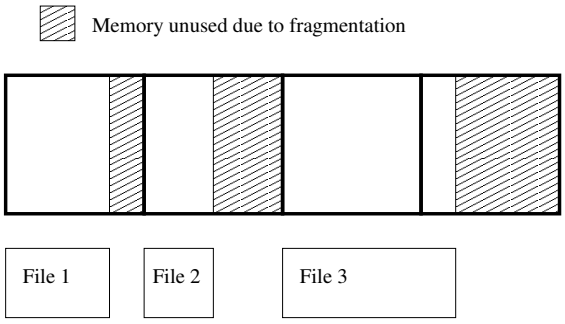

Figure 2: Page Cache Fragmentation with 4 KB pages

3.1 Alternate Storage for File Tails

We propose to provide an alternate method for caching the file tails. When the tail is sufficiently small, a buffer will be allocated from one or more memory pools, and a pointer to the buffer stored in the file's `mapping`.

In the case of a read, file system code (primarily in `mm/filemap.c`) will determine if the tail is resident in memory. If it is not, it will allocate the tail and read the data from disk. Then it will copy the data from the tail buffer to the user buffer.

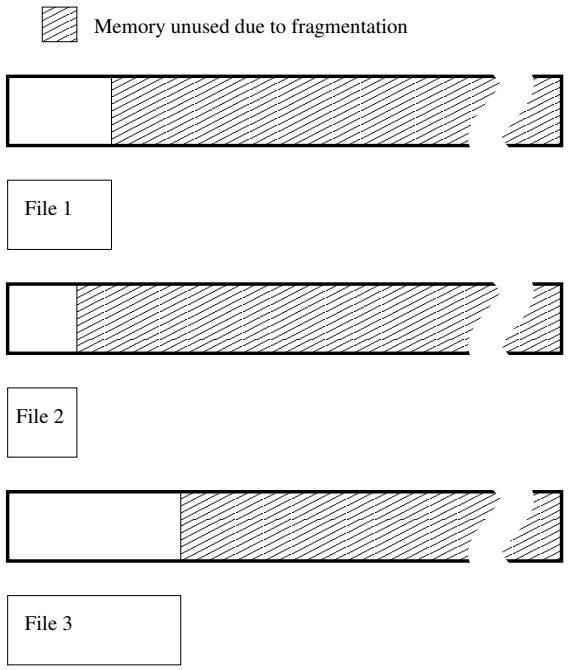

Figure 3: Page Cache Fragmentation with 64 KB pages

In the event that a full page is needed, the tail would be unpacked into a full page. Memory-mapping a region of a file containing the tail, writing to the tail, or an operation that increases the size of the file constitute actions that would require the tail be backed by a full page. Unpacking the tail consists of allocating a page, adding it to the page cache, copying the data from the tail buffer, zeroing the remainder of the page, and freeing the tail buffer.

3.2 Tail Allocation

Since the file tails will differ in size, and we want to store the tails as efficiently as possible, a single sized tail buffer will not satisfy our requirements. Two approaches we considered for addressing the issue are: piecing together a number of fixed sized buffers sufficient to store

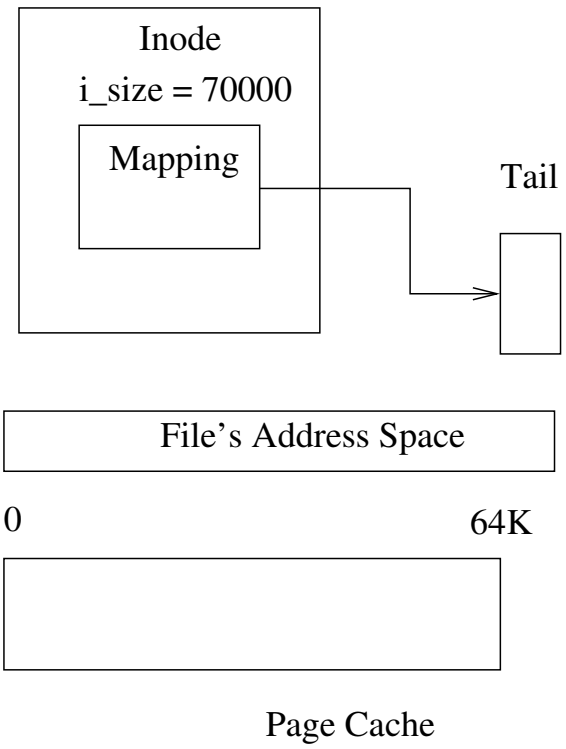

Figure 4: Tail Storage

the tail; or allocating the tail buffers from pools of different sized buffers.

The first approach requires storing pointers to multiple data buffers to store the tail. This could be done with either an array of pointers, or a linked list. The size of an array would depend on the fixed size of the individual buffers and the maximum length of a tail that we choose to store. For instance if we store the tails in 4 KB buffers, and choose tails that are 32 KB or smaller, we would need 8 members in the array. This array would either need to be stored within the mapping (`struct address_space`) or in a separately allocated buffer.

A linked list could handle any sized tail, but the list heads would need to be allocated somewhere. The obvious solution would be to allocate the list head and data buffer in a single allocation.

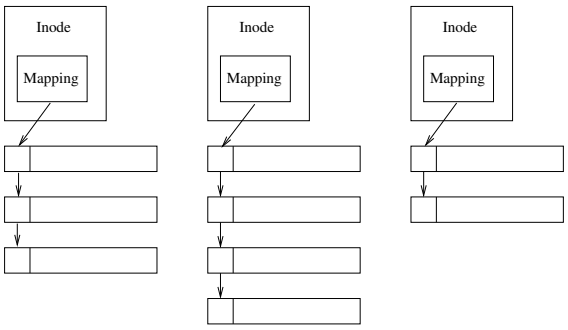

Figure 5: Tail in fixed-sized buffers

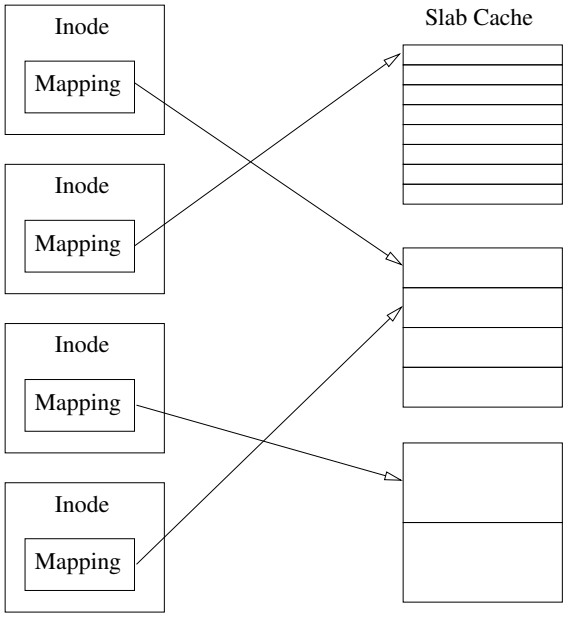

Figure 6: Tail in variable-sized buffers

The second approach allows each tail to be stored in one contiguous buffer. It requires a more complex allocator to allow different sized buffers to be allocated efficiently. Fortunately, such an allocator exists in `kmalloc`. For the initial implementation, we chose to simply use `kmalloc` and `kfree`.

Note that storing the tail data in the slab cache will always put it in low memory. This is not a real concern, since hardware supporting larger page sizes is 64-bit, so all physical memory is considered low memory. As is explained in the

next section, other design decisions are likely to make this feature incompatible with high-memory kernels in any case.

3.3 Tail I/O

Ideally, we want to avoid changing the file system interface. Reading file data is typically done through the `readpage()` address space operation which takes a `page` struct as an argument.

A simple, but inefficient, solution would be to read the data normally through the page cache, and pack the tail afterward. The disadvantage is the extra overhead involved in allocating the page, and copying the data. The ability to hold more small files in cache would probably justify this overhead if a better solution did not exist.

One solution is to allocate a dummy `page` struct that could be passed to `readpage()`. A new bitflag in `page->flags` would mark the page as a special container for the tail. `kmap()` and `kmap_atomic()` would have to be modified to recognize the flag, and return `page->mapping->tail` for the tail page. The use of the dummy `page` struct would have other benefits as well. The tail could then truly be represented in the page cache by having the page struct inserted into the radix tree. Note that the buffer allocated for the tail will need to be rounded up to the file system's block size, as I/O is performed in full disk blocks.

If such an approach were taken, the kernel configuration would have to ensure that the file tail support not be enabled on a high-memory-capable kernel. Although `kmap()` and `kmap_atomic()` may be easy to implement for tail pages, `kunmap()` and `kunmap_atomic()` do not take the page as an argument, and it would be difficult to guarantee their proper behavior.

A third possible approach to performing I/O on the tail data would be to introduce a new method to the address space operations that takes a pointer to a data buffer as an argument, rather than a page. This would require changes to any file system that wanted to take advantage of this feature, and will only be considered if other options turn out to be unworkable.

4 Limitations

As stated in the previous section, the implementation may depend on the kernel being built without high-memory support. Since this feature is primarily designed to address issues related to a large base page size, which are only implemented on 64-bit architectures, it is unlikely that this restriction will be problematic.

It is not a primary goal to support writing to packed tails. Any writes near the end of a file are likely to be followed by further writes that will extend past the end of the file forcing the tail to be unpacked anyway. However, we won't rule out the possibility of supporting this if it can be implemented with no additional overhead or complexity.

Memory-mapping a section of a file containing the tail will also result in the tail being unpacked. Protection is enforced per-page, so mapping a tail into an address space requires the tail be unpacked.

5 Future Work

As of this writing, the project is in a very early state, so much of what is described above can be considered future work. By the time this paper is presented, we expect to have a working code and performance results that we hope will justify our effort.

5.1 Page Allocation Revisited

We may want to re-evaluate the mechanism for allocating the tail buffers. Since the `kmalloc` slab is used as a general purpose memory allocator, data for the tails may be interspersed with other data within a physical page. File tails are easily reclaimable, so using a separate allocator is more likely to allow reclaim to free complete pages. It may prove to be beneficial to define several independent slab caches of different sizes that would be used only for tail buffers.

5.2 Memory Map Support

We may want to investigate whether it would be possible to allow some degree of memory mapping support against a tail. At the very least we should be able to delay unpacking the tail until the corresponding page is first referenced.

5.3 Tail Repacking

Data at the end of a file may occupy a full page if it had been recently written or memory-mapped. If the data has been written, leaving the page no longer dirty, or the page is no longer memory-mapped, it may be useful to pack the data into a tail buffer.

This would reduce the memory usage for these cached files, and increase the chance that the data will still be in memory if it is accessed again. A good heuristic is needed to ensure that tails are not packed and unpacked too often.

6 Conclusion

This paper proposes a solution to the problem of internal fragmentation in the page cache on kernels with a large page size. We intend to implement the proposal and present performance results on a number of industry standard benchmarks. We believe that this work will make it possible for more workloads to benefit from a large page size.

Legal Statement

References

[1] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002. http://www.usenix.org.

[2] Simon Winwood, Yefim Shuf, and Hubertus Franke. Multiple page size support in the linux kernel. *Proceedings of the Ottawa Linux Symposium*, June 2002.