zhuanlan.zhihu.com /p/374627314

Linux 文件 I/O 进化史（三）：Direct I/O 和 Linux AIO

Direct I/O

前面介绍的 buffered I/O 和 mmap，访问文件都是需要经过内核的 page cache。这对于数据库这种 self-caching 的应用可能不是特别友好：

1. 用户层的 cache 和内核的 page cache 其实是重复的，导致内存浪费。
2. 数据的传输：disk -> page cache -> user buffer 需要两次内存拷贝。

为此，Linux 提供了一种绕过 page cache 读写文件的方式：direct I/O。

要使用 direct I/O 需要在 open 文件的时候加上 O_DIRECT 的 flag。比如：

```
int fd = open(fname, O_RDWR | O_DIRECT);
```

使用 direct I/O 有一个很大的限制：buffer 的内存地址、每次读写数据的大小、文件的 offset 三者都要与底层设备的逻辑块大小对齐（一般是 512 字节）。

> Since Linux 2.6.0, alignment to the logical block size of the underlying storage (typically 512 bytes) suffices. The logical block size can be determined using the ioctl(2) BLKSSZGET operation or from the shell using the command: blockdev --getss

```
$ blockdev --getss /dev/vdb1
512
```

如果不满足对齐要求，系统会报 EINVAL（Invalid argument。） 错误：

```
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
  int fd = open("/tmp/direct_io_test", O_CREAT | O_RDWR | O_DIRECT);
  if (fd < 0) {
    perror("open");
    return -1;
  }
```

```
constexpr int kBlockSize = 512;

// buffer 地址不对齐
{
  char* p = nullptr;
  int ret = posix_memalign(
      (void**)&p, kBlockSize / 2,
      kBlockSize);  // 这里有一定概率可以分配出与 kBlockSize 对齐的内存
  assert(ret == 0);
  int n = write(fd, p, kBlockSize);
  assert(n < 0);
  perror("write");
  free(p);
}

// buffer 大小不对齐
{
  char* p = nullptr;
  int ret = posix_memalign((void**)&p, kBlockSize, kBlockSize / 2);
  assert(ret == 0);
  int n = write(fd, p, kBlockSize / 2);
  assert(n < 0);
  perror("write");
  free(p);
}

// 文件 offset 不对齐
{
  char* p = nullptr;
  int ret = posix_memalign((void**)&p, kBlockSize, kBlockSize);
  assert(ret == 0);
  off_t offset = lseek(fd, kBlockSize / 2, SEEK_SET);
  assert(offset == kBlockSize / 2);
  int n = write(fd, p, kBlockSize);
  assert(n < 0);
  perror("write");
  free(p);
}

// 三者对齐
{
  char* p = nullptr;
  int ret = posix_memalign((void**)&p, kBlockSize, kBlockSize);
```

```
    assert(ret == 0);
    off_t offset = lseek(fd, 0, SEEK_SET);
    assert(offset == 0);
    int n = write(fd, p, kBlockSize);
    assert(n == kBlockSize);
    printf("write ok\n");
    free(p);
  }

  return 0;
}
```

Direct I/O 与数据持久化

使用 buffered I/O 写入数据，write 返回之后，数据实际上只到达 page cache —— 还在内存中，需要等内核线程周期性将脏页刷新到持久化存储上，或应用程序调用 fsync 主动将数据刷新脏页。

理论上，使用 direct I/O 不会经过 page cache，当 write 返回之后，数据应该达到持久化存储。 但是，除了文件数据本身，文件的一些重要的元数据，比如文件的大小，也会影响数据的完整性。而 direct I/O 只是对文件数据本身有效，文件的元数据读写还是会经过内核缓存 —— 使用 direct I/O 读写文件，依然需要使用 fsync 来刷新文件的元数据。

但是，并不是所有文件元数据都会影响到数据的完整性，比如文件的修改时间。为此，MySQL 提供了一个新的刷盘参数：O_DIRECT_NO_FSYNC —— 使用 O_DIRECT 读写数据，只有在必要的使用进行 fsync。

> As of MySQL 8.0.14, fsync() is called after creating a new file, after increasing file size, and after closing a file, to ensure that file system metadata changes are synchronized. The fsync() system call is still skipped after each write operation.

Linux AIO

前面介绍的，无论是调用 read/write 读写文件（包括 buffered I/O 和 direct I/O），还是使用 mmap 映射文件，都是属于同步文件 I/O。

同步 I/O 接口的优点是：简单易用、逻辑清晰。但是除了这个，同步 I/O 似乎没别的优点了。同步的 I/O 接口，如果没有命中 page cache，会导致线程阻塞 => 这种情况下，往往会使用多线程来解决 => 随着 SSD 等持久化设备的性能提升，继续增加线程数 => 大量线程的基础开销 + 大量上下文切换 + 内核调度器的负载 => 系统性能很差 => 需要异步文件 I/O 来提升高并发读写时的性能。

Linux AIO 是内核提供的一套支持文件异步 I/O 的接口（只支持使用 O_DIRECT 的方式来读写文件）：

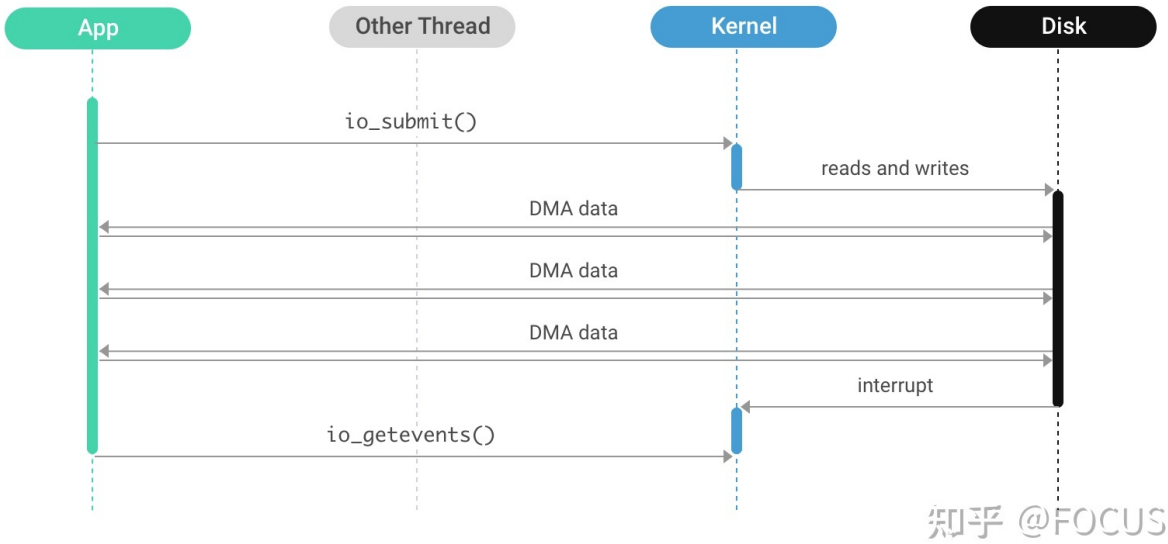
```
int io_setup(unsigned nr_events, aio_context_t *ctx_idp);
int io_submit(aio_context_t ctx_id, long nr, struct iocb **iocbpp);
int io_getevents(aio_context_t ctx_id, long min_nr, long nr,
                 struct io_event *events, struct timespec *timeout);
```

```
int io_cancel(aio_context_t ctx_id, struct iocb *iocb,
                      struct io_event *result);
int io_destroy(aio_context_t ctx_id);
```

1. `io_setup` 创建一个能支持 `nr_events` 个操作的异步 I/O context。
2. `io_submit` 提交异步 I/O 请求。
3. `io_getevents` 获取已完成的异步 I/O 结果。
4. `io_cancel` 取消之前提交的某个异步 I/O 请求。
5. `io_destroy` 取消所有提交的异步 I/O 请求，并销毁异步 I/O context。

正常情况下，Linux AIO 的流程如下：

1. 调用 `io_setup` 创建一个 I/O context 用于提交和收割 I/O 请求。
2. 创建 1~n 和 I/O 请求，调用 `io_submit` 提交请求。
3. I/O 请求执行完成，通过 DMA 直接将数据传输到 user buffer。
4. 调用 `io_getevents` 收割已完成的 I/O。
5. 重新执行第 2 步，或者确认不需要继续执行 AIO，调用 `io_destroy` 销毁 I/O context。



（图片来自 https://www.scylladb.com/2017/10/05/io-access-methods-scylla/）

这里用一个简单的例子介绍一下 Linux AIO 接口的基本使用方法。各个 Linux AIO 接口的详细介绍，建议参考相关的 Linux Manual Page。 这里要注意的是：

1. glibc 并没有提供 Linux AIO 系统调用接口的封装，使用的时候需要使用 syscall 简单封装一下。
2. libaio 是对 Linux AIO 的封装，接口很像但是不完全一样，不要混淆。

```
#include <assert.h>
#include <fcntl.h>
#include <inttypes.h>
#include <linux/aio_abi.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <sys/syscall.h>
#include <unistd.h>

int io_setup(unsigned nr, aio_context_t *ctxp) {
  return syscall(__NR_io_setup, nr, ctxp);
}

int io_destroy(aio_context_t ctx) { return syscall(__NR_io_destroy, ctx); }

int io_submit(aio_context_t ctx, long nr, struct iocb **iocbpp) {
  return syscall(__NR_io_submit, ctx, nr, iocbpp);
}

int io_getevents(aio_context_t ctx, long min_nr, long max_nr,
                 struct io_event *events, struct timespec *timeout) {
  return syscall(__NR_io_getevents, ctx, min_nr, max_nr, events, timeout);
}

int main(int argc, char *argv[]) {
  int fd = open("/tmp/linux_aio_test", O_RDWR | O_CREAT | O_DIRECT, 0666);
  if (fd < 0) {
    perror("open");
    return -1;
  }

  aio_context_t ctx = 0;
  int ret = io_setup(128, &ctx);
  if (ret < 0) {
    perror("io_setup");
    return -1;
  }

  struct iocb cb;
  memset(&cb, 0, sizeof(cb));
  cb.aio_fildes = fd;
  cb.aio_lio_opcode = IOCB_CMD_PWRITE;
  char *data = nullptr;
  ret = posix_memalign((void **)&data, 512, 4096);
  assert(ret == 0);
  memset(data, 'A', 4096);
  cb.aio_buf = (uint64_t)data;
  cb.aio_offset = 0;
```

```
  cb.aio_nbytes = 4096;

  struct iocb *cbs[1];
  cbs[0] = &cb;
  ret = io_submit(ctx, 1, cbs);
  if (ret != 1) {
    free(data);
    perror("io_submit");
    return -1;
  }

  struct io_event events[1];
  ret = io_getevents(ctx, 1, 1, events, nullptr);
  assert(ret == 1);
  printf("%ld %ld\n", events[0].res, events[0].res2);
  free(data);
  ret = io_destroy(ctx);
  if (ret < 0) {
    perror("io_destroy");
    return -1;
  }
  return 0;
}
```

小结

Linux AIO 是 Linux 下一个尝试解决文件异步 I/O 的解决方案，但是这个方案并不彻底、不完美。

- **Linux AIO 只支持 direct I/O。** 这意味着使用 Linux AIO 的所有读写操作都要受到 direct I/O 的限制：1）buffer 地址、buffer 大小、文件 offset 的对齐限制；2）无法使用 page cache。
- **不完备的异步，仍然有可能被阻塞。** 比如在 ext4 文件系统上，如果需要读取文件的元数据，此时调用可能会被阻塞。
- **较大的参数拷贝开销。** 每个 I/O 提交需要拷贝一个 64 字节的 `struct iocb` 对象；每个 I/O 完成需要拷贝一个 32 字节的 `struct io_event` 对象；一个 I/O 请求总共需要 96 字节的拷贝。这个拷贝开销是否可以承受，和单次 I/O 的大小有关：如果单次 I/O 本身就很大，相较之下，这点消耗可以忽略；而在大量小 I/O 的场景下，这样的拷贝影响比较大。
- 多线程提交或收割 I/O 请求会对 io_context_t 造成比较大的锁竞争。
- 每个 I/O 需要两次系统调用才能完成（`io_submit` 和 `io_getevents`），大量小 I/O 难以接受 —— 不过 `io_submit` 和 `io_getevents` 都支持批量操作，可以通过批量提交和批量收割来减少系统调用。

Linux AIO 从诞生之日（内核 2.5 版本）起到现在（2021 年），一直在不断完善。但大部分情况下，都是一些修修补补的工作，Linux AIO 一直都没有被实现完整 —— 特别是只支持 direct I/O，异步接口仍然可能被阻塞。

为了彻底解决 Linux AIO 在设计上的不完善，Linux 5.1 开始引入了全新的异步 I/O 接口：io_uring —— 希望 io_uring 能彻底解决 Linux 下的异步 I/O 问题。

参考资料

1. Linux Programmer's Manual - open (2)
2. Page-based direct I/O
3. InnoDB innodb_flush_method 配置
4. Why does O_DIRECT require I/O to be 512-byte aligned?
5. linux-aio15.14 InnoDB Startup Options and System Variableslinux-aio15.14 InnoDB Startup Options and System Variableslinux-aio15.14 InnoDB Startup Options and System Variableslinux-aio