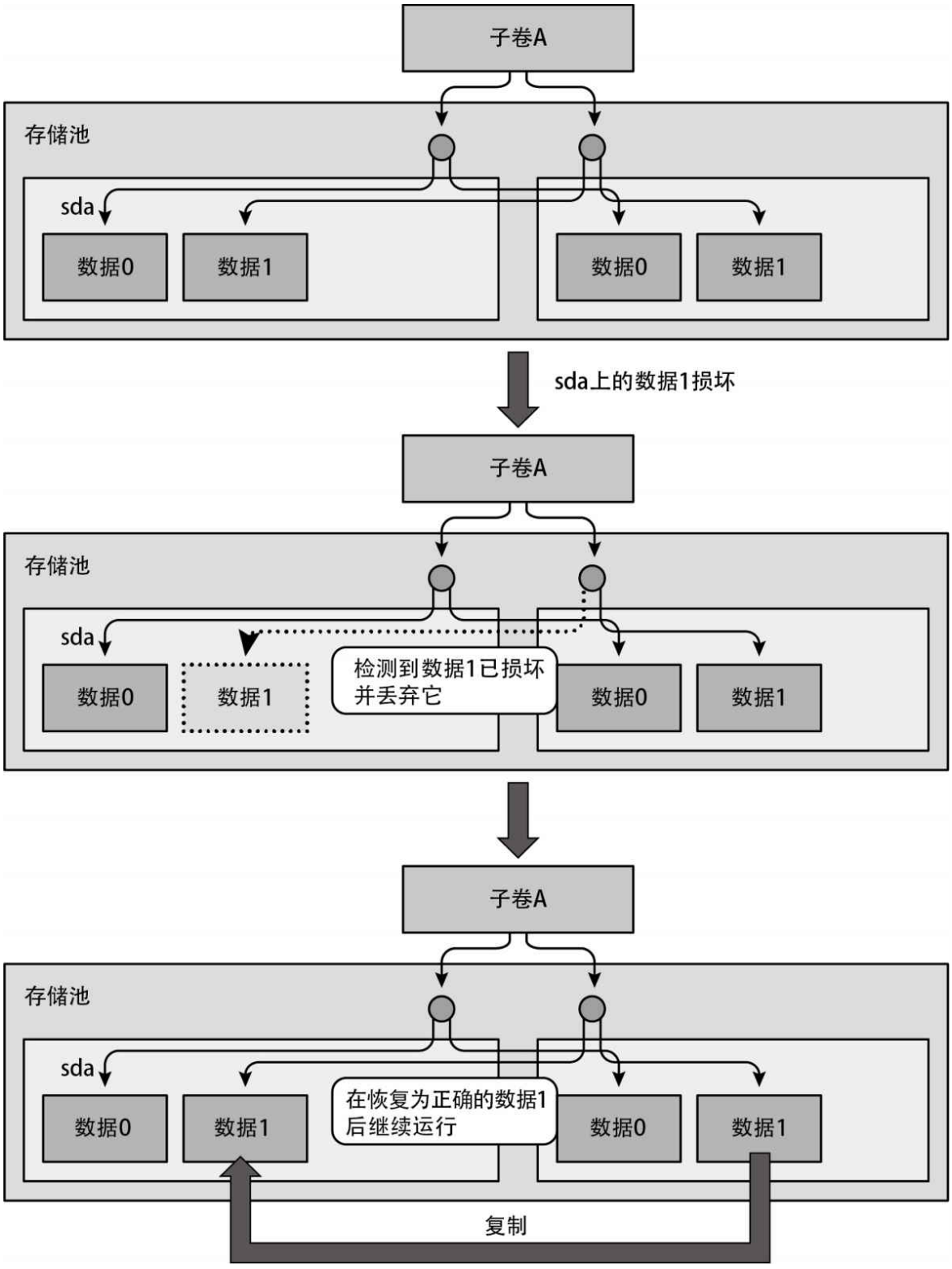

图 7-30　配置 RAID 1 时的数据修复流程

这种方式可以在申请读取的一方未发现数据损坏的情况下完成数据修复。

在写作本书时，虽然 ext4 与 XFS 可以通过为元数据附加校验和来检测并丢弃损坏的元数据，但是对数据的检测、丢弃与修复功能则只有 Btrfs 提供。

第 8 章　外部存储器

本章将介绍外部存储器以及与其相关的内核功能。

首先将介绍外部存储器中比较具有代表性的 HDD 的特性。接着介绍内核中利用 HDD 的特性来提高 I/O 性能的通用块层的相关内容。最后介绍近几年逐渐普及的、大有取代 HDD 的势头的 SSD。

8.1　HDD 的数据读写机制

HDD 用磁性信息表示数据，并将这些磁性数据记录在被称为**盘片**（platter）的磁盘上。HDD 读写数据的单位是**扇区**[1]，而非字节。在 HDD 的盘片上，沿半径方向与圆周方向划分出了多个扇区，并为每个扇区分配了序列号，如图 8-1 所示[2]。

[1]扇区的大小为 512 B 或者 4 KB。

[2]实际上，外侧的每一圈的扇区数量比内侧多。

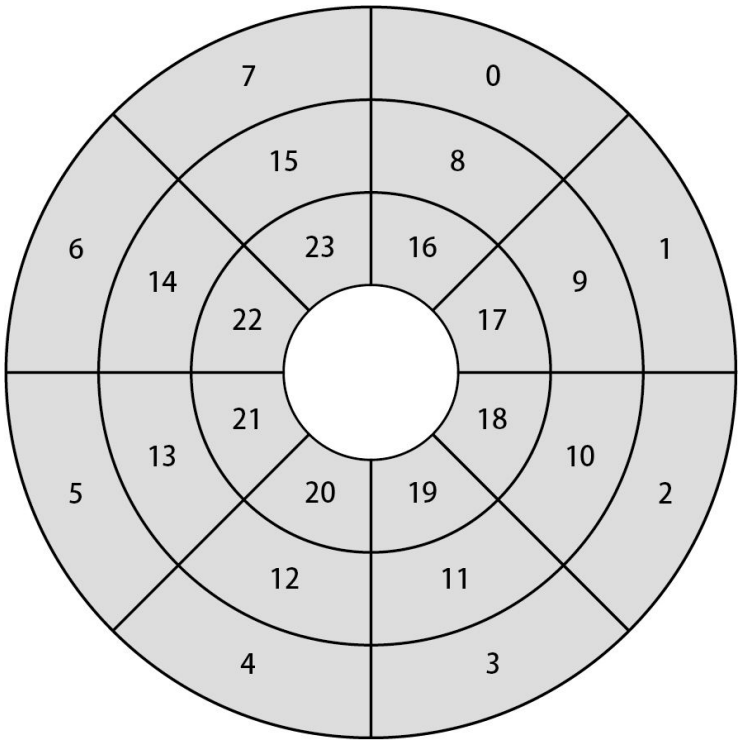

图 8-1　HDD 的扇区

HDD 通过名为**磁头**的部件读写盘片上各个扇区的数据。磁头安装在名为**磁头摆臂**的部件上，该部件令磁头可以在盘片上沿直径方向移动。在此基础上，通过转动盘片，即可把磁头定位到想要读取的任意一个扇区的上方。HDD 上的数据传输流程如下所示。

① 设备驱动程序将读写数据所需的信息传递给 HDD，其中包含扇区序列号、扇区数量以及访问类型（读取或写入）等信息。

② 通过摆动磁头摆臂并转动盘片，将磁头对准需要访问的扇区。

③ 执行数据读写操作。

④ 在执行读取的情况下，执行完 HDD 的读取处理就能结束数据传输。

在 HDD 中执行这一系列处理时的情形如图 8-2 所示。

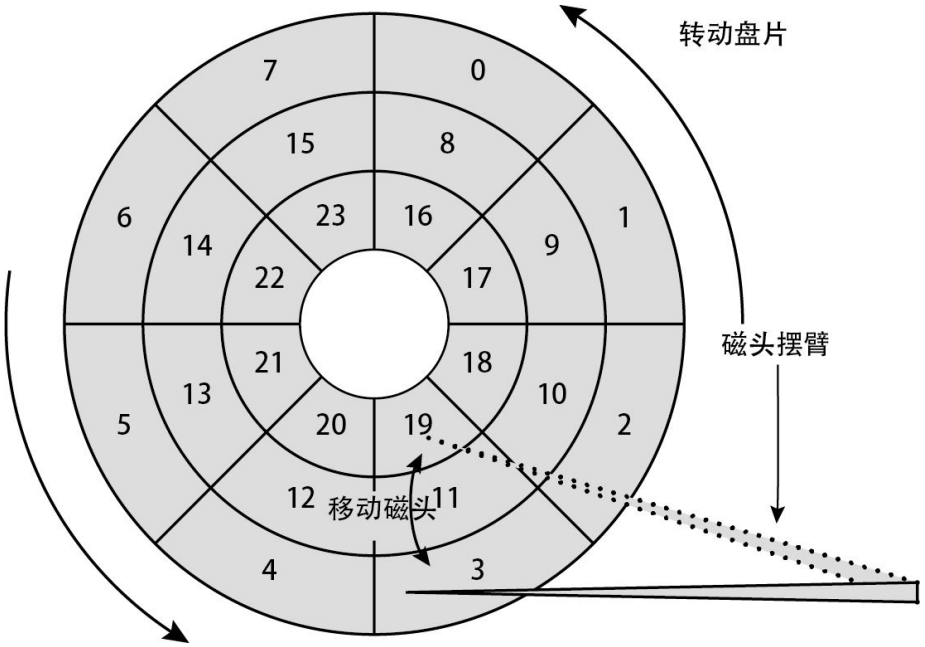

图 8-2　HDD 的数据传输流程

在前面的访问流程中，步骤①与步骤④是速度非常快的电子处理，而磁头摆臂的移动与盘片的转动则是速度慢得多的机械处理。因此，访问 HDD 时的延迟将受制于硬件的处理速度。这一延迟与只需电子处理的内存访问的延迟相比存在非常大的差距（详见 8.11 节）。访问过程中的各个步骤所消耗的时间大致上如图 8-3 所示。

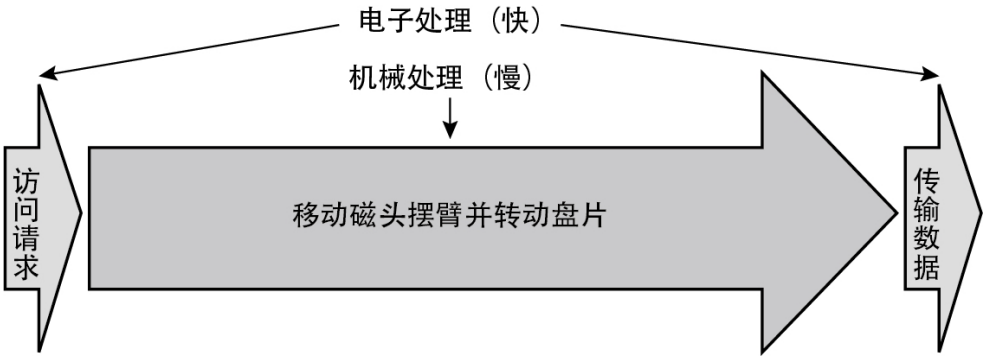

图 8-3　访问 HDD 所需要的时间

可以看到，大部分延迟来自于机械处理。

8.2 HDD 的性能特性

HDD 能在一次访问请求中读取多个连续扇区上的数据。因为磁头通过磁头摆臂在径向上对准位置后，只需转动盘片，就能读取多个连续扇区上的数据。一次读取的数据量取决于各个 HDD 自身。

在一次性读取扇区 0 到扇区 2 的数据时，磁头的移动轨迹如图 8-4 所示。

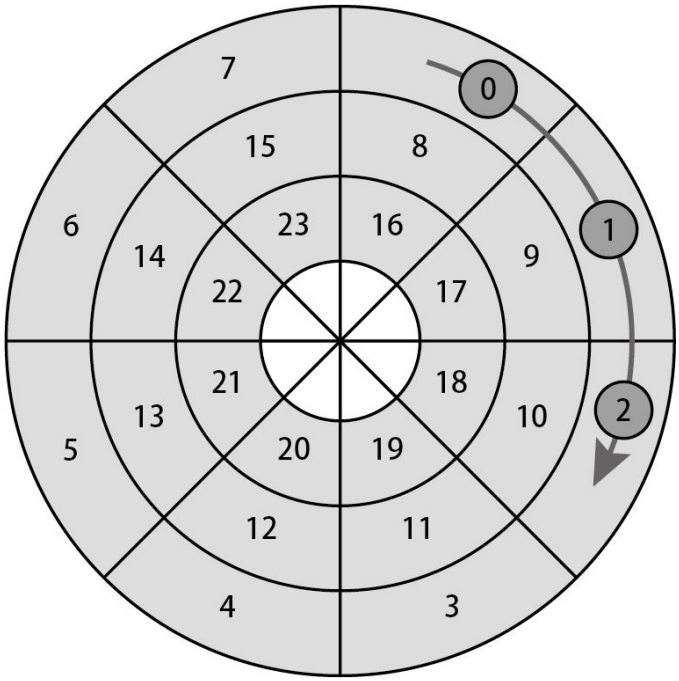

图 8-4 一次性读取扇区 0 到扇区 2 的数据时的情形

如果需要访问的扇区是连续的，却要分成多次来访问，就会增加访问处理的时间开销，如图 8-5 所示。

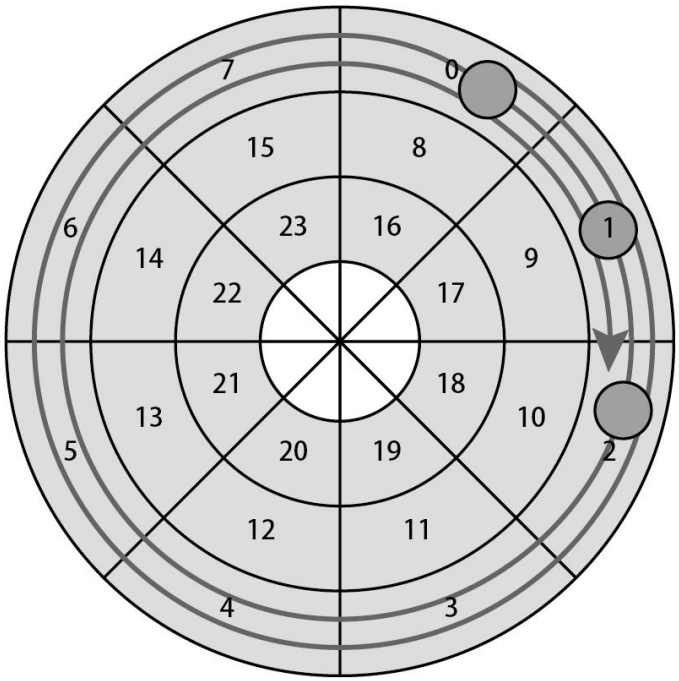

图 8-5　分成多次读取连续的扇区

与访问连续的扇区时的情形不同，在访问扇区 0、11、23 这种不连续的扇区时，则需要将访问请求分成多次发送给 HDD，这时访问轨迹会变长，如图 8-6 所示。

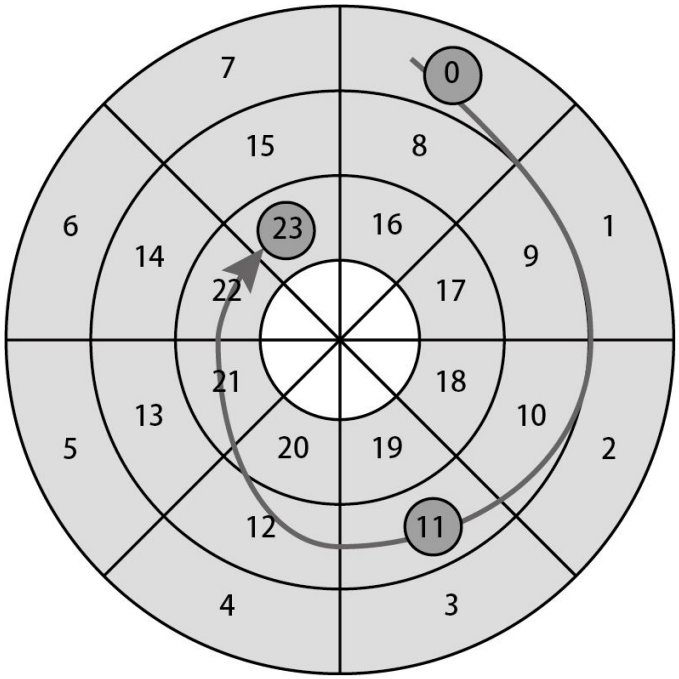

图 8-6　访问不连续的扇区

把这些处理各自消耗的时间排列到时间轴上，则如图 8-7 所示。

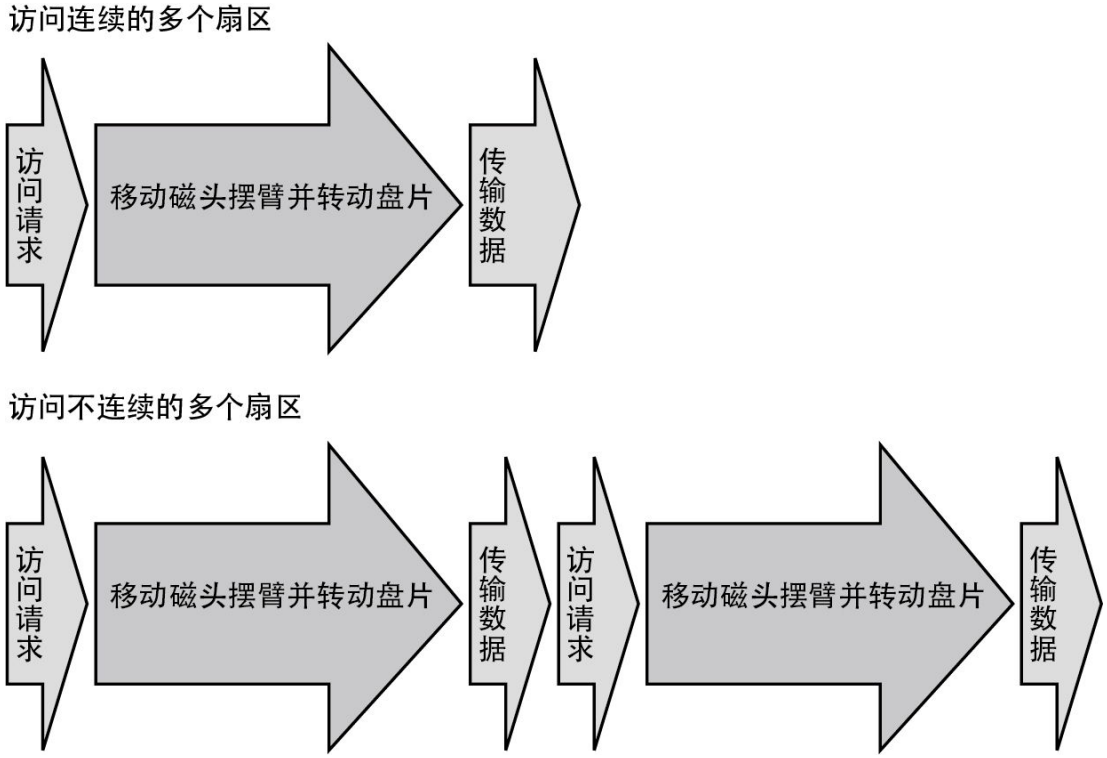

图 8-7　访问 HDD 的数据所消耗的时间（连续的扇区和不连续的扇区）

由于 HDD 具有这样的特性，所以各种文件系统都会尽量把文件中的数据存放在连续的区域上。大家在编写程序时也一样，尽量设计成下面这样比较好。

- 尽量将文件中的数据存放在连续的或者相近的区域上
- 把针对连续区域的访问请求汇集到一次访问请求中
- 对于文件，尽量以顺序访问的方式访问尽可能大的数据量

8.3 HDD 的实验

下面，让我们通过实验来确认一下 HDD 的性能是否如前面描述的那样。

为了能在实验中采集到原生数据，我们将直接读写块设备的数据。大家在各自的计算机上进行本实验时，请务必为测试准备一个未曾使用的分区，否则大家宝贵的数据就可能遭到损坏。

8.4　实验程序

本实验将测试下列内容。

- I/O 请求量的改变会引起什么样的性能变化？
- 顺序读取与随机读取有什么区别？

为此，需要编写一个实现下述要求的程序。

- 对指定分区中开头的 1 GB 的区域，发出总共 64 MB 的 I/O 请求
- 允许设定是执行读取还是写入，使用顺序访问还是随机访问，以及一次 I/O 请求的请求量
- 程序接收以下几个参数
  - 第 1 个参数：文件名
  - 第 2 个参数：是否启用内核的 I/O 支援功能（后述）
  - 第 3 个参数：访问的种类（**r =** 读取、**w =** 写入）
  - 第 4 个参数：访问的方式（**seq =** 顺序访问、**rand =** 随机访问）
  - 第 5 个参数：一次 I/O 请求的请求量（单位：KB）

完成后的程序如代码清单 8-1 所示。

代码清单 8-1　io 程序（io.c）

```
#define GNUSOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <err.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <linux/fs.h>

#define PART_SIZE    (1024*1024*1024)
#define ACCESS_SIZE  (64*1024*1024)
```

```
static char progname;

int main(int argc, char argv[])
{
    progname = argv[0];
    if (argc != 6) {
        fprintf(stderr, "usage: %s <filename> <kernel's help>
                <r/w> <access pattern>
                <block size[KB]>\n", progname);
        exit(EXIT_FAILURE);
    }

    char filename = argv[1];

    bool help;
    if (!strcmp(argv[2], "on")) {
        help = true;
    } else if (!strcmp(argv[2], "off")) {
        help = false;
    } else {
        fprintf(stderr, "kernel's help should be 'on' or
                'off': %s
", argv[2]);
        exit(EXIT_FAILURE);
    }

    int write_flag;
    if (!strcmp(argv[3], "r")) {
        write_flag = false;
    } else if (!strcmp(argv[3], "w")) {
        write_flag = true;
    } else {
        fprintf(stderr, "r/w should be 'r' or 'w': %s
",
                argv[3]);
        exit(EXIT_FAILURE);
    }

    bool random;
    if (!strcmp(argv[4], "seq")) {
        random = false;
    } else if (!strcmp(argv[4], "rand")) {
        random = true;
    } else {
        fprintf(stderr, "access pattern should be 'seq' or
                'rand': %s
", argv[4]);
        exit(EXIT_FAILURE);
    }
```

```
int part_size = PART_SIZE;
int access_size = ACCESS_SIZE;

int block_size = atoi(argv[5])  1024;
if (block_size == 0) {
    fprintf(stderr, "block size should be > 0: %s\n",
            argv[5]);
    exit(EXIT_FAILURE);
}
if (access_size % block_size != 0) {
    fprintf(stderr, "access size(%d) should be multiple of
            block size: %s\n", access_size, argv[5]);
    exit(EXIT_FAILURE);
}
int maxcount = part_size  block_size;
int count = access_size  block_size;

int offset = malloc(maxcount  sizeof(int));
if (offset == NULL)
    err(EXIT_FAILURE, "malloc() failed");

int flag = O_RDWR | O_EXCL;
if (!help)
    flag |= O_DIRECT;

int fd;
fd = open(filename, flag);
if (fd == -1)
    err(EXIT_FAILURE, "open() failed");

int i;
for (i = 0; i < maxcount; i++) {
    offset[i] = i;
}
if (random) {
    for (i = 0; i < maxcount; i++) {
        int j = rand() % maxcount;
        int tmp = offset[i];
        offset[i] = offset[j];
        offset[j] = tmp;
    }
}

int sector_size;
if (ioctl(fd, BLKSSZGET, &sector_size) == -1)
    err(EXIT_FAILURE, "ioctl() failed");

char buf;
int e;
e = posix_memalign((void *)&buf, sector_size, block_size);
if (e) {
```

```
        errno = e;
        err(EXIT_FAILURE, "posix_memalign() failed");
    }

    for (i = 0; i < count; i++) {
        ssize_t ret;
        if (lseek(fd, offset[i] * block_size, SEEK_SET) == -1)
            err(EXIT_FAILURE, "lseek() failed");
        if (write_flag) {
            ret = write(fd, buf, block_size);
            if (ret == -1)
                err(EXIT_FAILURE, "write() failed");
        } else {
            ret = read(fd, buf, block_size);
            if (ret == -1)
                err(EXIT_FAILURE, "read() failed");
        }
    }
    if (fdatasync(fd) == -1)
        err(EXIT_FAILURE, "fdatasync() failed");

    if (close(fd) == -1)
        err(EXIT_FAILURE, "close() failed");

    exit(EXIT_SUCCESS);
}
```

在阅读源代码时，需要注意以下几点。

- 为 **open()** 函数添加 **O_DIRECT** 标记，启用直接 I/O（Direct I/O）。在启用直接 I/O 后，可以禁用内核的 I/O 支援功能
- 通过 **ioctl()** 函数获取与指定设备对应的外部存储器上的扇区大小
- 源代码中的 **buf** 变量是用于保存外部存储器传输过来的数据的缓冲区，该缓冲区使用与 **malloc()** 函数类似的 **posix_memalign()** 函数来申请内存区域。在通过该函数获取内存区域时，该区域的起始地址是函数参数中指定的数值的倍数（对齐）。这是因为在使用直接 I/O 时，缓冲区在内存中的起始地址与大小必须为外部存储器上的扇区大小的倍数
- 通过 **fdatasync()** 函数等待上一层中的 I/O 请求完成处理。因为在标准 I/O（而不是直接 I/O）中，如果仅用 **write()** 函数发出 I/O 请求，并不会等待 I/O 请求完成处理

编译方法如下。

```
$ cc -o io io.c
$
```