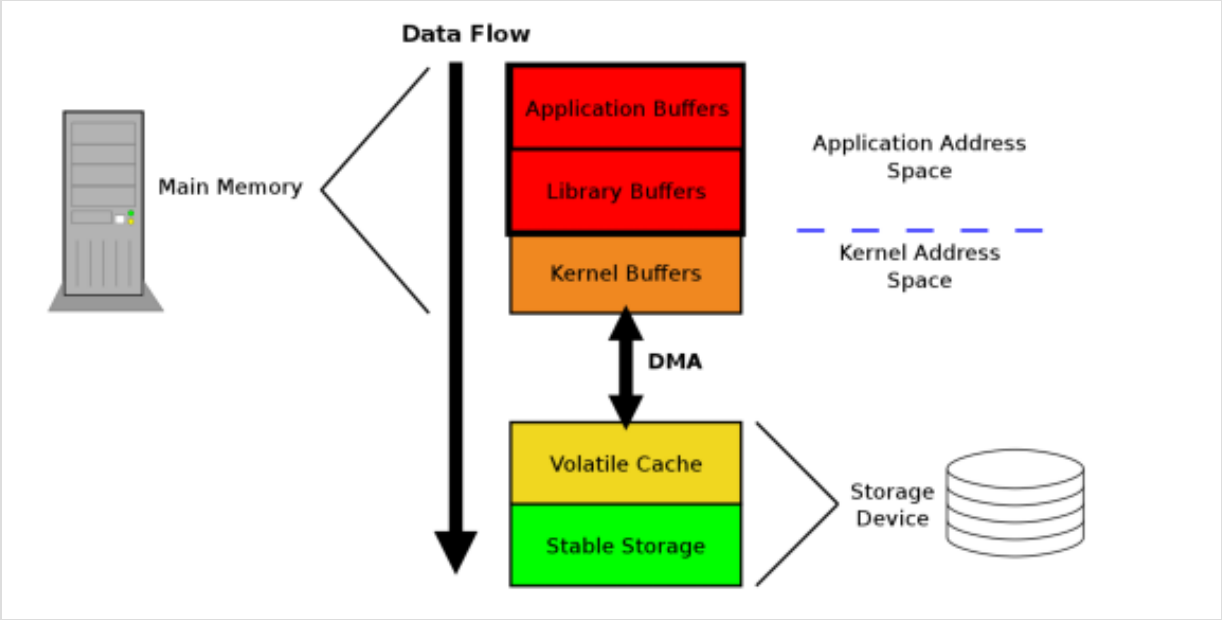
确保数据到达磁盘

Posted on 2019-02-26

> 本文主要讲述在Linux服务上数据从应用层写入磁盘的关键路径，聚焦于其中涉及到的各种缓存，讲述在c编程中如何确保数据能被持久化存储，避免系统异常时造成数据丢失。

IO buffering

Linux下，用户数据在最终达到持久化存储层之前会经过多层，如下所示：



最顶层是需要存入数据的应用，数据首先是以block的方式存储在应用本身的memory或者buffer中，这些buffer也可能会被转交给运行库，由运行库来管理这些缓存。无论这些数据是在应用程序本身的buffer或者运行库的buffer中，此时这些数据都还是停留在用户地址空间。用户空间的下一层是内核空间，它也会内存中维护自身的写回缓存，即page cache。脏页在page cache中停留一段时间后会被写入存储设备中（如硬盘）。同时存储设备也可能会维护自身的易失性缓存，在掉电时缓存中的数据是会丢失的。最后，在最底层的是非易失性存储设备，数据只有在到达此层时，才是安全的，不会在系统异常退出时丢失。

为了更详细阐述如上所述的各种缓存机制，以这样的一个应用程序为例：它通过一个socket读入数据后将数据写入本地文件中。在关闭这个socket之前，服务端会确认数据已经持久化存储，主要代码如下所示：

```
1   0 int
```

```
1 sock_read(int sockfd, FILE *outfp, size_t nrbytes)
2 {
3       int ret;
4       size_t written = 0;
5       char *buf = malloc(MY_BUF_SIZE);
6
7       if (!buf)
8               return -1;
9
10      while (written < nrbytes) {
11              ret = read(sockfd, buf, MY_BUF_SIZE);
12              if (ret =< 0) {
13                      if (errno == EINTR)
14                              continue;
15                      return ret;
16              }
17              written += ret;
18              ret = fwrite((void *)buf, ret, 1, outfp);
19              if (ret != 1)
20                      return ferror(outfp);
21      }
22
23      ret = fflush(outfp);
24      if (ret != 0)
25              return -1;
26
27      ret = fsync(fileno(outfp));
28      if (ret < 0)
29              return -1;
30      return 0;
31 }
32
```

在接受client端的连接之后，应用会将从socket中读出的数据写入自身buffer之中，如上函数的调用者在调用前已经知道client端发送的数据大小，同时打开一个文件流用以写入数据，该函数在返回前会确认将数据持久化存储。其中L5就是一个应用层的buffer，从socket中读出的数据首先会放入这个buffer中，同时鉴于网络传输的突发性或者低效，我们决定使用libc的流式函数（`fwrite()`及`fflush()`,代表上图中的运行库缓存）来缓存应用层读出的数据。L10–21就是用于从socket读出数据并写入文件流中，L23用以将文件流进行刷写，使数据进入内核空间。之后，在L27，数据被写入上图的`Stable Storage`这层。

IO APIs

下面我们来看各个API同上图中每层的关系，在下面的讨论中，我们将IO分成3类：系统IO（system IO）、流式IO以及内存映射IO（mmap）。

系统IO

系统IO是指通过系统调用将内核空间的数据写入storage层的操作，如下是部分相关的系统IO接口：

操作	相关函数
Open	open() create()
Write	write() aio_write() pwrite() pwritev()
Sync	fsync() sync()
Close	close()

流式IO

流式IO是指调用C的运行库中流式接口的IO操作，调用这些函数进行数据写入时可能不会引发系统调用，这意味着数据仍在用户空间中，如下是部分相关的流式IO接口：

操作	相关函数
Open	fopen() fdopen() freopen()
Write	fwrite() fputc() fputs() puts() putc() putcha()
Sync	fflush()
Close	fclose()

内存映射IO

内存映射文件同前文的系统IO比较相似，文件仍旧是通过相同的接口来打开和关闭，它是通过将文件数据通用户空间映射来实现文件访问，然后执行同其他应用层buffer一样的内存读写操作来读写文件，如下是部分相关的mmap接口：

操作	相关函数
Open	open() create()

Map	mmap()
Write	memcpy() memmove() read() 或者其他操作应用层缓存的函数
Sync	msync()
Unmap	munmap()
Close	close()

在打开文件时，有两个选项可以改变时的缓存行为:`O_SYNC`及`O_DIRECT`。使用`O_DIRECR`打开的文件的读写操作会绕过内核空间的`page cache`,直接将数据写入存储设备中，但是存储设备自身仍可能存在缓存，所以仍旧需要使用`fsync()`来将数据持久化存储,即`O_DIRECR`只和系统IO的API相关。裸设备(/dev/raw/raw/V)提供一种特殊的`O_DIRECT` IO方式，这些设备在打开时不需指定`O_DIRECT`选项，但仍旧提供`direct IO`语义。

同步IO是指对于一个使用`O_SYNC`或者`O_DSYNC`打开的IO操作（包括不管是否使用了`O_DIRECT`的系统IO及流式IO），POSIX语义定义了一下几种同步操作模式：

- O_SYNC：文件数据及所有元数据被同步写入磁盘中
- O_DSYNC：只有文件数据及访问该数据需要的元数据需要被同步写入磁盘
- O_RSYNC：尚未实现

  在同步模式下，用户数据及相关的元数据会立马被写入持久化存储设备中，需要注意的是，其他元数据（不涉及到访问该部分数据）可能不会立马被写入持久化设备中，这些元数据可能包括文件的访问时间、创建时间或者修改时间等。

  另外需要注意使用`O_SYNC`或`O_DSYNC` 打开一个文件，然后通过libc的流式接口来操作这个文件的情况，通过`fwrite()`写入的数据都会被从的运行库缓存，直到调用`fflush()`后数据才被写入磁盘中。因此，通过此类流式解救操作一个同步文件描述符时，不需要调用`fsync()`来同步数据，但是`fflush()`仍旧是必需的。

合理使用fsync

通过如下几条原则来判断是否调用`fsync()`：

- 首先，将数据持久化存储是否那么重要。如果是可擦洗或可再生的数据，那是没必要的
- 在创建新文件或者覆盖现有文件时，使用`fsync()`不止是同步文件数据本身，同时也是同步它的目录项，

才能确保之后能访问到这个文件，这个行为同文件系统及挂载选项也是相关的。

- 最后，如果在覆盖现有文件时系统了奔溃，可能会造成数据的丢失，为了解决这个问题，一个通用做法是先将数据写入一个临时文件，确保这个临时文件持久化存储后将这个临时文件重命名为待覆盖文件名，这样能确保文件的原子更新。相关的流程如下：
    - 创建一个临时文件
    - 将数据写入临时文件
    - `fsync()`这个临时文件
    - 将临时文件重命名
    - `fsync()`文件所在目录

写回(write-back)缓存

本节简单讨论下磁盘层面的缓存以及操作系统对于此种缓存的控制，本节中讨论的选项不影响应用程序该如何构造。

存储设备的写回缓存有许多不同的形式，如前文所述的易失性缓存，此类缓存在系统异常时会丢失。在实际中，大部分存储设备都可以被配置为无缓存模式或者写穿(`write-through`)模式，这些模式在数据被持久化存储之前是不会返回成功给客户端的。此外一些外部存储阵列可能有非易失性缓存或者带电的写入缓存，这样可以在系统掉电时仍旧持久化存储数据。

一些文件系统提供控制缓存刷写的挂载选项，例如在2.6.35之后的linux版本中，ext3、ext4及btrfs提供`-o barrier`这个选项来打开屏障（`write-back cache`，该选项是默认开的），或者`-o nobarrier`来关闭它。但是应用层不需要过多考虑这个选项，当文件系统的barriers被禁用后，`fsync`不会导致磁盘缓存的刷写。

O_DIRECT同O_SYNC的区别

如前文所述，`O_DIRECT`可以是IO绕过`page cache`直达`storage层`，但是`storage`层可能仍旧会存在缓存，此时数据仍旧可能是不安全的。但是使用`O_DIRECT`需要遵守一些限制：

- 用以传递数据的应用层缓存区，其内存边界必须对其块大小的整数倍
- 数据传输的开始点，即在文件中的偏移值必须也是块大小的整数倍
- 传输的数据长度必须是块大小的整数倍
  这些限制都要有应用层来确保，否则会导致EINVAL错误，显而易见的是，使用O_DIRECT容易造成存储空间的浪费

而`O_SYNC`用以将缓存中的数据刷写至磁盘中，此时数据还是会写入`page cache`中，但是会被立马刷写至磁盘中，直到数据持久化存储才会返回，可以确保数据的安全性。但是通过`fsync`的manual手册可以看到在一些老的内核或者小众文件系统中，`fsync`可能不知道如何刷写存储设备的缓存，此时需要别的方式来关闭存储设备的缓

存，如下所示：

```
NOTES
       On some UNIX systems (but not Linux), fd must be a writable file descriptor.

       In Linux 2.2 and earlier, fdatasync() is equivalent to fsync(), and so has no performance advantage.

       The fsync() implementations in older kernels and lesser used filesystems does not know how to flush disk caches.  In these cases disk caches need to be disabled using  hdparm(8)
       or sdparm(8) to guarantee safe operation.
```

在实际使用过程可以同时使用`O_DIRECT`及`O_DSYNC`这个两个选项来确保数据的安全性，这样数据在写入时会直接绕过page cache，持久化存储后才会返回。

References

- Ensuring data reaches disk
- How are the O_SYNC and O_DIRECT flags in open(2) different/alike

linux # storage