
# Direct I/O introduce

James Wang
QA Engineer
jnwang@suse.com

# Agenda

## File I/O patten performance matrix

# Matrix

| Loop=0xFFFF | close | fsync | fdatasync |
|---|---|---|---|
| O_DIRECT | Real = 1.855s<br>User = 0.008s<br>Sys = 0.656s | real 0m1.824s<br>user0m0.016s<br>sys 0m0.620s | real 0m1.861s<br>user0m0.000s<br>sys 0m0.684s |
| O_SYNC | real 4m22.478s<br>user0m0.184s<br>sys 0m7.920s | real 4m24.383s<br>user0m0.200s<br>sys 0m8.508s | real 4m24.720s<br>user0m0.208s<br>sys 0m8.612s |
| NULL | real 0m0.087s<br>user0m0.000s<br>sys 0m0.084s | real 0m0.329s<br>user0m0.000s<br>sys 0m0.092s | real 0m0.317s<br>user0m0.004s<br>sys 0m0.088s |

# What is Direct I/O?

**The idea behind direct I/O is that data blocks move directly between the storage device and user-space memory without going through the page cache.**

**Direct I/O is sometimes misunderstood as being a means of obtaining fast I/O performance. However, for most applications, using direct I/O can considerably degrade performance. This is because the kernel applies a number of optimizations to improve the performance of I/O done via the buffer cache, including performing sequential read-ahead, performing I/O in clusters of disk blocks, and allowing processes accessing the same file to share buffers in the cache. All of these optimizations are lost when we use direct I/O. Direct I/O is intended only for applications with specialized I/O requirements. For example, database systems that perform their own caching and I/O optimizations don't need the kernel to consume CPU time and memory performing the same tasks.**

# Why do the people use this feature?

**Developers use direct memory for either (or both) of two reasons:**

**(1) they believe they can manage caching of file contents better than the kernel can.**

**(2) they want to avoid overflowing the page cache with data which is unlikely to be of use in the near future.**

**It is a relatively little-used feature which is often combined with another obscure kernel capability: asynchronous I/O.**

**The biggest consumers, by far, of this functionality are large relational database systems, so it is not entirely surprising that a developer currently employed by Oracle is working in this area.**

# Where is it used by?

**When doing video streaming on small embedded systems (small = 32MB RAM, slow processor, no MMU), kernel read-ahead and write-behind turn out to be problematic because they cause too much memory pressure and in a rather lumpy way (which is the real problem - memory allocation failures start happening, and the I/O rate is very variable from one second to the next).**

**But not having read-ahead and write-behind makes latency too high, unless asynchronous I/O is used to keep the queues full. Asynchronous I/O doesn't work on Linux except with direct I/O. So we're dabbling in asynchronous, direct I/O for video streaming on small devices to make it more reliable.**

# READ/WRITE

**Read is sync.**

**WRITE is either sync or async.**

# O_DIRECT/O_SYNC

**O_DIRECT alone only promises that the kernel will avoid copying data from user space to kernel space, and will instead write it directly via DMA (Direct memory access; if possible). Data does not go into caches. There is no strict guarantee that the function will return only after all data has been transferred.**

**O_SYNC guarantees that the call will not return before all data has been transferred to the disk (as far as the OS can tell). This still does not guarantee that the data isn't somewhere in the hard-disk write cache, but it is as much as the OS can guarantee.**

***It does guarantee that the data has been transferred to the medium. The kernel will set the FUA (Force Unit Access) flag on the write if available, or it will send a separate command to flush the write cache.***

# When Should You Fsync?

**There are some simple rules to follow to determine whether or not an fsync() call is necessary. First and foremost, you must answer the question: is it important that this data is saved now to stable storage? If it's scratch data, then you probably don't need to fsync(). If it's data that can be regenerated, it might not be that important to fsync() it. If, on the other hand, you're saving the result of a transaction, or updating a user's configuration file, you very likely want to get it right. In these cases, use fsync().**

**The more subtle usages deal with newly created files, or overwriting existing files. A newly created file may require an fsync() of not just the file itself, but also of the directory in which it was created (since this is where the file system looks to find your file). This behavior is actually file system (and mount option) dependent. You can either code specifically for each file system and mount option combination, or just perform fsync() calls on the directories to ensure that your code is portable.**

**Similarly, if you encounter a system failure (such as power loss, ENOSPC or an I/O error) while overwriting a file, it can result in the loss of existing data. To avoid this problem, it is common practice (and advisable) to write the updated data to a temporary file, ensure that it is safe on stable storage, then rename the temporary file to the original file name (thus replacing the contents). This ensures an atomic update of the file, so that other readers get one copy of the data or another. The following steps are required to perform this type of update:**

**1)create a new temp file (on the same file system!)**

**2)write data to the temp file**

**3)fsync() the temp file**

**4)rename the temp file to the appropriate name**

**5)fsync() the containing directory**

# Talking is cheap

**Stop here.**

# Open – start of your IO

```
1032 SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
1033 {
1034     if (force_o_largefile())
1035         flags |= O_LARGEFILE;
1036
1037     return do_sys_open(AT_FDCWD, filename, flags, mode);
1038 }
1039
1040 SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
1041         umode_t, mode)
1042 {
1043     if (force_o_largefile())
1044         flags |= O_LARGEFILE;
1045
1046     return do_sys_open(dfd, filename, flags, mode);
1047 }
```

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
    struct filename *tmp;

    if (fd)
        return fd;

    tmp = getname(filename);
    if (IS_ERR(tmp))
        return PTR_ERR(tmp);

    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        struct file *f = do_filp_open(dfd, tmp, &op);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
            fd = PTR_ERR(f);
        } else {
            fsnotify_open(f);
            fd_install(fd, f);
        }
    }
    putname(tmp);
    return fd;
}
```

# Request queue

**Request queues keep track of outstanding block I/O requests. But they also play a crucial role in the creation of those requests. The request queue stores parameters that describe what kinds of requests the device is able to service: their maximum size, how many separate segments may go into a request, the hardware sector size, alignment requirements, etc. If your request queue is properly configured, it should never present you with a request that your device cannot handle.**

# The Anatomy of a Request

**In our simple example, we encountered the request structure. However, we have barely scratched the surface of that complicated data structure. In this section, we look, in some detail, at how block I/O requests are represented in the Linux kernel.**

**Each request structure represents one block I/O request, although it may have been formed through a merger of several independent requests at a higher level. The sectors to be transferred for any particular request may be distributed throughout main memory, although they always correspond to a set of consecutive sectors on the block device. The request is represented as a set of segments, each of which corresponds to one in-memory buffer. The kernel may join multiple requests that involve adjacent sectors on the disk, but it never combines read and write operations within a single request structure. The kernel also makes sure not to combine requests if the result would violate any of the request queue limits described in the previous section.**


SUSE®
We adapt. You succeed.

# Internal Use Slides

# **Presentation Title**(36pt)

Subhead or Second Line (20pt)

Presenter Name (14pt)
Title
Company/Email