操作系统

Operating System

极夜酱

目录

Chapter 1 设备管理

1.1 I/O 重定向

1.1.1 标准 I/O

程序对读入的数据进行处理，再输出数据。数据的输入（input）和输出（output）简称为 I/O，在没有指定输入输出的情况下，默认为标准输入和标准输出。

打开的文件都有一个文件描述符（fd, file descriptor），表现为一个数字：

- 标准输入 stdin（键盘）：fd = 0
- 标准输出 stdout（显示器）：fd = 1
- 标准错误输出 stderr（显示器）：fd = 2

标准 I/O

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int num;
    printf("(stdin) enter an integer: ");
    fscanf(stdin, "%d", &num);
    fprintf(stdout, "(stdout) num = %d\n", num);
    fprintf(stderr, "(stderr) This is an error message.\n");
    return 0;
}
```

运行结果

```
(stdin) enter an integer: 123
(stdout) num = 123
(stderr) This is an error message.
```

文件 I/O

```c
#include <stdio.h>
#include <stdlib.h>

void writeFile(const char *filename) {
    FILE *fp = fopen(filename, "w");
    if(!fp) {
        fprintf(stderr, "File open failed.\n");
        exit(1);
    }
    char *name = "小灰";
    int age = 17;
    double height = 182.3;
    fprintf(fp, "姓名：%s\n年龄：%d\n身高：%.2f\n",
                name, age, height);
    fclose(fp);
}

void readFile(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if(!fp) {
        fprintf(stderr, "File open failed.\n");
        exit(1);
    }
    char name[32];
    int age;
    double height;
    fscanf(fp, "姓名：%s\n年龄：%d\n身高：%lf\n",
```

```
28                  name, &age, &height);
29      printf("name: %s\n", name);
30      printf("age: %d\n", age);
31      printf("height: %.2f\n", height);
32      fclose(fp);
33  }
34
35  int main(int argc, char *argv[]) {
36      const char *filename = "info.txt";
37      writeFile(filename);
38      readFile(filename);
39      return 0;
40  }
```

运行结果

```
name: 小灰
age: 17
height: 182.30
```

1.1.2 I/O 重定向（I/O Redirection）

I/O 重定向就是改变标准输入与输出的默认位置。标准输入默认是键盘，通过改成其它输入，就是输入重定向，例如从文本文件里输入。标准输出默认是显示器，通过改成其它输出，就是输出重定向，例如输出到文件。

输出重定向用【>】表示，若文件不存在，则创建；若文件已存在，则覆盖。使用【>>】时若文件不存在，则创建，若文件已存在，则追加。错误输出重定向用【2>】和【2>>】表示。

输入重定向用【<】表示，但是在输入重定向中【<<】可不是表示输入追加。

I/O 重定向

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int data;
    scanf("%d", &data);
    printf("data = %d\n", data);
    return 0;
}
```

input.txt

```
12345
```

编译

```
gcc -Wall io_redirection.c -o io_redirection
```

运行

```
./io_redirection < input.txt > output.txt
```

运行结果 output.txt

```
num = 12345
```

1.2 I/O 控制方式

1.2.1 I/O 设备

现代计算机系统总是配有各种类型的外部设备，除了显示器、键盘、打印机、磁带、磁盘外，又出现了光盘、绘图仪、图形数字化仪、鼠标器、激光打印机、声音输入输出设备等，种类繁多。不同外设之间的差异较大，因此 I/O 性能经常成为系统的瓶颈。

操作系统设备管理的目标包括：

1. 向用户提供使用外部设备的方便、统一的接口，按照用户的要求和设备的类型，控制设备工作，完成用户的输入输出请求。方便是指用户能独立于具体设备的复杂物理特性而方便地使用设备；统一是指对不同设备尽量能统一操作方式。方便和统一要求对用户屏蔽实现具体设备 I/O 操作的细节，呈现给用户的是一种性能理想化的、操作简便的逻辑设备。系统的这种性能亦称为设备的独立性（设备无关性）。
2. 充分利用中断技术、通道技术和缓冲技术，提高 CPU 与设备、设备与设备间的并行工作能力，充分利用设备资源，提高外部设备的使用效率。
3. 设备管理就是要保证在多道程序环境下，当多个进程竞争使用设备时，按照一定的策略分配和管理设备，以使系统能有条不紊地工作。

1.2.2 I/O 控制方式

早期，计算机设计者没有将 CPU 的执行与 I/O 操作分开，甚至大多数人认为输入输出与计算的时间应该是同一数量级。后来，他们意识到，CPU 要比 I/O 操作速度高几个数量级。于是，硬件和软件设计师开始寻找一种技术，使 CPU 计算可以不用等待 I/O 操作而持续执行。

I/O 控制方式包括：

1. 程序控制 I/O（Programmed I/O）：处理器代表一个进程给 I/O 模块发送一个 I/O 命令，该进程进入忙等待（busy waiting），直到操作完成。

2. 中断驱动 I/O（Interrupt I/O）：处理器代表进程向 I/O 发送命令，如果来自进程的 I/O 指令是非阻塞的，那么处理器继续执行进程的后续指令。如果 I/O 指令时阻塞的，那么处理器执行来自操作系统的指令，将当前进程设置为阻塞态并且调度其它进程。

3. 直接存储器访问（DMA）：DMA 模块控制内存和 I/O 模块之间的数据交换。为传送一块数据，处理器给 DMA 模块发送请求，只有当整个数据块传送结束后，它才能被中断。

1.3 I/O 缓冲

1.3.1 I/O 缓冲（I/O Buffering）

在设备管理中，为了缓和 CPU 与 I/O 设备速度不匹配的矛盾，提高 CPU 与 I/O 设备的并行性、减少对 CPU 的中断频率，在 I/O 设备与处理机交换数据时都用到了缓冲区。

缓冲的种类包括：

单缓冲

输入时通道先将数据送入缓冲区，CPU 从缓冲区取数据处理。通道再送入后续数据，如此反复直到输入完成。输出情形正好相反。由于缓冲区属于互斥区，所以单缓冲并不能明显改善 CPU 与外部设备的并行性。

图 1.1: 单缓冲

双缓冲

分别设置输入缓冲区和输出缓冲区，CPU 和通道可以分别访问两个缓冲区，即在 CPU 访问一个缓冲区的同时，通道可以访问另一个缓冲区。双缓冲只是一种说明设备和设备、CPU 和设备并行操作的简单模型，并不能用于实际系统中的操作。因为计算机系统中的外围设备较多，另外双缓冲也很难匹配设备和处理机的处理速度。现代计算机系统一般使用多缓冲或缓冲池结构。

多缓冲

把多个缓冲区连接起来组成两部分，一部分专门用于输入，另一部分专门用于输出的缓冲结构。常组织成循环队列的结构，也称为循环缓冲。

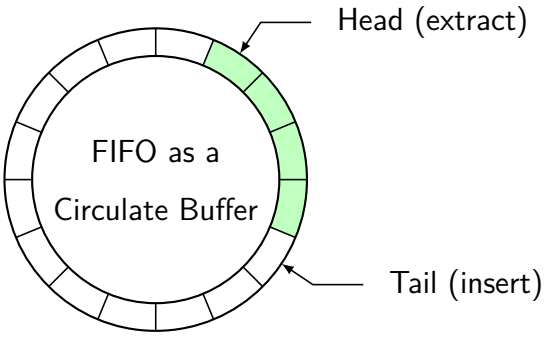


图 1.2: 多缓冲

缓冲池

把多个缓冲区连接起来统一管理，既可用于输入又可用于输出的缓冲结构。

1.4 磁盘调度

1.4.1 磁盘调度算法

多道程序系统中，各进程可能会不断提出不同对磁盘进行读写操作的请求。由于有时这些进程发送请求的速度比磁盘响应还要快，因此有必要为每个磁盘设备建立一个等待队列。磁盘调度算法的目的是为了提高磁盘的访问性能，一般通过优化磁盘的访问请求顺序实现。

图 1.3: 磁盘

假设有下面一个请求序列，每个数字代表磁道的位置：98, 183, 37, 122, 14, 124, 65, 67。初始磁头当前的位置是在第 53 磁道。

1.4.2 先来先服务（FCFS, First Come First Served）

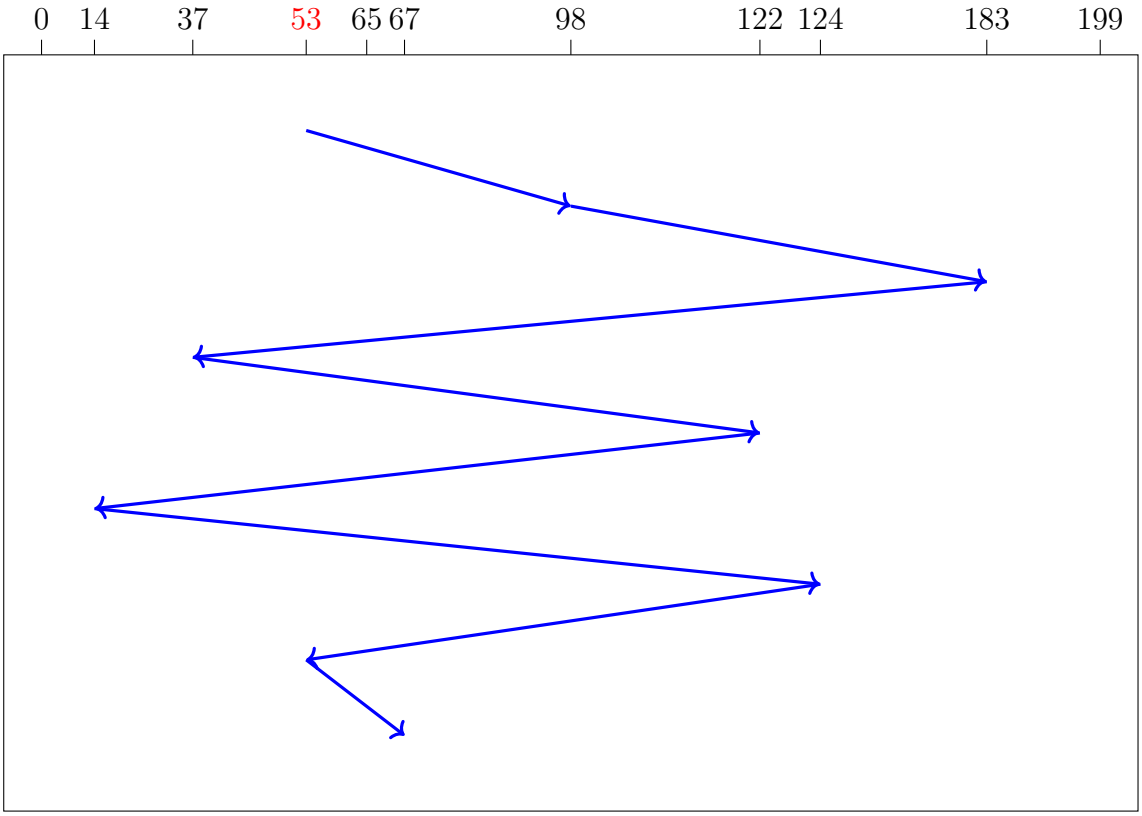


图 1.4: 先来先服务 FCFS

FCFS 算法比较简单粗暴，但是如果大量进程竞争使用磁盘，请求访问的磁道可能会很分散，那 FCFS 算法因为寻道时间过长，在性能上就会显得很差。

1.4.3 最短寻道时间优先（SSTF, Shortest Seek Time First）

SSTF 算法优先选择从当前磁头位置所需寻道时间最短的请求。

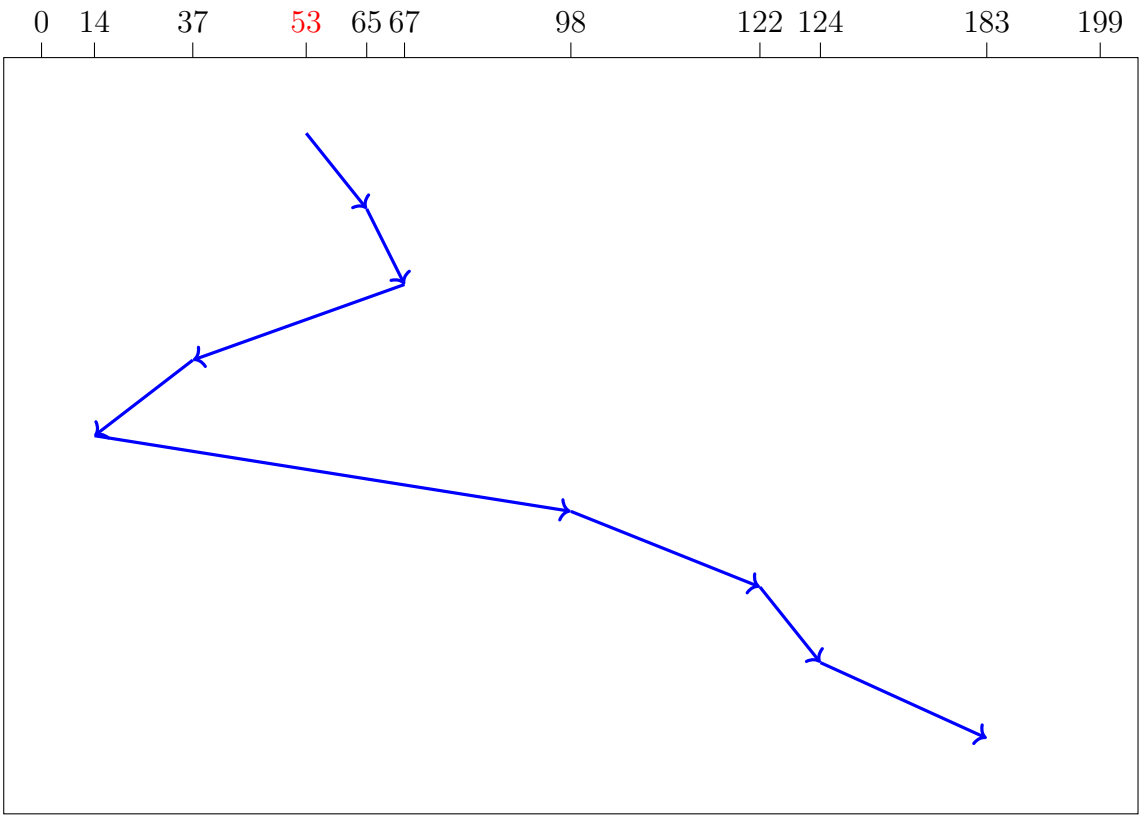


图 1.5: 最短寻道时间优先 SSTF

但 SSTF 算法可能存在某些请求的饥饿，对用户的服务请求的响应机会不是均等的。磁头在一小块区域来回移动，因而导致响应时间的变化幅度很大，有些请求的响应时间将不可预期。

1.4.4 扫描算法（SCAN）

为了防止 SSTF 算法产生饥饿的问题，可以规定磁头在一个方向上移动，访问所有未完成的请求，直到磁头到达该方向上的最后的磁道，才调换方向。这种算法也叫做电梯调度算法，比如电梯保持按一个方向移动，直到在那个方向上没有请求为止，然后改变方向。

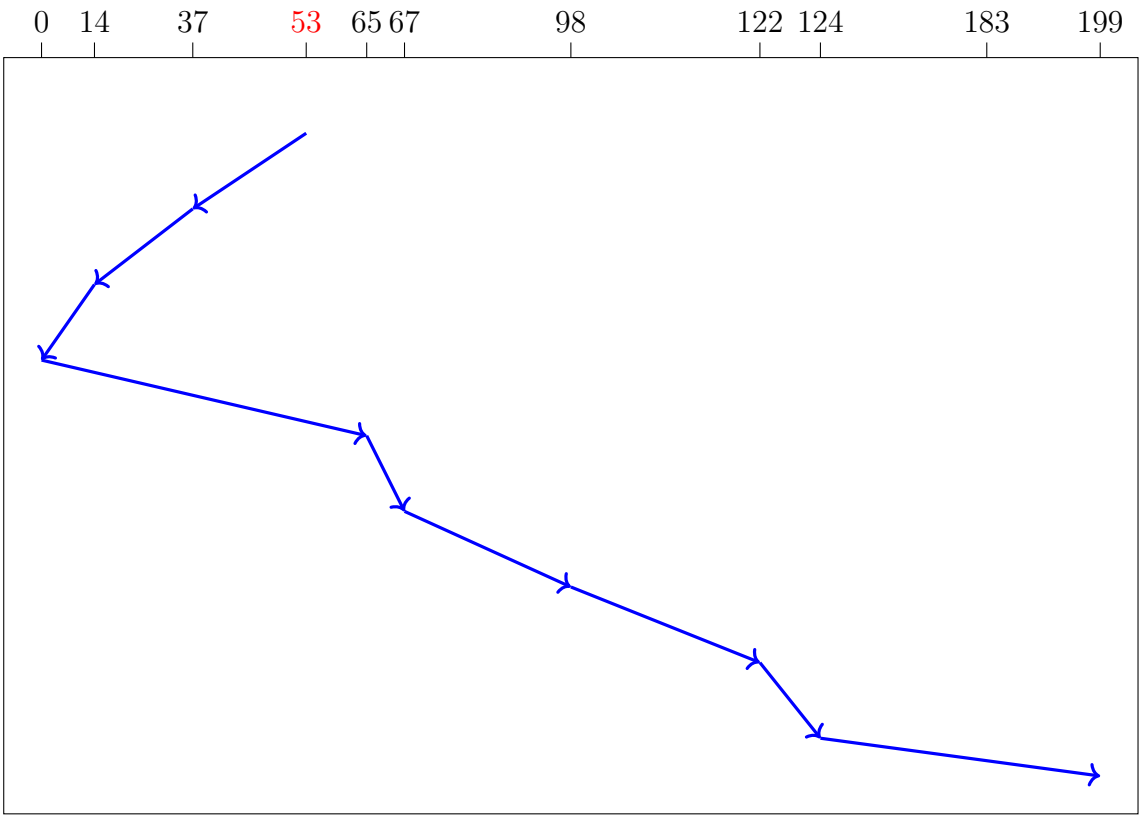


图 1.6: 扫描算法 SCAN

SCAN 算法性能较好，不会产生饥饿现象，但是存在这样的问题：中间部分的磁道会比较占便宜，中间部分相比其它部分响应的频率会比较多，也就是说每个磁道的响应频率存在差异。

1.4.5 循环扫描算法（CSCAN, Circular Scan）

CSCAN 算法规定只有磁头朝某个特定方向移动时，才处理磁道访问请求，而返回时直接快速移动至最靠边缘的磁道，也就是复位磁头，这个过程是很快的，并且返回中途不处理任何请求。

CSCAN 算法的特点就是磁道只响应一个方向上的请求，相比于 SCAN 算法，对于各个位置磁道响应频率相对比较平均。

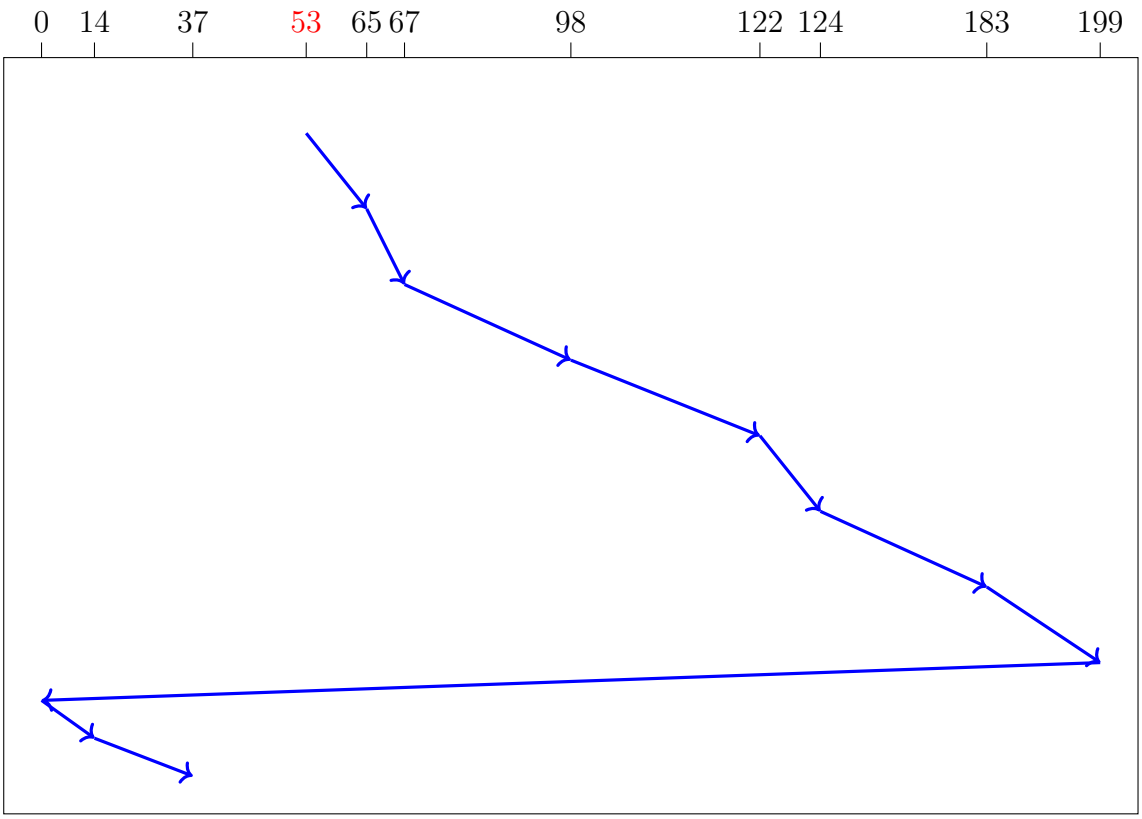


图 1.7: 循环扫描算法 CSCAN

1.5 RAID

1.5.1 RAID（Redundant Array of Independent Disks）

在单机时代，采用单块磁盘进行数据存储和读写的方式，由于寻址和读写的时间消耗，导致 I/O 性能非常低，且存储容量还会受到限制。另外，单块磁盘极其容易出现物理故障，经常导致数据的丢失。

1988 年美国加州大学伯克利分校的 D. A. Patterson 教授等首次在论文中提出了 RAID 概念，即廉价冗余磁盘阵列（Redundant Array of Inexpensive Disks）。由于当时大容量磁盘比较昂贵，RAID 的基本思想是将多个容量较小、相对廉价的磁盘进行有机组合，从而以较低的成本获得与昂贵大容量磁盘相当的容量、性能、可靠性。随着磁盘成本和价格的不断降低，廉价已经毫无意义。因此决定用 Independent 替代 Inexpensive。于时 RAID 变成了独立磁盘冗余阵列（Redundant Array of Independent Disks），但这仅仅是名称的变化，实质内容没有改变。

RAID 思想从提出后就广泛被业界所接纳，存储工业界投入了大量的时间和财力来研究和开发相关产品。RAID 主要优势包括：

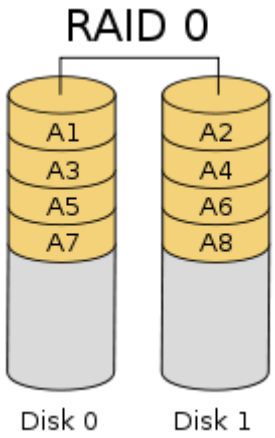
1. 大容量：扩大了磁盘的容量，由多个磁盘组成的 RAID 系统具有海量的存储空间。现在单个磁盘的容量就可以到 1TB 以上，这样 RAID 的存储容量就可以达到 PB 级，大多数的存储需求都可以满足。
2. 高性能：单个磁盘的 I/O 性能受到接口、带宽等技术的限制，性能往往很有限，容易成为系统性能的瓶颈。通过数据条带化，RAID 将数据 I/O 分散到各个成员磁盘上，从而获得比单个磁盘成倍增长的聚合 I/O 性能。
3. 可靠性：RAID 冗余技术大幅提升数据可用性和可靠性，保证了若干磁盘出错时，不会导致数据的丢失，不影响系统的连续运行。

1.5.2 RAID0

RAID0 是一种简单的、无数据校验的数据条带化技术，将所在磁盘条带化后组成大容量的存储空间，将数据分散存储在所有磁盘中，以独立访问方式实现多块

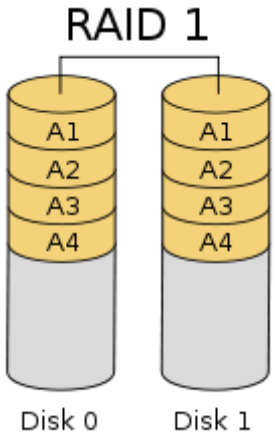
磁盘的并读访问。

RAID0 具有低成本、高读写性能，但是它不提供数据冗余保护，一旦数据损坏，将无法恢复。因此，RAID0 一般适用于对性能要求严格但对数据安全性和可靠性不高的应用，如视频、音频存储、临时数据缓存空间等。



1.5.3 RAID1

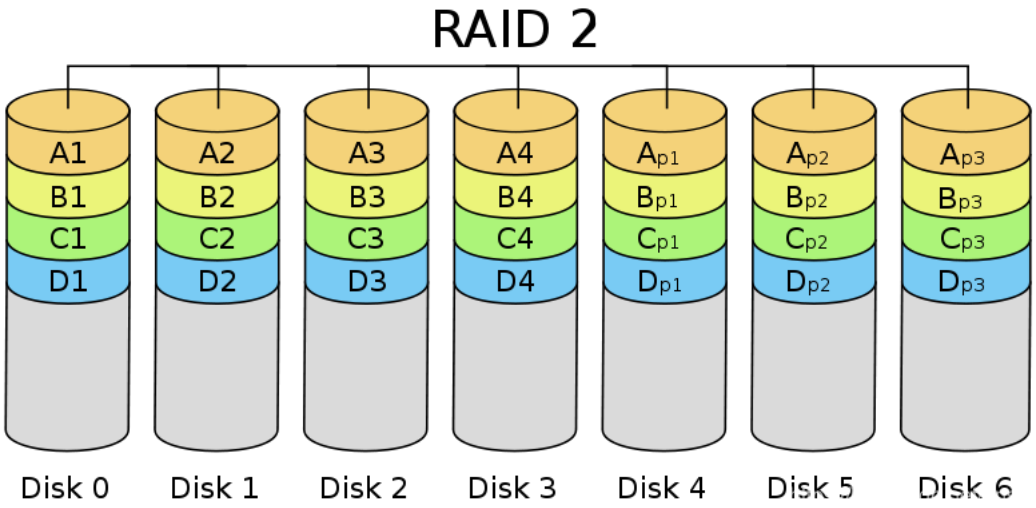
RAID1 称为镜像，它将数据完全一致地分别写到工作磁盘和镜像磁盘，它的磁盘空间利用率为 50%。RAID1 在数据写入时，响应时间会有所影响，但是读数据时没有影响。RAID1 提供了最佳的数据保护，一旦工作磁盘发生故障，系统自动从镜像磁盘读取数据。RAID1 拥有完全容错的能力，但实现成本高。



1.5.4 RAID2

RAID2 称为纠错海明码磁盘阵列，其设计思想是利用海明码实现数据校验冗余。海明码是一种在原始数据中加入若干校验码来进行错误检测和纠正的编码技术。海明码自身具备纠错能力，因此 RAID2 可以在数据发生错误的情况下对纠正错误，保证数据的安全性。

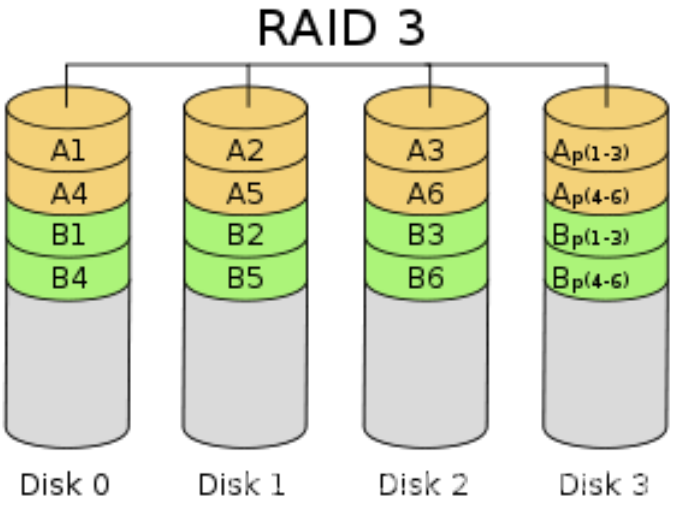
但是海明码的数据冗余开销太大，而且 RAID2 的数据输出性能受阵列中最慢磁盘驱动器的限制。并且海明码是按位运算，RAID2 数据重建非常耗时。因此 RAID2 在实际中很少应用，没有形成商业产品。



1.5.5 RAID3

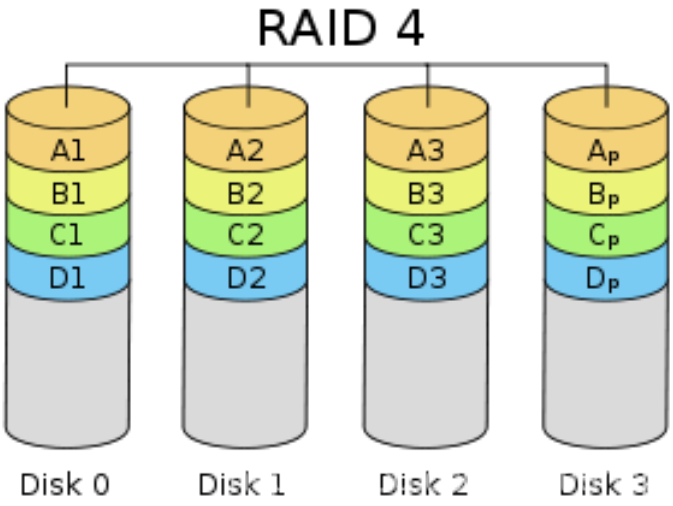
RAID3 是使用专用校验盘的并行访问阵列，它采用一个专用的磁盘作为校验盘，其余磁盘作为数据盘。不同磁盘上同一带区的数据作 XOR 校验，校验值写入校验盘中。RAID3 完好时读性能与 RAID0 完全一致，并行从多个磁盘条带读取数据，性能非常高，同时还提供了数据容错能力。如果 RAID3 中某一磁盘出现故障，不会影响数据读取，可以借助校验数据和其他完好数据来重建数据。

RAID3 只需要一个校验盘，阵列的存储空间利用率高，再加上并行访问的特征，能够为高带宽的大量读写提供高性能，适用大容量数据的顺序访问应用，如影像处理、流媒体服务等。



1.5.6 RAID4

RAID4 与 RAID3 的原理大致相同，区别在于条带化的方式不同。RAID4 按照块的方式来组织数据，保证单块的完整性，可以避免受到其他磁盘上同条带产生的不利影响。



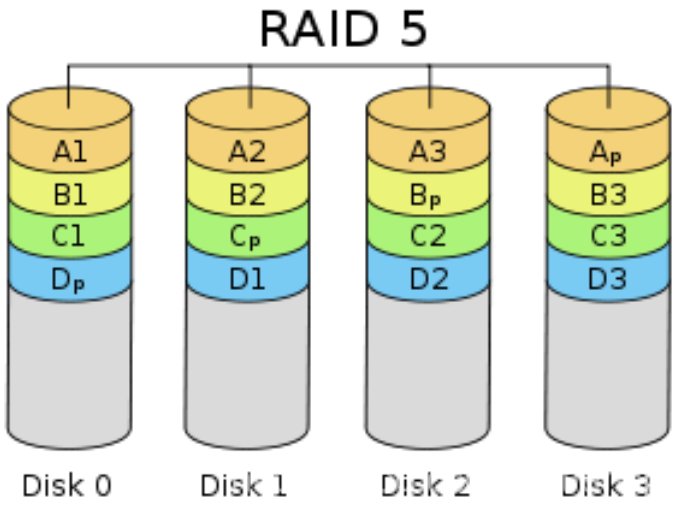
1.5.7 RAID5

RAID5 应该是目前最常见的 RAID 等级，它的原理与 RAID4 相似，区别在于校验数据分布在阵列中的所有磁盘上，而没有采用专门的校验磁盘。因此，RAID5

不存在 RAID4 中的并发写操作时的校验盘性能瓶颈问题。

RAID5 的磁盘上同时存储数据和校验数据，数据块和对应的校验信息存保存在不同的磁盘上，当一个数据盘损坏时，系统可以根据同一条带的其他数据块和对应的校验数据来重建损坏的数据。

RAID5 兼顾存储性能、数据安全和存储成本等各方面因素，是目前综合性能最佳的数据保护解决方案。RAID5 基本上可以满足大部分的存储应用需求，数据中心大多采用它作为应用数据的保护方案。



Chapter 2 文件系统

2.1 文件操作

2.1.1 打开文件

计算机除了拥有计算的能力之外，一定要有数据的存储能力，而对于数据的存储一般就可以通过文件的形式来完成。在 Python 中直接提供有文件的 I/O（Input/Output）处理函数操作，利用这些函数可以方便地实现读取和写入。

open() 函数的功能是进行文件的打开，在进行文件打开的时候如果不设置任何的模式类型，则默认为 r（只读模式）。

```
def open(
    file, mode='r', buffering=None, encoding=None,
    errors=None, newline=None, closefd=True
)
```

模式	描述
r	使用只读模式打开文件，此为默认模式
w	写模式，如果文件存在则覆盖，文件不存在则创建
x	写模式，新建一个文件，如果该文件已存在则会报错
a	内容追加模式
b	二进制模式
t	文本模式（默认）
+	打开一个文件进行更新（可读可写）

表 2.1: 文件打开模式

如果以只读的模式打开文件，并且文件路径不存在的话，就会出现 FileNotFoundError 错误信息。

文件操作

```python
def main():
    file = open(file="data.txt", mode="w")
    print("文件名称: %s" % file.name)
    print("访问模式: %s" % file.mode)
    print("文件状态: %s" % file.closed)
    print("关闭文件...")
    file.close()
    print("文件状态: %s" % file.closed)

if __name__ == "__main__":
    main()
```

运行结果

```
文件名称: data.txt
访问模式: w
文件状态: False
关闭文件...
文件状态: True
```

2.1.2 文件读写

当使用 open() 打开一个文件之后，接下来可以使用创建的文件对象进行读写操作。

使用读模式打开文件后，可以使用循环读取每一行的数据内容。Python 在进行文件读取操作的时候也可以进一步简化操作。文件对象本身是可以迭代的，在迭代的时候是以换行符进行分割，每次迭代就读取到一行数据内容。

方法	描述
def close(self)	关闭文件资源
def fileno(self)	获取文件描述符
def flush(self)	强制刷新缓冲区
def read(self, n: int = -1)	数据读取，默认读取全部
def readlines(self, hint: int = -1)	读取所有数据行
def readline(self, limit: int = -1)	读取每行数据
def truncate(self, size: int = None)	文件截取
def writable(self)	判断文件是否可以写入
def write(self, s: AnyStr)	文件写入
def writelines(self, lines, List[AnyStr])	写入一组数据

表 2.2: 文件读写方法

既然所有的文件对象最终都需要被开发者关闭，那么可以结合 with 语句实现自动的关闭处理。通过 with 实现所有资源对象的连接和释放是在 Python 中编写资源操作的重要技术手段，通过这样的操作可以极大地减少和优化代码结构。

读取文件

```
1 def main():
2     with open(file="data.txt", mode="r", encoding="utf-8") as file:
3         for line in file:
4             print(line, end='')
5
6 if __name__ == "__main__":
7     main()
```

data.txt

```
1 小灰    16
2 小白    17
3 小黄    21
```

运行结果

```
小灰 16
小白 17
小黄 21
```

写入文件

```python
def main():
    with open(file="data.txt", mode="w", encoding="utf-8") as file:
        info = {"小灰": 16, "小白": 17, "小黄": 21}
        for name, age in info.items():
            file.write("%s\t%d\n" % (name, age))

if __name__ == "__main__":
    main()
```

运行结果 data.txt

```
小灰 16
小白 17
小黄 21
```

2.2 文件缓冲

2.2.1 文件缓冲

在使用 open() 创建一个文件对象的时候，默认情况下是不会启用缓冲的。在 open() 中提供的 buffering 参数描述的就是文件处理缓冲的定义，在进行文件写入的时候利用缓冲可以避免频繁的 I/O 资源占用。

开启缓冲可以提高写入效率，buffering 参数设置有 3 种类型：

1. 全缓冲（buffering > 1）：当标准 I/O 缓存被填满后才会进行真正 I/O 操作，全缓冲的典型代表就是对磁盘文件的读写操作。
2. 行缓冲（buffering = 1）：在 I/O 操作中遇见换行符时才执行真正的 I/O 操作，例如在使用网络聊天工具时所编辑的文字在没有发送前是不会进行 I/O 操作的。
3. 不缓冲（buffering = 0）：直接进行终端设备的 I/O 操作，数据不经过缓冲保存。

如果想要观察到行缓冲的使用特点，就不能直接使用 with 语句，因为 with 最后会执行 close() 的关闭操作，而一旦关闭，则缓冲的内容会全部进行输出。

使用 flush() 进行缓冲区的强制清空，一旦强制清空之后，缓冲区的内容将全部输出。每次使用 close() 关闭文件流的时候默认情况下也会调用 flush() 进行缓冲区的清空处理。

文件缓冲

```
1 import os
2
3 def main():
4     file = open(file="data.txt", mode="w",
5                 encoding="utf-8", buffering=1)
6     file.write("This is a test.")
```

```
7       os.system("pause")  # 程序暂停
8       file.flush()        # 强制清空缓冲区
9       os.system("pause")  # 程序暂停
10      file.close()
11
12  if __name__ == "__main__":
13      main()
```

运行结果 data.txt

```
This is a test.
```

2.3 文件系统

2.3.1 文件系统

文件系统是 OS 中负责管理持久数据的子系统，也就是负责把用户的文件存到磁盘硬件中，即使计算机断电了，磁盘里的数据并不会丢失，所以可以持久化地保存文件。文件系统的基本数据单位是文件，它的目的是对磁盘上的文件进行组织管理。

Linux 最经典的一句话是 “一切皆文件”，不仅普通的文件和目录，就连块设备、管道、socket 等，也都是统一交给文件系统管理的。

文件和目录是按照层次关系管理的，这种结构可以用树表示。文件夹中包含了子文件夹或者文件，这样就形成了一个树的结构。不过也有一些特殊情况，例如团队在协同工作时会共享一些信息，两个人有着不同的目录，但是可以共享同一份文件。

Linux 文件系统会为每个文件分配索引结点（inode）和目录项（directory entry）这两个数据结构。索引结点用来记录文件的元信息，比如 inode 编号、文件大小、访问权限、创建时间、修改时间、数据在磁盘的位置等。目录项用来记录文件的名字、索引结点指针以及与其它目录项的层级关联关系。多个目录项关联起来，就会形成目录结构，但它与索引结点不同的是，目录项是由内核维护的一个数据结构，不存放于磁盘，而是缓存在内存。

由于索引结点唯一标识一个文件，而目录项记录着文件名，所以目录项和索引结点的关系是多对一。也就是说，一个文件可以有多个别名，例如，硬链接的实现就是多个目录项中的索引结点指向同一个文件。

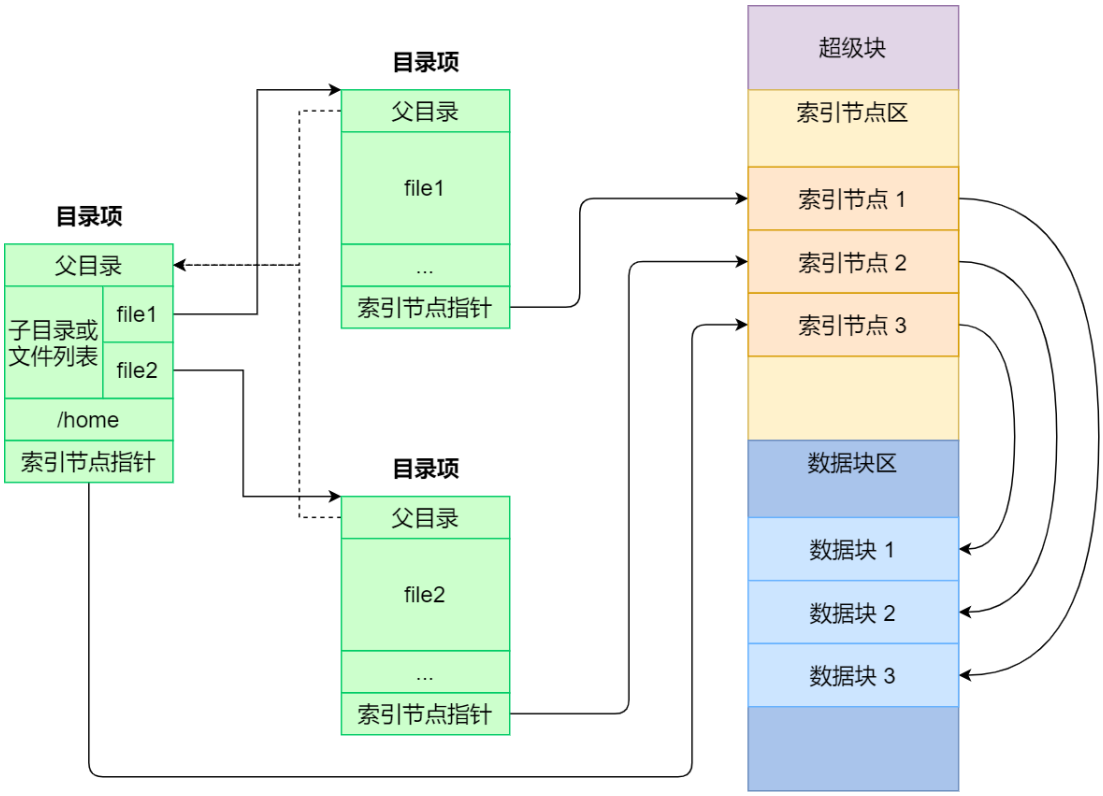


图 2.1: 文件系统

2.3.2 硬链接与软链接

链接简单说实际上是一种文件共享的方式，主流文件系统都支持链接文件。链接简单地理解为 Windows 中常见的快捷方式，Linux 中常用它来解决一些库版本的问题，通常也会将一些目录层次较深的文件链接到一个更易访问的目录中。

链接分为硬链接（hard link）和软链接（symbolic link）。从使用的角度讲，两者没有任何区别，都与正常的文件访问方式一样，支持读写，如果是可执行文件的话也可以直接执行。

硬链接通过同一个 inode 指向原始文件，软链接通过硬盘媒介中的描述指向原始文件。但是，当原始文件的位置发生改变后，inode 不会改变，所以硬链接还是正确的，而软链接就无法访问了。

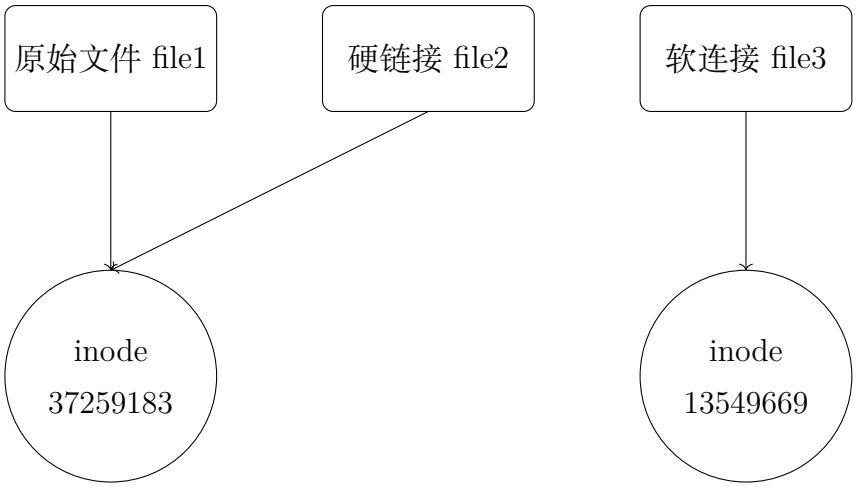

原始文件 file1
硬链接 file2
软连接 file3
inode
37259183
inode
13549669

图 2.2: 硬链接与软链接

2.4 文件存储

2.4.1 顺序分配

文件的数据是要存储在硬盘上面的，数据在磁盘上的存放方式，就像程序在内存中存放的方式那样，分为连续空间存放和非连续空间存放两种方式。不同的存储方式有各自的特点，它们的存储效率和读写性能各不相同。

连续空间存放方式顾名思义，文件存放在磁盘连续的物理空间中。这种模式下，文件的数据都是紧密相连，读写效率很高，因为一次磁盘寻道就可以读出整个文件。

使用连续存放的方式有一个前提，必须先知道一个文件的大小，这样文件系统才会根据文件的大小在磁盘上找到一块连续的空间分配给文件。所以文件头里需要指定起始块的位置和长度，有了这两个信息就可以很好地表示文件存放方式是一块连续的磁盘空间。

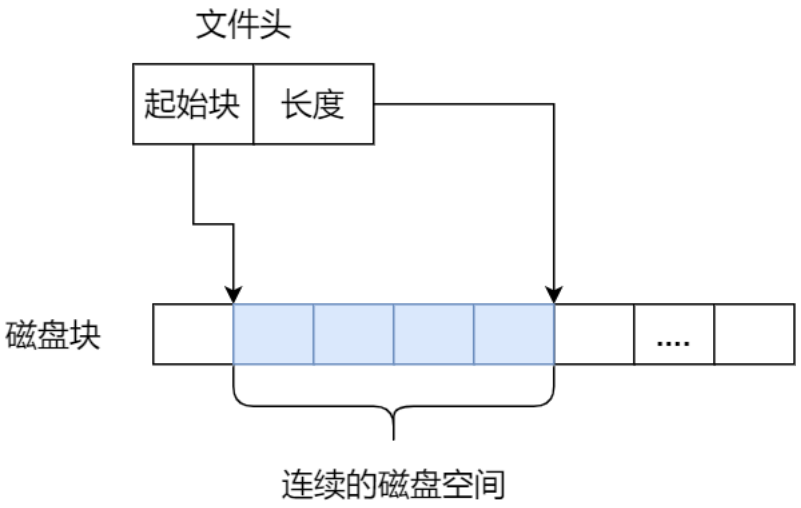


图 2.3: 连续空间存放

连续空间存放的方式虽然读写效率高，但是有磁盘空间碎片和文件长度不易扩展的缺陷。

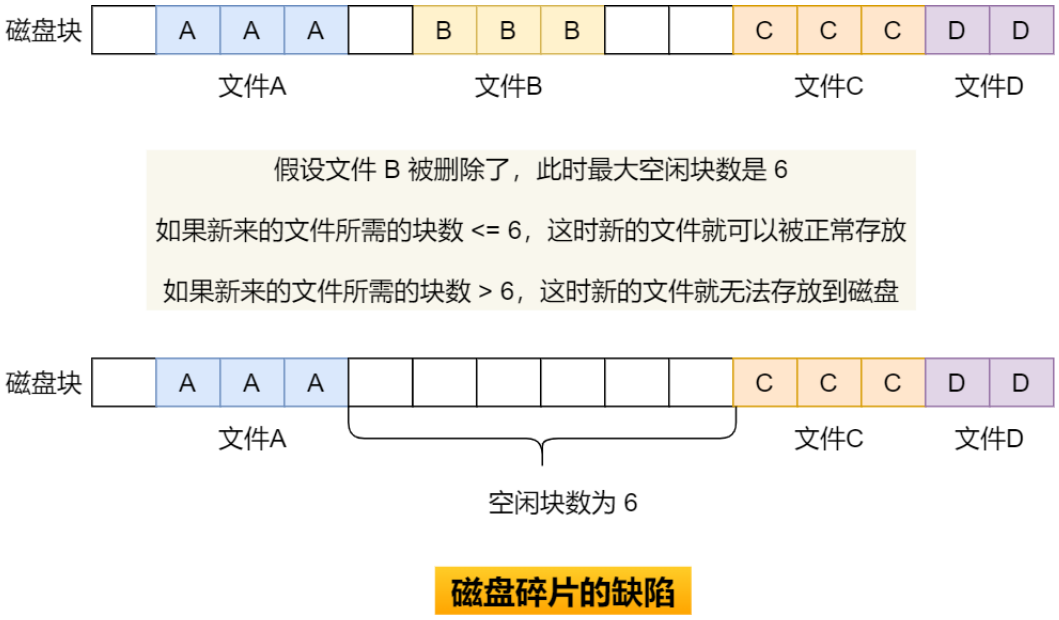


图 2.4: 磁盘碎片

2.4.2 链表分配

链表的方式存放是离散的，于是就可以消除磁盘碎片，可大大提高磁盘空间的利用率，同时文件的长度可以动态扩展。根据实现的方式的不同，链表可分为隐式链表和显式链接两种形式。

隐式链表实现的方式是文件头要包含第一块和最后一块的位置，并且每个数据块里面留出一个指针空间，用来存放下一个数据块的位置，这样从链头开始就可以顺着指针找到所有的数据块，所以存放是不连续的。

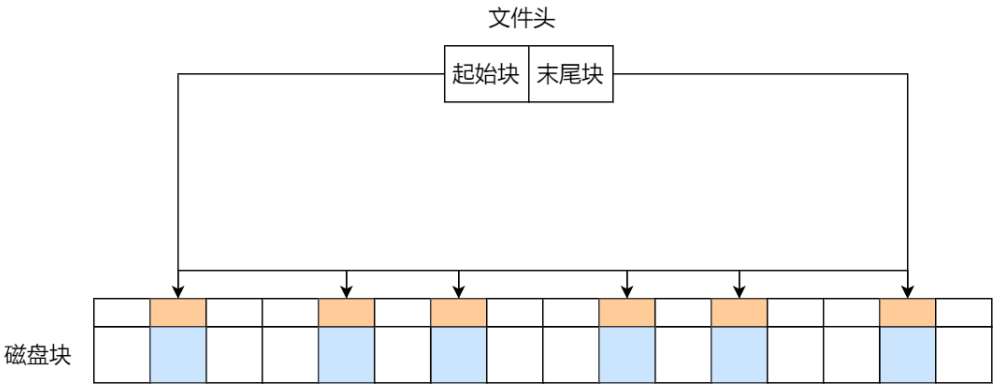


图 2.5: 隐式链表存放

隐式链表存放方式的缺点在于无法直接访问数据块，只能通过指针顺序访问，以及数据块指针消耗了一定的存储空间。隐式链表稳定性较差，系统在运行过程中由于软件或者硬件错误导致指针丢失或损坏，会导致文件数据的丢失。

如果取出每个磁盘块的指针，把它放在内存的一个表中，就可以解决上述隐式链表的两个不足。显式链接指把用于链接文件各数据块的指针，显式地存放在内存的一张链接表中，该表在整个磁盘仅设置一张，每个表项中存放链接指针，指向下一个数据块号。这个表格称为文件分配表（FAT, File Allocation Table）。

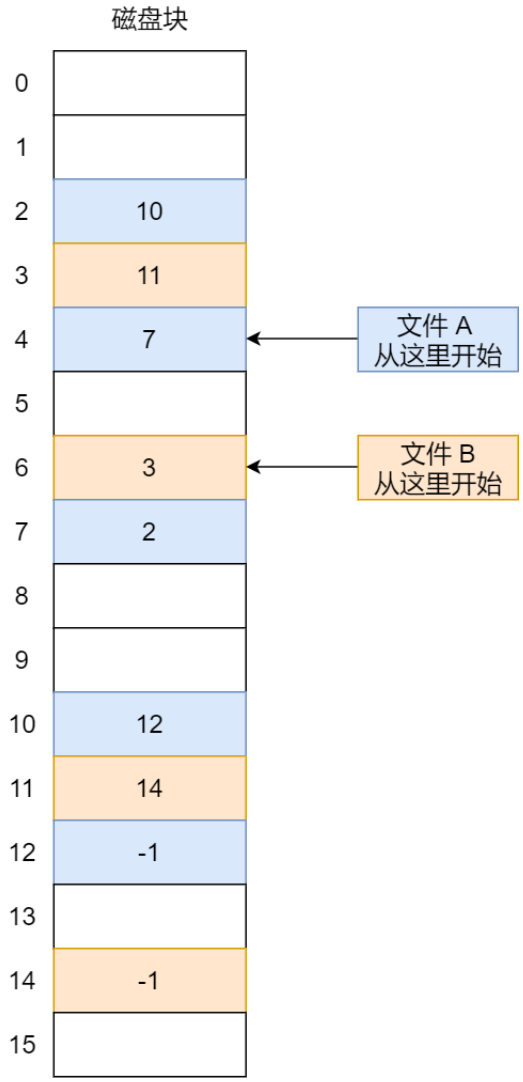


图 2.6: 文件分配表 FAT

2.4.3 索引分配

链表的方式解决了连续分配的磁盘碎片和文件动态扩展的问题，但是不能有效支持直接访问，索引的方式可以解决这个问题。索引的实现是为每个文件创建一个索引数据块，里面存放的是指向文件数据块的指针列表，类似于书的目录，通过目录就可以找到对应章节。文件头需要包含指向索引数据块的指针，这样就可以通过文件头知道索引数据块的位置，再通过索引数据块里的索引信息找到对应的数据块。

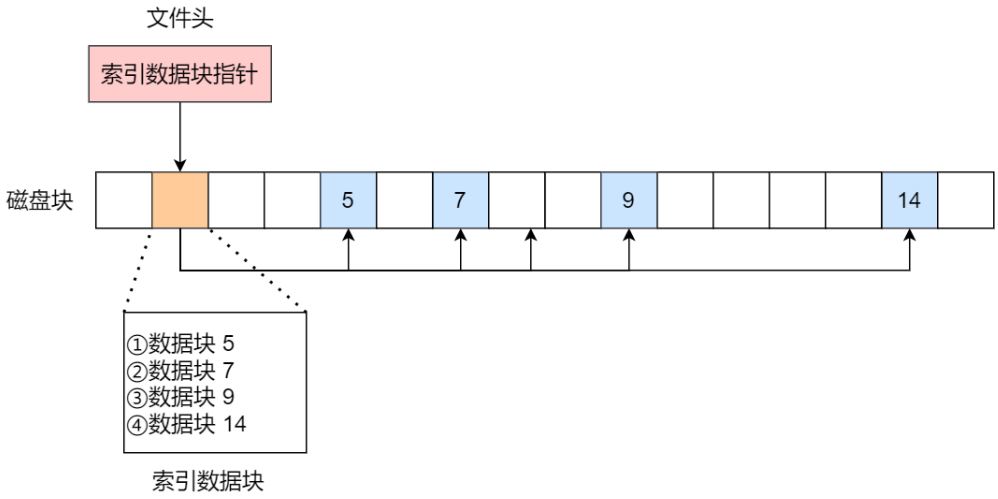


图 2.7: 索引分配

索引的方式优点在于：

1. 文件的创建、增大、缩小很方便
2. 不会有碎片的问题
3. 支持顺序读写和随机读写

如果文件很大，大到一个索引数据块放不下索引信息，这时可以通过链表与索引的组合，这种组合称为链式索引块，它的实现方式是在索引数据块留出一个存放下一个索引数据块的指针，于是当一个索引数据块的索引信息用完了，就可以通过指针的方式，找到下一个索引数据块的信息。

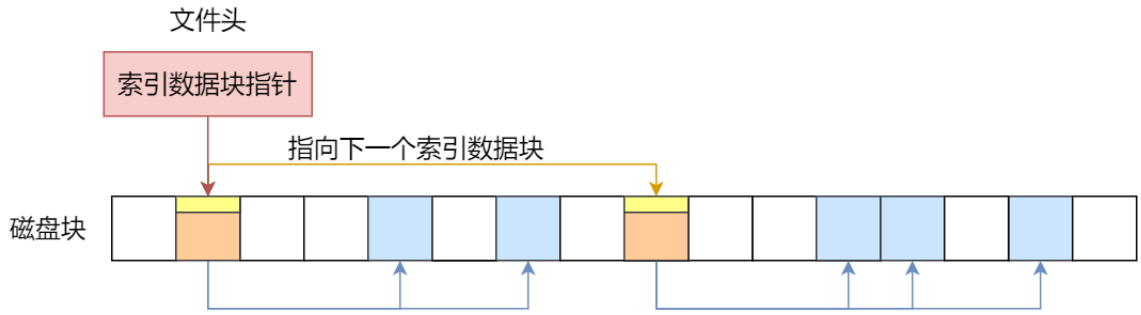


图 2.8: 链式索引块

2.5 空闲空间管理

2.5.1 空闲表法

文件的存储是针对已经被占用的数据块的组织和管理，如果要保存一个数据块，应该放在硬盘上的哪个位置呢？如果将所有的块扫描一遍寻找一个空闲空间，这种方式效率太低了，所以针对磁盘的空闲空间也要引入管理的机制。

空闲表法就是为所有空闲空间建立一张表，表内容包括空闲区的第一个块号和该空闲区的块个数。注意，这个方式是连续分配的。

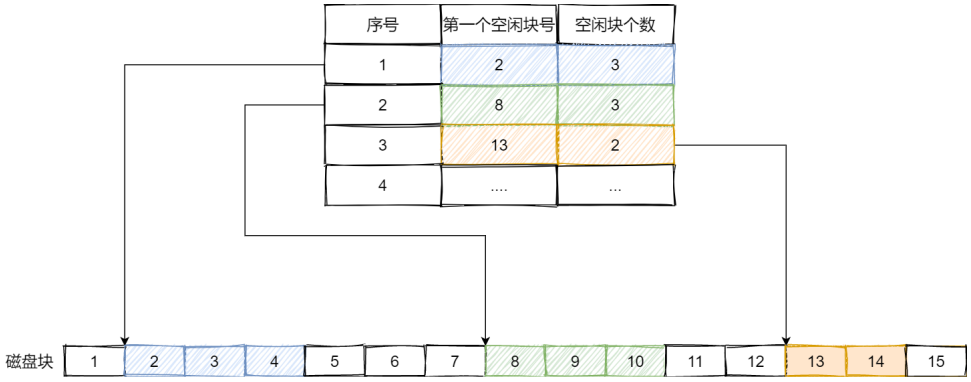


图 2.9: 空闲表法

当请求分配磁盘空间时，系统依次扫描空闲表里的内容，直到找到一个合适的空闲区域为止。当用户撤销一个文件时，系统回收文件空间。这时，也需顺序扫描空闲表，寻找一个空闲表条目并将释放空间的第一个物理块号及它占用的块数填到这个条目中。

这种方法仅当有少量的空闲区时才有较好的效果，因为如果存储空间中有着大量的小的空闲区，则空闲表变得很大，这样查询效率会很低。另外，这种分配技术适用于建立连续文件。

2.5.2 空闲链表法

除了空闲表，也可以使用链表的方式来管理空闲空间，每一个空闲块里有一个指针指向下一个空闲块，这样也能很方便的找到空闲块并管理起来。当创建文件需

要一块或几块时，就从链头上依次取下一块或几块。反之，当回收空间时，把这些空闲块依次接到链头上。

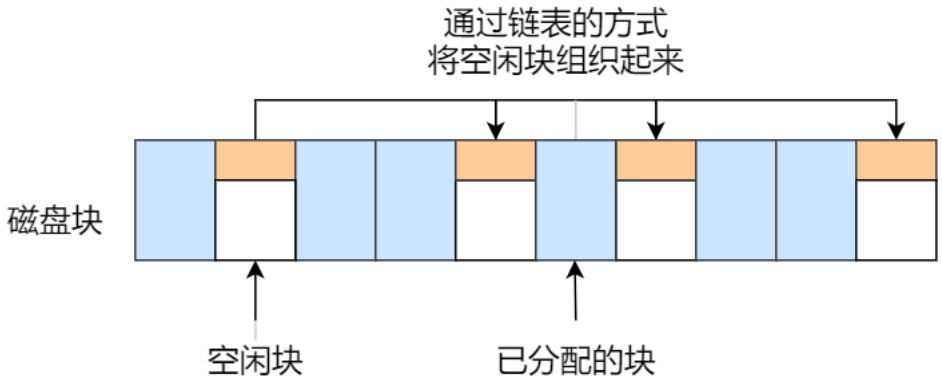


图 2.10: 空闲链表法

这种技术只要在主存中保存一个指针，令它指向第一个空闲块。其特点是简单，但不能随机访问，工作效率低，因为每当在链上增加或移动空闲块时需要做很多 I/O 操作，同时数据块的指针消耗了一定的存储空间。

空闲表法和空闲链表法都不适合用于大型文件系统，因为这会使空闲表或空闲链表太大。

2.5.3 位图法（Bit Table）

位图是利用二进制的一位来表示磁盘中一个盘块的使用情况，磁盘上所有的盘块都有一个二进制位与之对应。当值为 0 时，表示对应的盘块空闲，值为 1 时，表示对应的盘块已分配。Linux 文件系统就采用了位图的方式来管理空闲空间。