Understanding Caching

Jan 01, 2004 By James Bottomley (/user/801293)
in

Architectures that support Linux differ in how they handle caching at the hardware level. Here's how the kernel gets the best possible use out of every cache design.

(/issue/117)

From Issue #117
January 2004 (/issue/117)

The Aliasing Problem

Any time the kernel sets up more than one virtual mapping for the same physical page, cache line aliasing may occur. The kernel is careful to avoid aliasing, so it usually occurs only in one particular instance: when the user mmaps a file. Here, the kernel has one virtual address for pages of the file in the page cache, and the user may have one or more different virtual addresses. This is possible because nothing prevents the user from mmaping the file at multiple locations.

When a file is mmaped, the kernel adds the mapping to one of the inode's lists: i_mmap, for maps that cannot change the underlying data, or i_mmap_shared, for maps that can change the file's data. The API for bringing the cache aliases of a page into sync is:

```
void flush_dcache_page(struct page *page);
```

This API must be called every time data on the page is altered by the kernel, and it should be called before reading data from the page if page->mapping->i_mmap_shared is not empty. In architecture-specific code, flush_dcache_page loops over the i_mmap_shared list and flushes the cache data. It then loops over the i_mmap list and invalidates it, thus bringing all the aliases into sync.

Separate Instruction and Data Caches

In their quest for efficiency, processors often have separate caches for the instructions they execute and the data on which they operate. Often, these caches are separate mechanisms, and a data write may not be seen by the instruction cache. This causes problems if you are trying to execute instructions you just wrote into memory, for example, during module loading or when using a trampoline. You must use the following API:

```
void
flush_icache_range(unsigned long start,
                   unsigned long end);
```

to ensure that the instructions are seen by the instruction cache prior to execution. *start* and *end* are the starting and ending addresses, respectively, of the block of memory you modified to contain your instructions.

General Cache Flushing

Two APIs globally flush the CPU caches:

```
void flush_cache_all(void);
```

and

```
void flush_cache_mm(struct mm_struct *mm);
```

These flush all the caches in the system and only the lines in the cache belonging to the particular process address space *mm*. Both of these are extremely expensive operations and should be used only when absolutely necessary.

Caching in SMP Environments

When more than one CPU is in the system, a level of caching usually exists that is unique to each CPU. Depending on the architecture, it may be the responsibility of the kernel to ensure that changes in the cache of one CPU become visible to the other CPUs. Fortunately, most CPUs handle this type of coherency problem in hardware. Even if they don't, as long as you follow the APIs listed in this article, you can maintain coherency across all the CPUs.

Conclusion

I hope I've given you a brief overview of how caches work and how the kernel manages them. The contents of this article should be sufficient for you to understand caching in most kernel programming situations you're likely to encounter. Be aware, however, that if you get deeply into the guts of kernel cache management (particularly in the architecture-specific code), you likely will come across concepts and APIs not discussed here.

James Bottomley is the software architect for SteelEye. He maintains the SCSI subsystem, the Linux Voyager port and the 53c700 driver. He also has made contributions to PA-RISC Linux development in the area of DMA/device model abstraction.

Comments

Comment viewing options
Threaded list - expanded ▼ Date - newest first ▼ 50 comments per page ▼ Save settings
Select your preferred way to display the comments and click "Save settings" to activate your changes.

Very good article, but ... wh (/article/7105#comment-27905)
Submitted by Anonymous (not verified) on Thu, 04/07/2005 - 04:27.

Very good article, but ... where is figure 4 ?

figure 4 is just like the (/article/7105#comment-320027)
Submitted by Anonymous (not verified) on Wed, 03/05/2008 - 04:16.

figure 4 is just like the tag(hidden!) :#)