

# Linux 环境编程 从应用到内核

高峰 李彬 著

---

Programming in Linux Environment  
from Userspace to Kernel

---

- Linux领域第一本将应用编程与内核实现相结合的图书
- Linux环境编程的进阶指导，解析Linux接口的工作原理，帮助应用开发人员快速深入内核，掌握Linux系统运行机制



机械工业出版社  
China Machine Press

Linux/Unix 技术丛书

# Linux 环境编程：从应用到内核

高峰 李彬 著



机械工业出版社  
China Machine Press

## 图书在版编目 ( CIP ) 数据

Linux 环境编程：从应用到内核 / 高峰, 李彬著. —北京：机械工业出版社, 2016.5  
(Linux/Unix 技术丛书)

ISBN 978-7-111-53610-9

I. L… II. ①高… ②李… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字 ( 2016 ) 第 086961 号



## Linux 环境编程：从应用到内核

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：余 洁

责任校对：殷 虹

印 刷：

版 次：2016 年 6 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：38

书 号：ISBN 978-7-111-53610-9

定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：( 010 ) 88379426 88361066

投稿热线：( 010 ) 88379604

购书热线：( 010 ) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

## 为什么要写这本书

我从事 Linux 环境的开发工作已有近十年的时间，但我一直认为工作时间并不等于经验，更不等于能力。如何才能把工作时间转换为自己的经验和能力呢？我认为无非是多阅读、多思考、多实践、多分享。这也是我在 ChinaUnix 上的博客座右铭，目前我的博客一共有 247 篇博文，记录的大都是 Linux 内核网络部分的源码分析，以及相关的应用编程。机械工业出版社华章公司的 Lisa 正是通过我的博客找到我的，而这也促成了本书的出版。

其实在 Lisa 之前，就有另外一位编辑与我聊过，但当时我没有下好决心，认为自己无论是在技术水平，还是时间安排上，都不足以完成一本技术图书的创作。等到与 Lisa 洽谈的时候，我感觉自己的技术已经有了一些沉淀，同时时间也相对比较充裕，因此决定开始撰写自己技术生涯的第一本书。

对于 Linux 环境的开发人员，《Unix 环境高级编程》（后文均简称为 APUE）无疑是最为经典的入门书籍。其作者 Stevens 是我从业以来最崇拜的技术专家。他的 Advanced Programming in the Unix Environment、Unix Network Programming 系列及 TCP/IP Illustrated 系列著作，字字珠玑，本本经典。在我从业的最初几年，这几本书每本都阅读了好几遍，而这也为我进行 Linux 用户空间的开发奠定了坚实的基础。在掌握了这些知识以后，如何继续提高自己的技能呢？经过一番思考，我选择了阅读 Linux 内核源码，并尝试将内核与应用融会贯通。在阅读了一定量的内核源码之后，我才真正理解了 Linux 专家的这句话“Read the fucking codes”。只有阅读了内核源码，才能真正理解 Linux 内核的原理和运行机制，而此时，我也发现了 Stevens 著作的一个局限——APUE 和 UNP 毕竟是针对 Unix 环境而写的，Linux 虽然大部分与 Unix 兼容，但是在很多行为上与 Unix 还是完全不同的。这就导致了书中的一些内容与 Linux 环境中的实际效果是相互矛盾的。

现在有机会来写一本技术图书，我就想在向 Stevens 致敬的同时，写一本类似于 APUE

风格的技术图书，同时还要在 Linux 环境下，对 APUE 进行突破。大言不惭地说，我期待这本书可以作为 APUE 的补充，还可以作为 Linux 开发人员的进阶读物。事实上，本书的写作布局正是以 APUE 的章节作为参考，针对 Linux 环境，不仅对用户空间的接口进行阐述，同时还引导读者分析该接口在内核的源码实现，使得读者不仅可以知道接口怎么用，同时还可以理解接口是怎么工作的。对于 Linux 的系统调用，做到知其然，知其所以然。

## 读者对象

根据本书的内容，我觉得适合以下几类读者：

- ❑ 在 Linux 应用层方面有一定开发经验的程序员。
- ❑ 对 Linux 内核有兴趣的程序员。
- ❑ 热爱 Linux 内核和开源项目的技术人员。

## 如何阅读本书

本书定位为 APUE 的补充或进阶读物，所以假设读者已具备了一定的编程基础，对 Linux 环境也有所了解，因此在涉及一些基本概念和知识时，只是蜻蜓点水，简单略过。因为笔者希望把更多的笔墨放在更为重要的部分，而不是各种相关图书均有讲解的基本概念上。所以如果你是初学者，建议还是先学习 APUE、C 语言编程，并且在具有一定的操作系统知识后再来阅读本书。

Linux 环境编程涉及的领域太多，很难有某个人可以在 Linux 的各个领域均有比较深刻的认识，尤其是已有 APUE 这本经典图书在前，所以本书是由高峰、李彬两个人共同完成的。

高峰负责第 0、1、2、3、4、12、13、14、15 章，李彬负责第 5~11 章。两位不同的作者，在写作风格上很难保证一致，如果给各位读者带来了不便，在此给各位先道个歉。尽管是由两个人共同写作，并且负责的还是我们各自相对擅长的领域，可是在写作的过程中我们仍然感觉到很吃力，用了将近三年的时间才算完成本书。对比 APUE，本书一方面在深度上还是有所不及，另一方面在广度上还是没有涵盖 APUE 涉及的所有领域，这也让我们对 Stevens 大师更加敬佩。

本书使用的 Linux 内核源代码版本为 3.2.44，glibc 的源码版本为 2.17。

## 勘误和支持

由于作者的水平有限，主题又过于宏大，书中难免会出现一些错误或不准确的地方，

如有不妥之处，恳请读者批评指正。如果你发现有什么问题，或者有什么疑问，都可以发邮件至我的邮箱 [gfree.wind@gmail.com](mailto:gfree.wind@gmail.com)，期待您的指导！

## 致谢

首先要感谢伟大的 Linux 内核创始人 Linus，他开创了一个影响世界的操作系统。

其次要感谢机械工业出版社华章公司的编辑杨绣国老师（Lisa），感谢你的魄力，敢于找新人来写作，并敢于信任新人，让其完成这么大的一个项目。感谢你的耐心，正常的一年半的写作时间，被我们生生地延长到了将近三年的时间，感谢你在写作过程中对我们的鼓励和帮助。

然后要感谢我的搭档李彬，在我加入当前的创业公司后，只有很少的空闲时间和精力来投入写作。这时，是李彬在更紧张的时间内，承担了本书的一半内容。并且其写作态度极其认真，对质量精益求精。没有李彬的加入，本书很可能就半途而废了。再次感谢李彬，我的好搭档。

最后我要感谢我的亲人。感谢我的父母，没有你们的培养，绝没有我的今天；感谢我的妻子，没有你的支持，就没有我事业上的进步；感谢我的岳父岳母对我女儿的照顾，使我没有后顾之忧；最后要感谢的是我可爱的女儿高一涵小天使，你的诞生为我带来了无尽的欢乐和动力！

谨以此书，献给我最亲爱的家人，以及众多热爱 Linux 的朋友们。

高 峰  
中国北京  
2016 年 3 月

# 目 录 *Contents*

前 言	1.2.4 如何选择文件描述符 .....	17
第 0 章 基础知识 .....	1.2.5 文件描述符 fd 与文件管理结构 file .....	18
0.1 一个 Linux 程序的诞生记 .....	1.3 creat 简介 .....	19
0.2 程序的构成 .....	1.4 关闭文件 .....	19
0.3 程序是如何“跑”的 .....	1.4.1 close 介绍 .....	19
0.4 背景概念介绍 .....	1.4.2 close 源码跟踪 .....	19
0.4.1 系统调用 .....	1.4.3 自定义 files_operations .....	21
0.4.2 C 库函数 .....	1.4.4 遗忘 close 造成的问题 .....	22
0.4.3 线程安全 .....	1.4.5 如何查找文件资源泄漏 .....	25
0.4.4 原子性 .....	1.5 文件偏移 .....	26
0.4.5 可重入函数 .....	1.5.1 lseek 简介 .....	26
0.4.6 阻塞与非阻塞 .....	1.5.2 小心 lseek 的返回值 .....	26
0.4.7 同步与非同步 .....	1.5.3 lseek 源码分析 .....	27
第 1 章 文件 I/O .....	1.6 读取文件 .....	29
1.1 Linux 中的文件 .....	1.6.1 read 源码跟踪 .....	29
1.1.1 文件、文件描述符和文件表 .....	1.6.2 部分读取 .....	30
1.1.2 内核文件表的实现 .....	1.7 写入文件 .....	31
1.2 打开文件 .....	1.7.1 write 源码跟踪 .....	31
1.2.1 open 介绍 .....	1.7.2 追加写的实现 .....	33
1.2.2 更多选项 .....	1.8 文件的原子读写 .....	33
1.2.3 open 源码跟踪 .....	1.9 文件描述符的复制 .....	34
	1.10 文件数据的同步 .....	38

1.11 文件的元数据.....41	3.5.2 编译生成和使用动态库 ..... 80
1.11.1 获取文件的元数据.....41	3.5.3 程序的“平滑无缝”升级 ..... 82
1.11.2 内核如何维护文件的元数据.....42	3.6 避免内存问题..... 84
1.11.3 权限位解析.....43	3.6.1 尴尬的 realloc..... 84
1.12 文件截断.....45	3.6.2 如何防止内存越界 ..... 85
1.12.1 truncate 与 ftruncate 的简单 介绍 .....45	3.6.3 如何定位内存问题 ..... 86
1.12.2 文件截断的内核实现 .....45	3.7 “长跳转” longjmp ..... 90
1.12.3 为什么需要文件截断 .....48	3.7.1 setjmp 与 longjmp 的使用 ..... 90
<b>第2章 标准 I/O 库</b> ..... 50	3.7.2 “长跳转”的实现机制 ..... 91
2.1 stdin、stdout 和 stderr..... 50	3.7.3 “长跳转”的陷阱 ..... 93
2.2 I/O 缓存引出的趣题 ..... 51	<b>第4章 进程控制：进程的一生</b> ..... 96
2.3 fopen 和 open 标志位对比..... 52	4.1 进程 ID..... 96
2.4 fdopen 与 fileno ..... 55	4.2 进程的层次 ..... 98
2.5 同时读写的痛苦 ..... 56	4.2.1 进程组 ..... 99
2.6 ferror 的返回值..... 57	4.2.2 会话 ..... 102
2.7 clearerr 的用途 ..... 57	4.3 进程的创建之 fork()..... 103
2.8 小心 fgetc 和 getc..... 60	4.3.1 fork 之后父子进程的内存关系... 104
2.9 注意 fread 和 fwrite 的返回值..... 60	4.3.2 fork 之后父子进程与文件的 关系 ..... 107
2.10 创建临时文件..... 61	4.3.3 文件描述符复制的内核实现 ..... 110
<b>第3章 进程环境</b> ..... 66	4.4 进程的创建之 vfork() ..... 115
3.1 main 是 C 程序的开始吗 ..... 66	4.5 daemon 进程的创建 ..... 117
3.2 “活雷锋” exit..... 70	4.6 进程的终止 ..... 119
3.3 atexit 介绍..... 75	4.6.1 _exit 函数 ..... 119
3.3.1 使用 atexit..... 75	4.6.2 exit 函数 ..... 120
3.3.2 atexit 的局限性..... 76	4.6.3 return 退出 ..... 122
3.3.3 atexit 的实现机制..... 77	4.7 等待子进程 ..... 122
3.4 小心使用环境变量 ..... 78	4.7.1 僵尸进程 ..... 122
3.5 使用动态库 ..... 80	4.7.2 等待子进程之 wait()..... 124
3.5.1 动态库与静态库 ..... 80	4.7.3 等待子进程之 waitpid()..... 126
	4.7.4 等待子进程之等待状态值 ..... 129



4.7.5 等待子进程之 waitid().....	131	5.7 CPU 的亲合力.....	214
4.7.6 进程退出和等待的内核实现 .....	133	<b>第 6 章 信号</b> .....	219
4.8 exec 家族.....	141	6.1 信号的完整生命周期.....	219
4.8.1 execve 函数.....	141	6.2 信号的产生.....	220
4.8.2 exec 家族.....	142	6.2.1 硬件异常.....	220
4.8.3 execve 系统调用的内核实现 .....	144	6.2.2 终端相关的信号.....	221
4.8.4 exec 与信号.....	151	6.2.3 软件事件相关的信号.....	223
4.8.5 执行 exec 之后进程继承的 属性.....	152	6.3 信号的默认处理函数.....	224
4.9 system 函数.....	152	6.4 信号的分类.....	227
4.9.1 system 函数接口.....	153	6.5 传统信号的特点.....	228
4.9.2 system 函数与信号.....	156	6.5.1 信号的 ONESHOT 特性.....	230
4.10 总结.....	157	6.5.2 信号执行时屏蔽自身的特性 .....	232
<b>第 5 章 进程控制：状态、调度和 优先级</b> .....	158	6.5.3 信号中断系统调用的重启特性.....	233
5.1 进程的状态.....	158	6.6 信号的可靠性.....	236
5.1.1 进程状态概述.....	159	6.6.1 信号的可靠性实验.....	236
5.1.2 观察进程状态.....	171	6.6.2 信号可靠性差异的根源.....	240
5.2 进程调度概述.....	173	6.7 信号的安装.....	243
5.3 普通进程的优先级.....	181	6.8 信号的发送.....	246
5.4 完全公平调度的实现.....	186	6.8.1 kill、tkill 和 tkill.....	246
5.4.1 时间片和虚拟运行时间.....	186	6.8.2 raise 函数.....	247
5.4.2 周期性调度任务.....	190	6.8.3 sigqueue 函数.....	247
5.4.3 新进程的加入.....	192	6.9 信号与线程的关系.....	253
5.4.4 睡眠进程醒来.....	198	6.9.1 线程之间共享信号处理函数 .....	254
5.4.5 唤醒抢占.....	202	6.9.2 线程有独立的阻塞信号掩码 .....	255
5.5 普通进程的组调度.....	204	6.9.3 私有挂起信号和共享挂起 信号.....	257
5.6 实时进程.....	207	6.9.4 致命信号下，进程组全体 退出.....	260
5.6.1 实时调度策略和优先级.....	207	6.10 等待信号.....	260
5.6.2 实时调度相关 API.....	211	6.10.1 pause 函数.....	261
5.6.3 限制实时进程运行时间.....	213	6.10.2 sigsuspend 函数.....	262

6.10.3	sigwait 函数和 sigwaitinfo 函数	263	7.9	性能杀手：伪共享	323
6.11	通过文件描述符来获取信号	265	7.10	条件等待	328
6.12	信号递送的顺序	267	7.10.1	条件变量的创建和销毁	328
6.13	异步信号安全	272	7.10.2	条件变量的使用	329
6.14	总结	275			
<b>第 7 章</b>	<b>理解 Linux 线程 (1)</b>	276	<b>第 8 章</b>	<b>理解 Linux 线程 (2)</b>	333
7.1	线程与进程	276	8.1	线程取消	333
7.2	进程 ID 和线程 ID	281	8.1.1	函数取消接口	333
7.3	pthread 库接口介绍	284	8.1.2	线程清理函数	335
7.4	线程的创建和标识	285	8.2	线程局部存储	339
7.4.1	pthread_create 函数	285	8.2.1	使用 NPTL 库函数实现线程局部存储	340
7.4.2	线程 ID 及进程地址空间布局	286	8.2.2	使用 __thread 关键字实现线程局部存储	342
7.4.3	线程创建的默认属性	291	8.3	线程与信号	343
7.5	线程的退出	292	8.3.1	设置线程的信号掩码	344
7.6	线程的连接与分离	293	8.3.2	向线程发送信号	344
7.6.1	线程的连接	293	8.3.3	多线程程序对信号的处理	345
7.6.2	为什么要连接退出的线程	295	8.4	多线程与 fork()	345
7.6.3	线程的分离	299			
7.7	互斥量	300	<b>第 9 章</b>	<b>进程间通信：管道</b>	349
7.7.1	为什么需要互斥量	300	9.1	管道	351
7.7.2	互斥量的接口	304	9.1.1	管道概述	351
7.7.3	临界区的大小	305	9.1.2	管道接口	352
7.7.4	互斥量的性能	306	9.1.3	关闭未使用的管道文件描述符	356
7.7.5	互斥锁的公平性	310	9.1.4	管道对应的内存区大小	361
7.7.6	互斥锁的类型	311	9.1.5	shell 管道的实现	361
7.7.7	死锁和活锁	314	9.1.6	与 shell 命令进行通信 (popen)	362
7.8	读写锁	316	9.2	命名管道 FIFO	365
7.8.1	读写锁的接口	317	9.2.1	创建 FIFO 文件	365
7.8.2	读写锁的竞争策略	318	9.2.2	打开 FIFO 文件	366
7.8.3	读写锁总结	323			

9.3 读写管道文件 .....	367
9.4 使用管道通信的示例 .....	372

## 第 10 章 进程间通信: System V

IPC .....	375
10.1 System V IPC 概述 .....	375
10.1.1 标识符与 IPC Key .....	376
10.1.2 IPC 的公共数据结构 .....	379
10.2 System V 消息队列 .....	383
10.2.1 创建或打开一个消息队列 .....	383
10.2.2 发送消息 .....	385
10.2.3 接收消息 .....	388
10.2.4 控制消息队列 .....	390
10.3 System V 信号量 .....	391
10.3.1 信号量概述 .....	391
10.3.2 创建或打开信号量 .....	393
10.3.3 操作信号量 .....	395
10.3.4 信号量撤销值 .....	399
10.3.5 控制信号量 .....	400
10.4 System V 共享内存 .....	402
10.4.1 共享内存概述 .....	402
10.4.2 创建或打开共享内存 .....	403
10.4.3 使用共享内存 .....	405
10.4.4 分离共享内存 .....	407
10.4.5 控制共享内存 .....	408

## 第 11 章 进程间通信: POSIX IPC ..410

11.1 POSIX IPC 概述 .....	411
11.1.1 IPC 对象的名字 .....	411
11.1.2 创建或打开 IPC 对象 .....	413
11.1.3 关闭和删除 IPC 对象 .....	414
11.1.4 其他 .....	414

11.2 POSIX 消息队列 .....	415
11.2.1 消息队列的创建、打开、关闭及删除 .....	415
11.2.2 消息队列的属性 .....	418
11.2.3 消息的发送和接收 .....	422
11.2.4 消息的通知 .....	423
11.2.5 I/O 多路复用监控消息队列 .....	427
11.3 POSIX 信号量 .....	428
11.3.1 创建、打开、关闭和删除有名信号量 .....	430
11.3.2 信号量的使用 .....	431
11.3.3 无名信号量的创建和销毁 .....	432
11.3.4 信号量与 futex .....	433
11.4 内存映射 mmap .....	436
11.4.1 内存映射概述 .....	436
11.4.2 内存映射的相关接口 .....	438
11.4.3 共享文件映射 .....	439
11.4.4 私有文件映射 .....	455
11.4.5 共享匿名映射 .....	455
11.4.6 私有匿名映射 .....	456
11.5 POSIX 共享内存 .....	456
11.5.1 共享内存的创建、使用和删除 .....	457
11.5.2 共享内存与 tmpfs .....	458

## 第 12 章 网络通信: 连接的建立 ..462

12.1 socket 文件描述符 .....	462
12.2 绑定 IP 地址 .....	463
12.2.1 bind 的使用 .....	464
12.2.2 bind 的源码分析 .....	465
12.3 客户端连接过程 .....	468
12.3.1 connect 的使用 .....	468

12.3.2 connect 的源码分析 .....	469	14.2 数据包从内核空间到用户空间的 流程 .....	537
12.4 服务器端连接过程 .....	477	14.3 UDP 数据包的接收流程 .....	540
12.4.1 listen 的使用 .....	477	14.4 TCP 数据包的接收流程 .....	544
12.4.2 listen 的源码分析 .....	478	14.5 TCP 套接字的三个接收队列 .....	553
12.4.3 accept 的使用 .....	480	14.6 从网卡到套接字 .....	556
12.4.4 accept 的源码分析 .....	480	14.6.1 从硬中断到软中断 .....	556
12.5 TCP 三次握手的实现分析 .....	483	14.6.2 软中断处理 .....	557
12.5.1 SYN 包的发送 .....	483	14.6.3 传递给协议栈流程 .....	559
12.5.2 接收 SYN 包, 发送 SYN+ ACK 包 .....	485	14.6.4 IP 协议处理流程 .....	564
12.5.3 接收 SYN+ACK 数据包 .....	494	14.6.5 大师的错误? 原始套接字的 接收 .....	568
12.5.4 接收 ACK 数据包, 完成三次 握手 .....	499	14.6.6 注册传输层协议 .....	571
<b>第 13 章 网络通信: 数据报文的 发送 .....</b>	<b>505</b>	14.6.7 确定 UDP 套接字 .....	571
13.1 发送相关接口 .....	505	14.6.8 确定 TCP 套接字 .....	576
13.2 数据包从用户空间到内核空间的 流程 .....	506	<b>第 15 章 编写安全无错代码 .....</b>	<b>582</b>
13.3 UDP 数据包的发送流程 .....	510	15.1 不要用 memcmp 比较结构体 .....	582
13.4 TCP 数据包的发送流程 .....	517	15.2 有符号数和无符号数的移位 区别 .....	583
13.5 IP 数据包的发送流程 .....	527	15.3 数组和指针 .....	584
13.5.1 ip_send_skb 源码分析 .....	528	15.4 再论数组首地址 .....	587
13.5.2 ip_queue_xmit 源码分析 .....	531	15.5 “神奇”的整数类型转换 .....	588
13.6 底层模块数据包的发送流程 .....	532	15.6 小心 volatile 的原子性误解 .....	589
<b>第 14 章 网络通信: 数据报文的 接收 .....</b>	<b>536</b>	15.7 有趣的问题: “x == x” 何时 为假? .....	591
14.1 系统调用接口 .....	536	15.8 小心浮点陷阱 .....	593
		15.8.1 浮点数的精度限制 .....	593
		15.8.2 两个特殊的浮点值 .....	593
		15.9 Intel 移位指令陷阱 .....	595



# 基础知识

基础知识是构建技术大厦不可或缺的稳定基石，因此，本书首先来介绍一下书中所涉及的一些基础知识。这里以第 0 章命名，表明我们要注重基础，从 0 开始，同时也是向伟大的 C 语言致敬。

基础知识看似简单，但是想要真正理解它们，是需要花一番功夫的。除了需要积累经验以外，更需要对它们进行不断的思考和理解，这样，才能写出高可靠性的程序。这些基础知识很多都可以独立成文，限于篇幅，这里只能是简单的介绍，都是笔者根据自己的经验和理解进行的总结和概括，相信对读者会有所帮助。感兴趣的朋友可以自己查找更多的资料，以得到更准确、更细致的介绍。



**注意** 本书中的示例代码为了简洁明了，没有考虑代码的健壮性，例如不检查函数的返回值、使用全局变量等。

## 0.1 一个 Linux 程序的诞生记

一本编程书籍如果开篇不写一个“hello world”，就违背了“自古以来”的传统了。因此本节也将以 hello world 为例来说明一个 Linux 程序的诞生过程，示例代码如下：

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

下面使用 gcc 生成可执行程序：`gcc -g -Wall 0_1_hello_world.c -o hello_world`。这样，一个 Linux 可执行程序就诞生了。

整个过程看似简单，其实涉及预处理、编译、汇编和链接等多个步骤。只不过 gcc 作为一个工具集自动完成了所有的步骤。下面就分别来看看其中所涉及各个步骤。

首先来了解一下什么是预处理。预处理用于处理预处理命令。对于上面的代码来说，唯一的预处理命令就是 `#include`。它的作用是将头文件的内容包含到本文件中。注意，这里的“包含”指的是该头文件中的所有代码都会在 `#include` 处展开。可以通过“`gcc -E 0_1_hello_world.c`”在预处理后自动停止后面的操作，并把预处理的结果输出到标准输出。因此使用“`gcc -E 0_1_hello_world.c > 0_1_hello_world.i`”，可得到预处理后的文件。

理解了预处理，在出现一些常见的错误时，才能明白其中的原因。比如，为什么不能在头文件中定义全局变量？这是因为定义全局变量的代码会存在于所有以 `#include` 包含该头文件的文件中，也就是说所有的这些文件，都会定义一个同样的全局变量，这样就不可避免地造成了冲突。

编译环节是指对源代码进行语法分析，并优化产生对应的汇编代码的过程。同样，可以使用 gcc 得到汇编代码，而非最终的二进制文件，即“`gcc -S 0_1_hello_world.c -o 0_1_hello_world.s`”。gcc 的 `-S` 选项会让 gcc 在编译完成后停止后面的工作，这样只会产生对应的汇编文件。

汇编的过程比较简单，就是将源代码翻译成可执行的指令，并生成目标文件。对应的 gcc 命令为“`gcc -c 0_1_hello_world.c -o 0_1_hello_world.o`”。

链接是生成最终可执行程序的最后一步，也是比较复杂的一步。它的工作就是将各个目标文件——包括库文件（库文件也是一种目标文件）链接成一个可执行程序。在这个过程中，涉及的概念比较多，如地址和空间的分配、符号解析、重定位等。在 Linux 环节下，该工作是由 GNU 的链接器 ld 完成的。

实际上我们可以使用 `-v` 选项来查看完整和详细的 gcc 编译过程，命令如下。

```
gcc -g -Wall -v 0_1_hello_world.c -o hello_world。
```

由于输出过多，此处就不粘贴结果了。感兴趣的朋友可以自行执行命令，查看输出。通过 `-v` 选项，可以看到 gcc 在背后做了哪些具体的工作。

## 0.2 程序的构成

Linux 下二进制可执行程序的格式一般为 ELF 格式。以 0.1 节的 hello world 为例，使用 `readelf` 查看其 ELF 格式，内容如下：

```
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
```

```

OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x8048320
Start of program headers: 52 (bytes into file)
Start of section headers: 5148 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 9
Size of section headers: 40 (bytes)
Number of section headers: 36
Section header string table index: 33
Section Headers:
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al
[ 0] NULL 00000000 000000 000000 00 0 0 0
[ 1] .interp PROGBITS 08048154 000154 000013 00 A 0 0 1
[ 2] .note.ABI-tag NOTE 08048168 000168 000020 00 A 0 0 4
[ 3] .note.gnu.build-id NOTE 08048188 000188 000024 00 A 0 0 4
[ 4] .gnu.hash GNU_HASH 080481ac 0001ac 000020 04 A 5 0 4
[ 5] .dynsym DYNSYM 080481cc 0001cc 000050 10 A 6 1 4
[ 6] .dynstr STRTAB 0804821c 00021c 00004a 00 A 0 0 1
[ 7] .gnu.version VERSYM 08048266 000266 00000a 02 A 5 0 2
[ 8] .gnu.version_r VERNEED 08048270 000270 000020 00 A 6 1 4
[ 9] .rel.dyn REL 08048290 000290 000008 08 A 5 0 4
[10] .rel.plt REL 08048298 000298 000018 08 A 5 12 4
[11] .init PROGBITS 080482b0 0002b0 000024 00 AX 0 0 4
[12] .plt PROGBITS 080482e0 0002e0 000040 04 AX 0 0 16
[13] .text PROGBITS 08048320 000320 000188 00 AX 0 0 16
[14] .fini PROGBITS 080484a8 0004a8 000015 00 AX 0 0 4
[15] .rodata PROGBITS 080484c0 0004c0 000015 00 A 0 0 4
[16] .eh_frame_hdr PROGBITS 080484d8 0004d8 000034 00 A 0 0 4
[17] .eh_frame PROGBITS 0804850c 00050c 0000c4 00 A 0 0 4
[18] .init_array INIT_ARRAY 08049f08 000f08 000004 00 WA 0 0 4
[19] .fini_array FINI_ARRAY 08049f0c 000f0c 000004 00 WA 0 0 4
[20] .jcr PROGBITS 08049f10 000f10 000004 00 WA 0 0 4
[21] .dynamic DYNAMIC 08049f14 000f14 0000e8 08 WA 6 0 4
[22] .got PROGBITS 08049ffc 000ffc 000004 04 WA 0 0 4
[23] .got.plt PROGBITS 0804a000 001000 000018 04 WA 0 0 4
[24] .data PROGBITS 0804a018 001018 000008 00 WA 0 0 4
[25] .bss NOBITS 0804a020 001020 000004 00 WA 0 0 4
[26] .comment PROGBITS 00000000 001020 00006b 01 MS 0 0 1
[27] .debug_aranges PROGBITS 00000000 00108b 000020 00 0 0 1
[28] .debug_info PROGBITS 00000000 0010ab 000094 00 0 0 1
[29] .debug_abbrev PROGBITS 00000000 00113f 000044 00 0 0 1
[30] .debug_line PROGBITS 00000000 001183 000043 00 0 0 1
[31] .debug_str PROGBITS 00000000 0011c6 0000cb 01 MS 0 0 1
[32] .debug_loc PROGBITS 00000000 001291 000038 00 0 0 1
[33] .shstrtab STRTAB 00000000 0012c9 000151 00 0 0 1
[34] .symtab SYMTAB 00000000 0019bc 000490 10 35 51 4
[35] .strtab STRTAB 00000000 001e4c 00025a 00 0 0 1

```

由于输出过多，后面的结果并没有完全展示出来。ELF 文件的主要内容就是由各个



section 及 symbol 表组成的。在上面的 section 列表中，大家最熟悉的应该是 text 段、data 段和 bss 段。text 段为代码段，用于保存可执行指令。data 段为数据段，用于保存有非 0 初始值的全局变量和静态变量。bss 段用于保存没有初始值或初值为 0 的全局变量和静态变量，当程序加载时，bss 段中的变量会被初始化为 0。这个段并不占用物理空间——因为完全没有必要，这些变量的值固定初始化为 0，因此何必占用宝贵的物理空间？

其他段没有这三个段有名，下面来介绍一下其中一些比较常见的段：

- ❑ debug 段：顾名思义，用于保存调试信息。
- ❑ dynamic 段：用于保存动态链接信息。
- ❑ fini 段：用于保存进程退出时的执行程序。当进程结束时，系统会自动执行这部分代码。
- ❑ init 段：用于保存进程启动时的执行程序。当进程启动时，系统会自动执行这部分代码。
- ❑ rodata 段：用于保存只读数据，如 const 修饰的全局变量、字符串常量。
- ❑ symtab 段：用于保存符号表。

其中，对于与调试相关的段，如果不使用 -g 选项，则不会生成，但是与符号相关的段仍然会存在，这时可以使用 strip 去掉符号信息，感兴趣的朋友可以自己参考 strip 的说明进行实验。一般在嵌入式的产品中，为了减少程序占用的空间，都会使用 strip 去掉非必要的段。

### 0.3 程序是如何“跑”的

在日常工作中，我们经常会说“程序‘跑’起来了”，那么它到底是怎么“跑”的呢？在 Linux 环境下，可以使用 strace 跟踪系统调用，从而帮助自己研究系统程序加载、运行和退出的过程。此处仍然以 hello\_world 为例。

```
strace ./hello_world
execve("./hello_world", [ "./hello_world" ], [ /* 59 vars */ ]) = 0
brk(0) = 0x872a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
    0xb7778000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=80063, ...}) = 0
mmap2(NULL, 80063, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7764000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\0\226\1\0004\0\0\0"...
    512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1730024, ...}) = 0
mmap2(NULL, 1743580, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75ba000
mprotect(0xb775d000, 4096, PROT_NONE) = 0
mmap2(0xb775e000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
```



```

DENYWRITE, 3, 0x1a3) = 0xb775e000
mmap2(0xb7761000, 10972, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
ANONYMOUS, -1, 0) = 0xb7761000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0xb75b9000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb75b9900, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb775e000, 8192, PROT_READ) = 0
mprotect(0x8049000, 4096, PROT_READ) = 0
mprotect(0xb779b000, 4096, PROT_READ) = 0
munmap(0xb7764000, 80063) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7777000
write(1, "Hello world!\n", 13Hello world!
) = 13
exit_group(0) = ?

```

下面就针对 `strace` 输出说明其含义。在 Linux 环境中，执行一个命令时，首先是由 shell 调用 `fork`，然后在子进程中来真正执行这个命令（这一过程在 `strace` 输出中无法体现）。`strace` 是 `hello_world` 开始执行后的输出。首先是调用 `execve` 来加载 `hello_world`，然后 `ld` 会分别检查 `ld.so.nohwcap` 和 `ld.so.preload`。其中，如果 `ld.so.nohwcap` 存在，则 `ld` 会加载其中未优化版本的库。如果 `ld.so.preload` 存在，则 `ld` 会加载其中的库——在一些项目中，我们需要拦截或替换系统调用或 C 库，此时就会利用这个机制，使用 `LD_PRELOAD` 来实现。之后利用 `mmap` 将 `ld.so.cache` 映射到内存中，`ld.so.cache` 中保存了库的路径，这样就完成了所有的准备工作。接着 `ld` 加载 c 库——`libc.so.6`，利用 `mmap` 及 `mprotect` 设置程序的各个内存区域，到这里，程序运行的环境已经完成。后面的 `write` 会向文件描述符 1（即标准输出）输出 "Hello world!\n"，返回值为 13，它表示 `write` 成功的字符个数。最后调用 `exit_group` 退出程序，此时参数为 0，表示程序退出的状态——此例中 `hello-world` 程序返回 0。

## 0.4 背景概念介绍

### 0.4.1 系统调用

系统调用是操作系统提供的服务，是应用程序与内核通信的接口。在 x86 平台上，有多种陷入内核的途径，最早是通过 `int 0x80` 指令来实现的，后来 Intel 增加了一个新的指令 `sysenter` 来代替 `int 0x80`——其他 CPU 厂商也增加了类似的指令。新指令 `sysenter` 的性能消耗大约是 `int 0x80` 的一半左右。即使是这样，相对于普通的函数调用来说，系统调用的性能消耗也是巨大的。所以在追求极致性能的程序中，都在尽力避免系统调用，譬如 C 库的 `gettimeofday` 就避免了系统调用。

用户空间的程序默认是通过栈来传递参数的。对于系统调用来说，内核态和用户态使用的是不同的栈，这使得系统调用的参数只能通过寄存器的方式进行传递。

有心的朋友可能会想到一个问题：在写代码的时候，程序员根本不用关心参数是如何传递的，编译器已经默默地为我们做了一切——压栈、出栈、保存返回地址等操作，但是编译器如何知道调用的函数是普通函数，还是系统调用呢？如果是后者，编译器就不能简单地使用栈来传递参数了。

为了解决这个问题，我们就要看 0.4.2 节介绍的 C 库函数了。

## 0.4.2 C 库函数

0.4.1 节提到 C 库函数为编译器解决了系统调用的问题。Linux 环境下，使用的 C 库一般都是 glibc，它封装了几乎所有的系统调用，代码中使用的“系统调用”，实际上就是调用 C 库中的函数。C 库函数同样位于用户态，所以编译器可以统一处理所有的函数调用，而不用区分该函数到底是不是系统调用。

下面以具体的系统调用 open 来看看 glibc 库是如何封装系统调用的。在 glibc 的代码中，用了大量的编译器特性以及编程的技巧，可读性不高。open 在 glibc 中对应的实现函数实际上是 \_\_open\_nocancel。至于如何定位到它，感兴趣的朋友可以用 \_\_open\_nocancel 或 open 作为关键字，在 glibc 的源码中搜索，找出它们之间的关系。

```
int
__open_nocancel (const char *file, int oflag, ...)
{
    int mode = 0;

    if (oflag & O_CREAT)
    {
        va_list arg;
        va_start (arg, oflag);
        mode = va_arg (arg, int);
        va_end (arg);
    }

    return INLINE_SYSCALL (openat, 4, AT_FDCWD, file, oflag, mode);
}
```

其中 INLINE\_SYSCALL 是我们关心的内容，这个宏完成了对真正系统调用的封装：INLINE\_SYSCALL->INTERNAL\_SYSCALL。实现 INTERNAL\_SYSCALL 的一个实例为。

```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
({ \
    register unsigned int resultvar; \
    EXTRAVAR_##nr \
    asm volatile ( \
        LOADARGS_##nr \
        "movl %1, %%eax\n\t" \
        "int $0x80\n\t" \
        RESTOREARGS_##nr \
        : "=a" (resultvar) \
        : "i" (__NR_##name) ASMFMT_##nr(args) : "memory", "cc"); \
    err = resultvar; \
})
```

```
(int) resultvar; })
```

其中，关键的代码是用嵌入式汇编写的，在此只做简单说明。“`move %1, %%eax`”表示将第一个参数（即 `__NR_##name`）赋给寄存器 `eax`。`__NR_##name` 为对应的系统调用号，对于本例中的 `open` 来说，其为 `__NR_openat`。系统调用号在文件 `/usr/include/asm/unistd_32(64).h` 中定义，代码如下：

```
[fgao@fgao understanding_apue]#cat /usr/include/asm/unistd_32.h | grep openat
#define __NR_openat 295
```

也就是说，在 Linux 平台下，系统调用的约定是使用寄存器 `eax` 来传递系统调用号的。至于参数的传递，在 `glibc` 中也有详细的说明，参见文件 `sysdeps/unix/sysv/linux/i386/sysdep.h`。

### 0.4.3 线程安全

线程安全，顾名思义是指代码可以在多线程环境下“安全”地执行。何谓安全？即符合正确的逻辑结果，是程序员期望的正常执行结果。为了实现线程安全，该代码要么只能使用局部变量或资源，要么就是利用锁等同步机制，来实现全局变量或资源的串行访问。

下面是一个经典的多线程不安全代码：

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

static int counter = 0;
#define LOOPS 10000000

static void * thread(void * unused)
{
    int i;

    for (i = 0; i < LOOPS; ++i) {
        ++counter;
    }
    return NULL;
}

int main(void)
{
    pthread_t t1, t2;

    pthread_create(&t1, NULL, thread, NULL);
    pthread_create(&t2, NULL, thread, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Counter is %d by threads\n", counter);
    return 0;
}
```



**注意** 之所以这里的 LOOPS 选了一个比较大的数“10000000”，是为了保证第一个线程不要在第二个线程开始执行前就退出了。大家可以根据自己的运行环境来修改这个数值。

以上代码创建了两个线程，用来实现对同一个全局变量进行自加运算，循环次数为一千万次。下面来看一下运行结果：

```
[fgao@fgao chapter0]#./threads_counter
Counter is 10843915 by threads
```

为什么最后的结果不是期望的 20000000 呢？下面反汇编将来揭开这个秘密——反汇编是理解程序行为的不二利器，因为它更贴近机器语言，也就是说，反汇编更贴近 CPU 运行的真相。

下面对线程函数 thread 进行反汇编，代码如下：

```
080484a4 <thread>:
80484a4:    55                push    %ebp
80484a5:    89 e5             mov     %esp,%ebp
80484a7:    83 ec 10          sub     $0x10,%esp
80484aa:    c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%ebp)
80484b1:    eb 11            jmp     80484c4 <thread+0x20>
80484b3:    a1 94 98 04 08    mov     0x8049894,%eax
80484b8:    83 c0 01          add     $0x1,%eax
80484bb:    a3 94 98 04 08    mov     %eax,0x8049894
80484c0:    83 45 fc 01       addl    $0x1,-0x4(%ebp)
80484c4:    81 7d fc 7f 96 98 00 cmpl    $0x98967f,-0x4(%ebp)
80484cb:    7e e6            jle     80484b3 <thread+0xf>
80484cd:    b8 00 00 00 00    mov     $0x0,%eax
80484d2:    c9               leave   %eax
80484d3:    c3              ret
```

其中加粗部分对应的是 ++counter 的汇编代码，其逻辑如下：

- 1) 将 counter 的值赋给寄存器 EAX；
- 2) 对寄存器 EAX 的值加 1；
- 3) 将 EAX 的值赋给 counter。

假设目前 counter 的值为 0，那么当两个线程同时执行 ++counter 时，会有如下情况（每个线程会有独立的上下文执行环境，所以可视为每个线程都有一个“独立”的 EAX）：

thread1	thread2
eax = counter => eax = 0	
	eax = counter => 0 eax = 0
eax = eax+1 => eax = 1	
counter = eax => counter = 1	eax = eax + 1 => eax = 1
	counter = eax => counter = 1

上面两个线程都对 counter 执行了自增动作，但是最终的结果是“1”而不是“2”。这只是众多错误时序情况中的一种。之所以会产生这样的错误，就是因为 ++counter 的执行指

令并不是原子的，多个线程对 `counter` 的并发访问造成了最后的错误结果。利用锁就可以保证 `counter` 自增指令的串行化，如下所示：

```

thread1                      thread2
lock
eax = counter => eax = 0
eax = eax + 1 => eax = 1
counter = eax => counter = 1
unlock

                                lock
                                eax = counter => eax = 1
                                eax = eax + 1 => eax = 2
                                counter = eax => counter = 2
                                unlock

```

通过加锁，可以视 `counter` 的自增指令为“原子指令”，最后的结果终于是期望的答案了。

#### 0.4.4 原子性

以前原子被认为是物理组成的最小单元，所以在计算机领域，就借其不可分割的这层含义作为隐喻。对于计算机科学来说，如果变量是原子的，那么对这个变量的任何访问和更改都是原子的。如果操作是原子的，那么这个操作将是不可分割的，要么成功，要么失败，不会有任何的中间状态。

列举一个原子操作的例子，用户 A 向用户 B 转账 1000 元。简单来说，这里最起码有两个步骤：

- 1) 用户 A 的账号减少 1000 元；
- 2) 用户 B 的账号增加 1000 元。

如果在上述步骤 1 结束的时候，转账发生了故障，比如电力中断，是否会造成用户 A 的账号减少了 1000 元，而用户 B 的账号没有变化呢？这种情况对于原子操作是不会发生的。当电力中断导致转账操作进行到一半就失败时，用户 A 的账号肯定不会减少 1000 元。因为这个操作的原子性，保证了用户 A 减少 1000 元和用户 B 增加 1000 元，必须同时成立，而不会存在一个中间结果。至于这个操作是如何做到原子性的，可以参看数据库的事务是如何实现的——原子性是事务的一个特性之一。

#### 0.4.5 可重入函数

从字面上理解，可重入就是可重复进入。在编程领域，它不仅仅意味着可以重复进入，还要求在进入后能成功执行。这里的重复进入，是指当前进程已经处于该函数中，这时程序会允许当前进程的某个执行流程再次进入该函数，而不会引发问题。这里的执行流程不仅仅包括多线程，还包括信号处理、`longjump` 等执行流程。所以，可重入函数一定是线程安全的，而线程安全函数则不一定是可重入函数。

从以上定义来看，很难说出哪些函数是可重入函数，但是可以很明显看出哪些函数是不可以重入的函数。当函数使用锁的时候，尤其是互斥锁的时候，该函数是不可重入的，

否则会造成死锁。若函数使用了静态变量，并且其工作依赖于这个静态变量时，该函数也是不可重入的，否则会造成该函数工作不正常。

下面来看一个死锁的例子代码如下：

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

#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static const char * const caller[2] = {"mutex_thread", "signal handler"};
static pthread_t mutex_tid;
static pthread_t sleep_tid;
static volatile int signal_handler_exit = 0;

static void hold_mutex(int c)
{
    printf("enter hold_mutex [caller %s]\n", caller[c]);

    pthread_mutex_lock(&mutex);

    /* 这里的循环是为了保证锁不会在信号处理函数退出前被释放掉 */
    while (!signal_handler_exit && c != 1) {
        sleep(5);
    }

    pthread_mutex_unlock(&mutex);

    printf("leave hold_mutex [caller %s]\n", caller[c]);
}

static void *mutex_thread(void *arg)
{
    hold_mutex(0);
}

static void *sleep_thread(void *arg)
{
    sleep(10);
}

static void signal_handler(int signum)
{
    hold_mutex(1);
    signal_handler_exit = 1;
}

int main()
{

```

```

    signal(SIGUSR1, signal_handler);

    pthread_create(&mutex_tid, NULL, mutex_thread, NULL);
    pthread_create(&sleep_tid, NULL, sleep_thread, NULL);

    pthread_kill(sleep_tid, SIGUSR1);
    pthread_join(mutex_tid, NULL);
    pthread_join(sleep_tid, NULL);

    return 0;
}

```

先看看运行结果：

```

[fgao@fgao chapter0]#gcc -g 0_8_signal_mutex.c -o signal_mutex -lpthread
[fgao@fgao chapter0]#./signal_mutex
enter hold_mutex [caller signal handler]
enter hold_mutex [caller mutex_thread]

```

为什么会死锁呢？就是因为函数 `hold_mutex` 是不可重入的函数——其中使用了 `pthread_mutex` 互斥量。当 `mutex_thread` 获得 `mutex` 时，`sleep_thread` 就收到了信号，再次调用就进入了 `hold_mutex`。结果始终无法拿到 `mutex`，信号处理函数无法返回，正常的程序流程也无法继续，这就造成了死锁。

#### 0.4.6 阻塞与非阻塞

这里的阻塞与非阻塞，都是指 I/O 操作。在 Linux 环境下，所有的 I/O 系统调用默认都是阻塞的。那么何谓阻塞呢？阻塞的系统调用是指，当进行系统调用时，除非出错（被信号打断也视为出错），进程将会一直陷入内核态直到调用完成。非阻塞的系统调用是指无论 I/O 操作成功与否，调用都会立刻返回。

#### 0.4.7 同步与非同步

这里的同步与非同步，也是指 I/O 操作。当把阻塞、非阻塞、同步和非同步放在一起时，不免会让人眼花缭乱。同步是否就是阻塞，非同步是否就是非阻塞呢？实际上在 I/O 操作中，它们是不同的概念。同步既可以是阻塞的，也可以是非阻塞的，而常用的 Linux 的 I/O 调用实际上都是同步的。这里的同步和非同步，是指 I/O 数据的复制工作是否同步执行。

以系统调用 `read` 为例。阻塞的 `read` 会一直陷入内核态直到 `read` 返回；而非阻塞的 `read` 在数据未准备好的情况下，会直接返回错误，而当有数据时，非阻塞的 `read` 同样会一直陷入内核态，直到 `read` 完成。这个 `read` 就是同步的操作，即 I/O 的完成是在当前执行流程下同步完成的。

如果是非同步即异步，则 I/O 操作不是随系统调用同步完成的。调用返回后，I/O 操作并没有完成，而是由操作系统或者某个线程负责真正的 I/O 操作，等完成后通知原来的线程。



# 文件 I/O

文件 I/O 是操作系统不可或缺的部分，也是实现数据持久化的手段。对于 Linux 来说，其“一切皆是文件”的思想，更是突出了文件在 Linux 内核中的重要地位。本章主要讲述 Linux 文件 I/O 部分的系统调用。



**注意** 为了分析系统调用的实现，从本章开始会涉及 Linux 内核源码。但是本书并不是一本介绍内核源码的书籍，所以书中对内核源码的分析不会面面俱到。分析内核源码的目的是为了更好地理解系统调用，是为应用而服务的。因此，本书对内核源码的追踪和分析，只是浅尝辄止。

## 1.1 Linux 中的文件

### 1.1.1 文件、文件描述符和文件表

Linux 内核将一切视为文件，那么 Linux 的文件是什么呢？其既可以是事实上的真正的物理文件，也可以是设备、管道，甚至还可以是一块内存。狭义的文件是指文件系统物理文件，而广义的文件则可以是 Linux 管理的所有对象。这些广义的文件利用 VFS 机制，以文件系统的形式挂载在 Linux 内核中，对外提供一致的文件操作接口。

从数值上看，文件描述符是一个非负整数，其本质就是一个句柄，所以也可以认为文件描述符就是一个文件句柄。那么何为句柄呢？一切对于用户透明的返回值，即可视为句柄。用户空间利用文件描述符与内核进行交互；而内核拿到文件描述符后，可以通过它得到用于管理文件的真正的数据结构。

使用文件描述符即句柄，有两个好处：一是增加了安全性，句柄类型对用户完全透明，



用户无法通过任何 hacking 的方式，更改句柄对应的内部结果，比如 Linux 内核的文件描述符，只有内核才能通过该值得到对应的文件结构；二是增加了可扩展性，用户的代码只依赖于句柄的值，这样实际结构的类型就可以随时发生变化，与句柄的映射关系也可以随时改变，这些变化都不会影响任何现有的用户代码。

Linux 的每个进程都会维护一个文件表，以便维护该进程打开文件的信息，包括打开的文件个数、每个打开文件的偏移量等信息。

### 1.1.2 内核文件表的实现

内核中进程对应的结构是 `task_struct`，进程的文件表保存在 `task_struct->files` 中。其结构代码如下所示。

```
struct files_struct {
    /* count为文件表files_struct的引用计数 */
    atomic_t count;
    /* 文件描述符表 */
    /*
     * 为什么有两个fdtable呢？这是内核的一种优化策略。fdt为指针，而fdtab为普通变量。一般情况下，
     * fdt是指向fdtab的，当需要它的时候，才会真正动态申请内存。因为默认大小的文件表足以应付大多数
     * 情况，因此这样就可以避免频繁的内存申请。
     * 这也是内核的常用技巧之一。在创建时，使用普通的变量或者数组，然后让指针指向它，作为默认情况使用。
     * 只有当进程使用量超过默认值时，才会动态申请内存。
     */
    struct fdtable __rcu *fdt;
    struct fdtable fdtab;
    /*
     * written part on a separate cache line in SMP
     */
    /* 使用__cacheline_aligned_in_smp可以保证file_lock是以cache
     * line 对齐的，避免了false sharing */
    spinlock_t file_lock __cacheline_aligned_in_smp;
    /* 用于查找下一个空闲的fd */
    int next_fd;
    /* 保存执行exec需要关闭的文件描述符的位图 */
    struct embedded_fd_set close_on_exec_init;
    /* 保存打开的文件描述符的位图 */
    struct embedded_fd_set open_fds_init;
    /* fd_array为一个固定大小的file结构数组。struct file是内核用于文
     * 件管理的结构。这里使用默认大小的数组，就是为了可以涵盖大多数情况，避免动
     * 态分配 */
    struct file __rcu * fd_array[NR_OPEN_DEFAULT];
};
```

下面看看 `files_struct` 是如何使用默认的 `fdtab` 和 `fd_array` 的，`init` 是 Linux 的第一个进程，它的文件表是一个全局变量，代码如下：

```
struct files_struct init_files = {
    .count      = ATOMIC_INIT(1),
    .fdt        = &init_files.fdtab,
    .fdtab      = {
```

```
        .max_fds      = NR_OPEN_DEFAULT,
        .fd           = &init_files.fd_array[0],
        .close_on_exec = (fd_set *)&init_files.close_on_exec_init,
        .open_fds     = (fd_set *)&init_files.open_fds_init,
    },
    .file_lock = __SPIN_LOCK_UNLOCKED(init_task.file_lock),
};
```


init\_files.fdt 和 init\_files.fdtab.fd 都分别指向了自己已有的成员变量，并以此作为一个默认值。后面的进程都是从 init 进程 fork 出来的。fork 的时候会调用 dup\_fd，而在 dup\_fd 中其代码结构如下：

```
newf = kmem_cache_alloc(files_cachep, GFP_KERNEL);
if (!newf)
    goto out;

atomic_set(&newf->count, 1);

spin_lock_init(&newf->file_lock);
newf->next_fd = 0;
new_fdt = &newf->fdtab;
new_fdt->max_fds = NR_OPEN_DEFAULT;
new_fdt->close_on_exec = (fd_set *)&newf->close_on_exec_init;
new_fdt->open_fds = (fd_set *)&newf->open_fds_init;
new_fdt->fd = &newf->fd_array[0];
new_fdt->next = NULL;
```

初始化 new\_fdt，同样是为了让 new\_fdt 和 new\_fdt->fd 指向其本身的成员变量 fdtab 和 fd\_array。

 **说明** /proc/pid/status 为对应 pid 的进程的当前运行状态，其中 FDSize 值即为当前进程 max\_fds 的值。

因此，初始状态下，files\_struct、fdtable 和 files 的关系如图 1-1 所示。

## 1.2 打开文件

### 1.2.1 open 介绍

open 在手册中有两个函数原型，如下所示：

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags,
        mode_t mode);
```

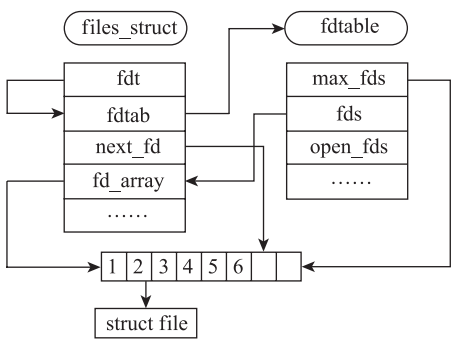


图 1-1 文件表、文件描述符表及文件结构关系图

这样的函数原型有些违背了我们的直觉。C 语言是不支持函数重载的，为什么 open 的系统调用可以有两个这样的 open 原型呢？内核绝对不可能为这个功能创建两个系统调用。

在 Linux 内核中，实际上只提供了一个系统调用，对应的是上述两个函数原型中的第二个。那么 `open` 有两个函数原型又是怎么回事呢？当我们调用 `open` 函数时，实际上调用的是 `glibc` 封装的函数，然后由 `glibc` 通过自陷指令，进行真正的系统调用。也就是说，所有的系统调用都要先经过 `glibc` 才会进入操作系统。这样的话，实际上是 `glibc` 提供了一个变参函数 `open` 来满足两个函数原型，然后通过 `glibc` 的变参函数 `open` 实现真正的系统调用来调用原型二。

可以通过一个小程序来验证我们的猜想，代码如下：

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

int main(void)
{
    int fd = open("test_open.txt", O_CREAT, 0644, "test");

    close(fd);
    return 0;
}
```

在这个程序中，调用 `open` 的时候，传入了 4 个参数，如果 `open` 不是变参函数，就会报错，如 “too many arguments to function ‘open’”。但是请看下面的编译输出：

```
[fgao@fgao-ThinkPad-R52 chapter2]#gcc -g -Wall 2_2_1_test_open.c
[fgao@fgao-ThinkPad-R52 chapter2]#
```

没有任何的警告和错误。这就证实了我们的猜想，`open` 是 `glibc` 的一个变参函数。`fcntl.h` 中 `open` 函数的声明也确定了这点：

```
extern int open (__const char *__file, int __oflag, ...) __nonnull ((1));
```

下面来说明一下 `open` 的参数：

- ❑ **pathname**：表示要打开的文件路径。
- ❑ **flags**：用于指示打开文件的选项，常用的有 `O_RDONLY`、`O_WRONLY` 和 `O_RDWR`。这三个选项必须有且只能有一个被指定。为什么 `O_RDWR != O_RDONLY | O_WRONLY` 呢？Linux 环境中，`O_RDONLY` 被定义为 0，`O_WRONLY` 被定义为 1，而 `O_RDWR` 却被定义为 2。之所以有这样违反常规的设计遗留至今，就是为了兼容以前的程序。除了以上三个选项，Linux 平台还支持更多的选项，APUE 中对此也进行了介绍。
- ❑ **mode**：只在创建文件时需要，用于指定所创建文件的权限位（还要受到 `umask` 环境变量的影响）。

### 1.2.2 更多选项

除了常用的几个打开文件的选项，APUE 还介绍了一些常用的 POSIX 定义的选项。下

面列出了 Linux 平台支持的大部分选项：

- ❑ `O_APPEND`：每次进行写操作时，内核都会先定位到文件尾，再执行写操作。
- ❑ `O_ASYNC`：使用异步 I/O 模式。
- ❑ `O_CLOEXEC`：在打开文件的时候，就为文件描述符设置 `FD_CLOEXEC` 标志。这是一个新的选项，用于解决在多线程下 `fork` 与用 `fcntl` 设置 `FD_CLOEXEC` 的竞争问题。某些应用使用 `fork` 来执行第三方的业务，为了避免泄露已打开文件的内容，那些文件会设置 `FD_CLOEXEC` 标志。但是 `fork` 与 `fcntl` 是两次调用，在多线程下，可能会在 `fcntl` 调用前，就已经 `fork` 出子进程了，从而导致该文件句柄暴露给子进程。关于 `O_CLOEXEC` 的用途，将在第 4 章详细讲解。
- ❑ `O_CREAT`：当文件不存在时，就创建文件。
- ❑ `O_DIRECT`：对该文件进行直接 I/O，不使用 VFS Cache。
- ❑ `O_DIRECTORY`：要求打开的路径必须是目录。
- ❑ `O_EXCL`：该标志用于确保是此次调用创建的文件，需要与 `O_CREAT` 同时使用；当文件已经存在时，`open` 函数会返回失败。
- ❑ `O_LARGEFILE`：表明文件为大文件。
- ❑ `O_NOATIME`：读取文件时，不更新文件最后的访问时间。
- ❑ `O_NONBLOCK`、`O_NDELAY`：将该文件描述符设置为非阻塞的（默认都是阻塞的）。
- ❑ `O_SYNC`：设置为 I/O 同步模式，每次进行写操作时都会将数据同步到磁盘，然后 `write` 才能返回。
- ❑ `O_TRUNC`：在打开文件的时候，将文件长度截断为 0，需要与 `O_RDWR` 或 `O_WRONLY` 同时使用。在写文件时，如果是作为新文件重新写入，一定要使用 `O_TRUNC` 标志，否则可能会造成旧内容依然存在于文件中的错误，如生成配置文件、`pid` 文件等——在第 2 章中，我会例举一个未使用截断标志而导致问题的示例代码。



**注意** 并不是所有的文件系统都支持以上选项。

### 1.2.3 open 源码跟踪

我们经常这样描述“打开一个文件”，那么这个所谓的“打开”，究竟“打开”了什么？内核在这个过程中，又做了哪些事情呢？这一切将通过分析内核源码来得到答案。

跟踪内核 `open` 源码 `open->do_sys_open`，代码如下：

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode)
{
    struct open_flags op;
    /* flags为用户层传递的参数，内核会对flags进行合法性检查，并根据mode生成新的flags值赋给
       lookup */
    int lookup = build_open_flags(flags, mode, &op);
    /* 将用户空间的文件名参数复制到内核空间 */
    char *tmp = getname(filename);
```

```

int fd = PTR_ERR(tmp);

if (!IS_ERR(tmp)) {
    /* 未出错则申请新的文件描述符 */
    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        /* 申请新的文件管理结构file */
        struct file *f = do_filp_open(dfd, tmp, &op, lookup);
        if (IS_ERR(f)) {
            put_unused_fd(fd);
            fd = PTR_ERR(f);
        } else {
            /* 产生文件打开的通知事件 */
            fsnotify_open(f);
            /* 将文件描述符fd与文件管理结构file对应起来, 即安装 */
            fd_install(fd, f);
        }
    }
    putname(tmp);
}
return fd;
}

```

从 `do_sys_open` 可以看出, 打开文件时, 内核主要消耗了两种资源: 文件描述符与内核管理文件结构 `file`。

#### 1.2.4 如何选择文件描述符

根据 POSIX 标准, 当获取一个新的文件描述符时, 要返回最低的未使用的文件描述符。Linux 是如何实现这一标准的呢?

在 Linux 中, 通过 `do_sys_open->get_unused_fd_flags->alloc_fd(0, (flags))` 来选择文件描述符, 代码如下:

```

int alloc_fd(unsigned start, unsigned flags)
{
    struct files_struct *files = current->files;
    unsigned int fd;
    int error;
    struct fdtable *fdt;
    /* files为进程的文件表, 下面需要更改文件表, 所以需要先锁文件表 */
    spin_lock(&files->file_lock);
repeat:
    /* 得到文件描述符表 */
    fdt = files_fdtable(files);
    /* 从start开始, 查找未用的文件描述符。在打开文件时, start为0 */
    fd = start;
    /* files->next_fd为上一次成功找到的fd的下一个描述符。使用next_fd, 可以快速找到未用的文件描述符; */
    if (fd < files->next_fd)
        fd = files->next_fd;

    /*

```

当小于当前文件表支持的最大文件描述符个数时，利用位图找到未用的文件描述符。如果大于max\_fds怎么办呢？如果大于当前支持的最大文件描述符，那它肯定是未用的，就不需要用位图来确认了。

```

*/
if (fd < fdt->max_fds)
    fd = find_next_zero_bit(fdt->open_fds->fds_bits,
        fdt->max_fds, fd);
/* expand_files用于在必要时扩展文件表。何时是必要的时候呢？比如当前文件描述符已经超过了当前文件表支持的最大值的时候。 */
error = expand_files(files, fd);
if (error < 0)
    goto out;

/*
 * If we needed to expand the fs array we
 * might have blocked - try again.
 */
if (error)
    goto repeat;

/* 只有在start小于next_fd时，才需要更新next_fd，以尽量保证文件描述符的连续性。*/
if (start <= files->next_fd)
    files->next_fd = fd + 1;

/* 将打开文件位图open_fds对应fd的位置置位 */
FD_SET(fd, fdt->open_fds);
/* 根据flags是否设置了O_CLOEXEC，设置或清除fdt->close_on_exec */
if (flags & O_CLOEXEC)
    FD_SET(fd, fdt->close_on_exec);
else
    FD_CLR(fd, fdt->close_on_exec);
error = fd;
#endif 1
/* Sanity check */
if (rcu_dereference_raw(fdt->fd[fd]) != NULL) {
    printk(KERN_WARNING "alloc_fd: slot %d not NULL!\n", fd);
    rcu_assign_pointer(fdt->fd[fd], NULL);
}
#endif

out:
    spin_unlock(&files->file_lock);
    return error;
}

```

### 1.2.5 文件描述符 fd 与文件管理结构 file

前文已经说过，内核使用 fd\_install 将文件管理结构 file 与 fd 组合起来，具体操作请看如下代码：

```

void fd_install(unsigned int fd, struct file *file)
{
    struct files_struct *files = current->files;

```

```

struct fdtable *fdt;
spin_lock(&files->file_lock);
/* 得到文件描述符表 */
fdt = files_fdt(files);
BUG_ON(fdt->fd[fd] != NULL);
/*
将文件描述符表中的file类型的指针数组中对应fd的项指向file。
这样文件描述符fd与file就建立了对应关系
*/
rcu_assign_pointer(fdt->fd[fd], file);
spin_unlock(&files->file_lock);
}

```

当用户使用 `fd` 与内核交互时，内核可以用 `fd` 从 `fdt->fd[fd]` 中得到内部管理文件的结构 `struct file`。

## 1.3 creat 简介

`creat` 函数用于创建一个新文件，其等价于 `open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)`。APUE 介绍了引入 `creat` 的原因：

由于历史原因，早期的 Unix 版本中，`open` 的第二个参数只能是 0、1 或者 2。这样就没有办法打开一个不存在的文件。因此，一个独立系统调用 `creat` 被引入，用于创建新文件。现在的 `open` 函数，通过使用 `O_CREAT` 和 `O_TRUNC` 选项，可以实现 `creat` 的功能，因此 `creat` 已经不是必要的了。

内核 `creat` 的实现代码如下所示：

```

SYSCALL_DEFINE2(creat, const char __user *, pathname, int, mode)
{
    return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
}

```

这样就确定了 `creat` 无非是 `open` 的一种封装实现。

## 1.4 关闭文件

### 1.4.1 close 介绍

`close` 用于关闭文件描述符。而文件描述符可以是普通文件，也可以是设备，还可以是 `socket`。在关闭时，VFS 会根据不同的文件类型，执行不同的操作。

下面将通过跟踪 `close` 的内核源码来了解内核如何针对不同的文件类型执行不同的操作。

### 1.4.2 close 源码跟踪

首先，来看一下 `close` 的源码实现，代码如下：

```

SYSCALL_DEFINE1(close, unsigned int, fd)
{
    struct file * filp;
    /* 得到当前进程的文件表 */
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    int retval;

    spin_lock(&files->file_lock);
    /* 通过文件表, 取得文件描述符表 */
    fdt = files_fdt(files);
    /* 参数fd大于文件描述符表记录的最大描述符, 那么它一定是非法的描述符 */
    if (fd >= fdt->max_fds)
        goto out_unlock;
    /* 利用fd作为索引, 得到file结构指针 */
    filp = fdt->fd[fd];
    /*
    检查filp是否为NULL。正常情况下, filp一定不为NULL。
    */
    if (!filp)
        goto out_unlock;
    /* 将对应的filp置为0 */
    rcu_assign_pointer(fdt->fd[fd], NULL);
    /* 清除fd在close_on_exec位图中的位 */
    FD_CLR(fd, fdt->close_on_exec);
    /* 释放该fd, 或者说将其置为unused。 */
    __put_unused_fd(files, fd);
    spin_unlock(&files->file_lock);
    /* 关闭file结构 */
    retval = filp_close(filp, files);
    /* can't restart close syscall because file table entry was cleared */
    if (unlikely(retval == -ERESTARTSYS ||
                retval == -ERESTARTNOINTR ||
                retval == -ERESTARTNOHAND ||
                retval == -ERESTART_RESTARTBLOCK))
        retval = -EINTR;

    return retval;

out_unlock:
    spin_unlock(&files->file_lock);
    return -EBADF;
}
EXPORT_SYMBOL(sys_close);

```

\_\_put\_unused\_fd 源码如下所示:

```

static void __put_unused_fd(struct files_struct *files, unsigned int fd)
{
    /* 取得文件描述符表 */
    struct fdtable *fdt = files_fdt(files);
    /* 清除fd在open_fds位图的位 */
    FD_CLR(fd, fdt->open_fds);
    /* 如果fd小于next_fd, 重置next_fd为释放的fd */
}

```



```

    if (fd < files->next_fd)
        files->next_fd = fd;
}

```

看到这里，我们来回顾一下之前分析过的 `alloc_fd` 函数，就可以总结出完整的 Linux 文件描述符选择策略：

- ❑ Linux 选择文件描述符是按从小到大的顺序进行寻找的，文件表中 `next_fd` 用于记录下一次开始寻找的起点。当有空闲的描述符时，即可分配。
- ❑ 当某个文件描述符关闭时，如果其小于 `next_fd`，则 `next_fd` 就重置为这个描述符，这样下一次分配就会立刻重用这个文件描述符。

以上的策略，总结成一句话就是“Linux 文件描述符策略永远选择最小的可用的文件描述符”。——这也是 POSIX 标准规定的。

其实我并不喜欢这样的策略。因为这样迅速地重用刚刚释放的文件描述符，容易引发难以调试和定位的 bug——尽管这样的 bug 是应用层造成的。比如一个线程关闭了某个文件描述符，然后又创建了一个新的文件描述符，这时文件描述符就被重用了，但其值是一样的。如果有另外一个线程保存了之前的文件描述符的值，那它就会再次访问这个文件描述符。此时，如果是普通文件，就会读错或写错文件。如果是 socket，就会与错误的对端通信。这样的错误发生时，可能并不会被察觉到。即使发现了错误，要找到根本原因，也非常困难。

如果不重用这个描述符呢？在文件描述符被关闭后，创建新的描述符且不使用相同的值。这样再次访问之前的描述符时，内核可以返回错误，应用层可以更早地得知错误的发生。

虽然造成这样错误的原因是应用层自己，但是如果内核可以尽早地让错误发生，对于应用开发人员来说，会是一个福音。因为调试 bug 的时候，bug 距离造成错误的地点越近，时间发生得越早，就越容易找到根本原因。这也是为什么释放内存以后，要将指针置为 NULL 的原因。

从 `__put_unused_fd` 退出后，`close` 会接着调用 `filp_close`，其调用路径为 `filp_close->fput`。在 `fput` 中，会对当前文件 struct `file` 的引用计数减一并检查其值是否为 0。当引用计数为 0 时，表示该 struct `file` 没有被其他人使用，则可以调用 `__fput` 执行真正的文件释放操作，然后调用要关闭文件所属文件系统的 `release` 函数，从而实现针对不同的文件类型来执行不同的关闭操作。

下一节让我们来看看 Linux 如何针对不同的文件类型，挂载不同的文件操作函数 `files_operations`。

### 1.4.3 自定义 `files_operations`

不失一般性，这里也选择 socket 文件系统作为示例，来说明 Linux 如何挂载文件系统指定的文件操作函数 `files_operations`。

`socket.c` 中定义了其文件操作函数 `file_operations`，代码如下：

```
static const struct file_operations socket_file_ops = {
    .owner =      THIS_MODULE,
    .llseek =     no_llseek,
    .aio_read =   sock_aio_read,
    .aio_write =  sock_aio_write,
    .poll =       sock_poll,
    .unlocked_ioctl = sock_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = compat_sock_ioctl,
#endif
    .mmap =       sock_mmap,
    .open =       sock_no_open, /* special open code to disallow open via /proc */
    .release =    sock_close,
    .fsync =      sock_fsync,
    .sendpage =   sock_sendpage,
    .splice_write = generic_splice_sendpage,
    .splice_read = sock_splice_read,
};
```

函数 `sock_alloc_file` 用于申请 `socket` 文件描述符及文件管理结构 `file` 结构。它调用 `alloc_file` 来申请管理结构 `file`，并将 `socket_file_ops` 作为参数，如下所示：

```
file = alloc_file(&path, FMODE_READ | FMODE_WRITE,
    &socket_file_ops);
```

进入 `alloc_file`，来看看如下代码：

```
struct file *alloc_file(struct path *path, fmode_t mode,
    const struct file_operations *fop)
{
    struct file *file;

    /* 申请一个file */
    file = get_empty_filp();
    if (!file)
        return NULL;

    file->f_path = *path;
    file->f_mapping = path->dentry->d_inode->i_mapping;
    file->f_mode = mode;
    /* 将自定义的文件操作函数赋给file->f_op */
    file->f_op = fop;

    .....
}
```

在初始化 `file` 结构的时候，`socket` 文件系统将其自定义的文件操作赋给了 `file->f_op`，从而实现了在 VFS 中可以调用 `socket` 文件系统自定义的操作。

#### 1.4.4 遗忘 close 造成的问题

我们只需要关注 `close` 文件的时候内核做了哪些事情，就可以确定遗忘 `close` 会带来什么样的后果，如下：

- ❑ 文件描述符始终没有被释放。
- ❑ 用于文件管理的某些内存结构没有被释放。

对于普通进程来说，即使应用忘记了关闭文件，当进程退出时，Linux 内核也会自动关闭文件，释放内存（详细过程见后文）。但是对于一个常驻进程来说，问题就变得严重了。

先看第一种情况，如果文件描述符没有被释放，那么再次申请新的描述符时，就不得不扩展当前的文件表了，代码如下：

```
int expand_files(struct files_struct *files, int nr)
{
    struct fdtable *fdt;

    fdt = files_fdtable(files);

    /*
     * N.B. For clone tasks sharing a files structure, this test
     * will limit the total number of files that can be opened.
     */
    if (nr >= rlimit(RLIMIT_NOFILE))
        return -EMFILE;

    /* Do we need to expand? */
    if (nr < fdt->max_fds)
        return 0;

    /* Can we expand? */
    if (nr >= sysctl_nr_open)
        return -EMFILE;

    /* All good, so we try */
    return expand_fdtable(files, nr);
}
```

从上面的代码可以看出，在扩展文件表的时候，会检查打开文件的个数是否超出系统的限制。如果文件描述符始终不释放，其个数迟早会到达上限，并返回 EMFILE 错误（表示 Too many open files (POSIX.1)）。

再看第二种情况，即文件管理的某些内存结构没有被释放。仍然是查看打开文件的代码，代码如下其中，`get_empty_filp` 用于获得空闲的 `file` 结构。

```
struct file *get_empty_filp(void)
{
    const struct cred *cred = current_cred();
    static long old_max;
    struct file *f;

    /*
     * Privileged users can go above max_files
     */
    /* 这里对打开文件的个数进行检查，非特权用户不能超过系统的限制 */
    if (get_nr_files() >= files_stat.max_files && !capable(CAP_SYS_ADMIN)) {
        /*
```

```

再次检查per cpu的文件个数的总和， 为什么要做两次检查呢。后文会详细介绍 */
if (percpu_counter_sum_positive(&nr_files) >= files_stat.max_files)
    goto over;
}

/* 未到达上限，申请一个新的file结构 */
f = kmem_cache_zalloc(filp_cachep, GFP_KERNEL);
if (f == NULL)
    goto fail;

/* 增加file结构计数 */
percpu_counter_inc(&nr_files);
f->f_cred = get_cred(cred);
if (security_file_alloc(f))
    goto fail_sec;

INIT_LIST_HEAD(&f->f_u.fu_list);
atomic_long_set(&f->f_count, 1);
rwlock_init(&f->f_owner.lock);
spin_lock_init(&f->f_lock);
eventpoll_init_file(f);
/* f->f_version: 0 */
return f;

over:
/* 用完了file配额，打印log报错 */
/* Ran out of filps - report that */
if (get_nr_files() > old_max) {
    pr_info("VFS: file-max limit %lu reached\n", get_max_files());
    old_max = get_nr_files();
}
goto fail;

fail_sec:
    file_free(f);
fail:
    return NULL;
}

```

下面来说说为什么上面的代码要做两次检查——这也是我们学习内核代码的好处之一，可以学到很多的编程技巧和设计思路。

对于 file 的个数，Linux 内核使用两种方式来计数。一是使用全局变量，另外一个使用 per cpu 变量。更新全局变量时，为了避免竞争，不得不使用锁，所以 Linux 使用了一种折中的解决方案。当 per cpu 变量的个数变化不超过正负 percpu\_counter\_batch（默认为 32）的范围时，就不更新全局变量。这样就减少了对全局变量的更新，可是也造成了全局变量的值不准确的问题。于是在全局变量的 file 个数超过限制时，会再对所有的 per cpu 变量求和，再次与系统的限制相比较。想了解这个计数手段的详细信息，可以阅读 percpu\_counter\_add 的相关代码。

### 1.4.5 如何查找文件资源泄漏

在前面的小节中，我们看到了常驻进程忘记关闭文件的危害。可是，软件不可能不出现 bug，如果常驻进程程序真的出现了这样的问题，如何才能快速找到根本原因呢？通过审查打开文件的代码？时间长效率低。那是否还有其他办法呢？下面我们来介绍一种能快速查找文件资源泄漏的方法。

首先，创建一个“错误”的程序，代码如下：

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

int main(void)
{
    int cnt = 0;

    while (1) {
        char name[64];

        snprintf(name, sizeof(name), "%d.txt", cnt);

        int fd = creat(name, 644);

        sleep(10);
        ++cnt;
    }

    return 0;
}
```

在这段代码的循环过程中，打开了一个文件，但是一直没有被关闭，以此来模拟服务程序的文件资源泄漏，然后让程序运行一段时间：

```
[fgao@fgao chapter1]#./hold_file &
[1] 3000
```

接下来请出利器 lsof，查看相关信息，如下所示：

```
[fgao@fgao chapter1]#lsof -p 3000
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
a.out    3000 fgao   cwd   DIR   253,2    4096 1321995 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1
a.out    3000 fgao   rtd   DIR   253,1    4096      2 /
a.out    3000 fgao   txt   REG   253,2   6115 1308841 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/a.out
a.out    3000 fgao   mem   REG   253,1  157200 1443950 /lib/ld-2.14.90.so
a.out    3000 fgao   mem   REG   253,1 2012656 1443951 /lib/libc-2.14.90.so
a.out    3000 fgao    0u   CHR  136,3      0t0      6 /dev/pts/3
a.out    3000 fgao    1u   CHR  136,3      0t0      6 /dev/pts/3
a.out    3000 fgao    2u   CHR  136,3      0t0      6 /dev/pts/3
a.out    3000 fgao    3w   REG   253,2      0 1309088 /home/fgao/works/my_git_codes/my_books/understanding_apue/sample_codes/chapter1/0.txt
```

```

a.out  3000 fgao  4w  REG  253,2      0 1312921 /home/fgao/works/my_git_
        codes/my_books/understanding_apue/sample_codes/chapter1/1.txt
a.out  3000 fgao  5w  REG  253,2      0 1327890 /home/fgao/works/my_git_
        codes/my_books/understanding_apue/sample_codes/chapter1/2.txt
a.out  3000 fgao  6w  REG  253,2      0 1327891 /home/fgao/works/my_git_
        codes/my_books/understanding_apue/sample_codes/chapter1/3.txt
a.out  3000 fgao  7w  REG  253,2      0 1327892 /home/fgao/works/my_git_
        codes/my_books/understanding_apue/sample_codes/chapter1/4.txt
a.out  3000 fgao  8w  REG  253,2      0 1327893 /home/fgao/works/my_git_
        codes/my_books/understanding_apue/sample_codes/chapter1/5.txt
a.out  3000 fgao  9w  REG  253,2      0 1327894 /home/fgao/works/my_git_
        codes/my_books/understanding_apue/sample_codes/chapter1/6.txt

```

从 lsof 的输出结果可以清晰地看出，hold\_file 打开的哪些文件没有被关闭。其实从 /proc/3000/fd 中也可以得到类似的结果。但是 lsof 拥有更多的选项和功能（如指定某个目录），可以应对更复杂的情况。具体细节就需要读者自行阅读 lsof 的说明文档了。

## 1.5 文件偏移

文件偏移是基于某个打开文件来说的，一般情况下，读写操作都会从当前的偏移位置开始读写（所以 read 和 write 都没有显式地传入偏移量），并且在读写结束后更新偏移量。

### 1.5.1 lseek 简介

lseek 的原型如下：

```
off_t lseek(int fd, off_t offset, int whence);
```

该函数用于将 fd 的文件偏移量设置为以 whence 为起点，偏移为 offset 的位置。其中 whence 可以为三个值：SEEK\_SET、SEEK\_CUR 和 SEEK\_END，分别表示为“文件的起始位置”、“文件的当前位置”和“文件的末尾”，而 offset 的取值正负均可。lseek 执行成功后，会返回新的文件偏移量。

在 Linux 3.1 以后，Linux 又增加了两个新的值：SEEK\_DATA 和 SEEK\_HOLE，分别用于查找文件中的数据和空洞。

### 1.5.2 小心 lseek 的返回值

对于 Linux 中的大部分系统调用来说，如果返回值是负数，那它一般都是错误的，但是对于 lseek 来说这条规则不适用。且看 lseek 的返回值说明：



当 lseek 执行成功时，它会返回最终以文件起始位置为起点的偏移位置。如果出错，则返回 -1，同时 errno 被设置为对应的错误值。

也就是说，一般情况下，对于普通文件来说，lseek 都是返回非负的整数，但是对于某些设备文件来说，是允许返回负的偏移量。因此要想判断 lseek 是否真正出错，必须在调用

lseek 前将 `errno` 重置为 0，然后再调用 `lseek`，同时检查返回值是否为 -1 及 `errno` 的值。只有当两个同时成立时，才表明 `lseek` 真正出错了。

因为这里的文件偏移都是内核的概念，所以 `lseek` 并不会引起任何真正的 I/O 操作。

### 1.5.3 lseek 源码分析

`lseek` 的源码位于 `read_write.c` 中，如下：

```
SYSCALL_DEFINE3(lseek, unsigned int, fd, off_t, offset, unsigned int, origin)
{
    off_t retval;
    struct file * file;
    int fput_needed;

    retval = -EBADF;
    /* 根据fd得到file指针 */
    file = fget_light(fd, &fput_needed);
    if (!file)
        goto bad;
    retval = -EINVAL;
    /* 对初始位置进行检查，目前linux内核支持的初始位置有1.5.1节中提到的五个值 */
    if (origin <= SEEK_MAX) {
        loff_t res = vfs_llseek(file, offset, origin);
        /* 下面这段代码，先使用res来给retval赋值，然后再次判断res
           是否与retval相等。为什么会有这样的逻辑呢？什么时候两者会不相等呢？
           只有在retval与res的位数不相等的情况下。
           retval的类型是off_t->__kernel_off_t->long;
           而res的类型是loff_t->__kernel_off_t->long long;
           在32位机上，前者是32位，而后者是64位。当res的值超过了retval
           的范围时，两者将会不等。即实际偏移量超过了long类型的表示范围。
           */
        retval = res;
        if (res != (loff_t)retval)
            retval = -EOVERFLOW; /* LFS: should only happen on 32 bit
                                   platforms */
    }
    fput_light(file, fput_needed);
bad:
    return retval;
}
```

然后进入 `vfs_llseek`，代码如下：

```
loff_t vfs_llseek(struct file *file, loff_t offset, int origin)
{
    loff_t (*fn)(struct file *, loff_t, int);

    /* 默认的lseek操作是no_llseek，当file没有对应的llseek实现时，就
       会调用no_llseek，并返回-ESPIPE错误*/
    fn = no_llseek;
    if (file->f_mode & FMODE_LSEEK) {
        if (file->f_op && file->f_op->llseek)
            fn = file->f_op->llseek;
    }
```

```

    }
    return fn(file, offset, origin);
}

```

当 file 支持 llseek 操作时，就会调用具体的 llseek 函数。在此，选择 default\_llseek 作为实例，代码如下：

```

loff_t default_llseek(struct file *file, loff_t offset, int origin)
{
    struct inode *inode = file->f_path.dentry->d_inode;
    loff_t retval;

    mutex_lock(&inode->i_mutex);
    switch (origin) {
        case SEEK_END:
            /* 最终偏移等于文件的大小加上指定的偏移量 */
            offset += i_size_read(inode);
            break;
        case SEEK_CUR:
            /* offset为0时，并不改变当前的偏移量，而是直接返回当前偏移量 */
            if (offset == 0) {
                retval = file->f_pos;
                goto out;
            }
            /* 若offset不为0，则最终偏移等于指定偏移加上当前偏移 */
            offset += file->f_pos;
            break;
        case SEEK_DATA:
            /*
             * In the generic case the entire file is data, so as
             * long as offset isn't at the end of the file then the
             * offset is data.
             */
            /* 如注释所言，对于一般文件，只要指定偏移不超过文件大小，那么指定偏移的位置就是数据位置 */
            if (offset >= inode->i_size) {
                retval = -ENXIO;
                goto out;
            }
            break;
        case SEEK_HOLE:
            /*
             * There is a virtual hole at the end of the file, so
             * as long as offset isn't i_size or larger, return
             * i_size.
             */
            /* 只要指定偏移不超过文件大小，那么下一个空洞位置就是文件的末尾 */
            if (offset >= inode->i_size) {
                retval = -ENXIO;
                goto out;
            }
            offset = inode->i_size;
            break;
    }
}

```



```

    retval = -EINVAL;
/* 对于一般文件来说，最终的offset必须大于或等于0，或者该文件的模式要求只能产生无符号的偏移
   量。否则就会报错 */
if (offset >= 0 || unsigned_offsets(file)) {
    /* 当最终偏移不等于当前位置时，则更新文件的当前位置 */
    if (offset != file->f_pos) {
        file->f_pos = offset;
        file->f_version = 0;
    }
    retval = offset;
}
out:
    mutex_unlock(&inode->i_mutex);
    return retval;
}

```

## 1.6 读取文件

Linux 中读取文件操作时，最常用的就是 read 函数，其原型如下：

```
ssize_t read(int fd, void *buf, size_t count);
```

read 尝试从 fd 中读取 count 个字节到 buf 中，并返回成功读取的字节数，同时将文件偏移向前移动相同的字节数。返回 0 的时候则表示已经到了“文件尾”。read 还有可能读取比 count 小的字节数。

使用 read 进行数据读取时，要注意正确地处理错误，也就是说 read 返回 -1 时，如果 errno 为 EAGAIN、EWOULDBLOCK 或 EINTR，一般情况下都不能将其视为错误。因为前两者是由于当前 fd 为非阻塞且没有可读数据时返回的，后者是由于 read 被信号中断所造成的。这两种情况基本上都可以视为正常情况。

### 1.6.1 read 源码跟踪

先来看看 read 的源码，代码如下：

```

SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    /* 通过文件描述符fd得到管理结构file */
    file = fget_light(fd, &fput_needed);
    if (file) {
        /* 得到文件的当前偏移量 */
        loff_t pos = file_pos_read(file);
        /* 利用vfs进行真正的read */
        ret = vfs_read(file, buf, count, &pos);
        /* 更新文件偏移量 */
        file_pos_write(file, pos);
    }
}

```

```

        /* 归还管理结构file, 如有必要, 就进行引用计数操作*/
        fput_light(file, fput_needed);
    }

    return ret;
}

```

再进入 `vfs_read`, 代码如下:

```

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;

    /* 检查文件是否为读取打开 */
    if (!(file->f_mode & FMODE_READ))
        return -EBADF;
    /* 检查文件是否支持读取操作 */
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    /* 检查用户传递的参数buf的地址是否可写 */
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;

    /* 检查要读取的文件范围实际可读取的字节数 */
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        /* 根据上面的结构, 调整要读取的字节数 */
        count = ret;
        /*
         * 如果定义read操作, 则执行定义的read操作
         * 如果没有定义read操作, 则调用do_sync_read——其利用异步aio_read来完成同步的read操作。
         */
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            /* 读取了一定的字节数, 进行通知操作 */
            fsnotify_access(file);
            /* 增加进程读取字节的统计计数 */
            add_rchar(current, ret);
        }
        /* 增加进程系统调用的统计计数 */
        inc_syscr(current);
    }

    return ret;
}

```

上面的代码为 `read` 公共部分的源码分析, 具体的读取动作是由实际的文件系统决定的。

## 1.6.2 部分读取

前文中介绍 `read` 可以返回比指定 `count` 少的字节数, 那么什么时候会发生这种情况

呢？最直接的想法是在 fd 中没有指定 count 大小的数据时。但这种情况下，系统是不是也可以阻塞到满足 count 个字节的数据呢？那么内核到底采取的是哪种策略呢？

让我们来看看 socket 文件系统中 UDP 协议的 read 实现：socket 文件系统只定义了 aio\_read 操作，没有定义普通的 read 函数。根据前文，在这种情况下 do\_sync\_read 会利用 aio\_read 实现同步读操作。

其调用链为 sock\_aio\_read->do\_sock\_read->\_\_sock\_recvmsg->\_\_sock\_recvmsg\_nose->udp\_recvmsg，代码如下所示：

```
int udp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
                size_t len, int noblock, int flags, int *addr_len)
.....
    ulen = skb->len - sizeof(struct udphdr);
    copied = len;
    if (copied > ulen)
        copied = ulen;
    .....
```

当 UDP 报文的数据长度小于参数 len 时，就会只复制真正的数据长度，那么对于 read 操作来说，返回的读取字节数自然就小于参数 count 了。

看到这里，是否已经得到本小节开头部分问题的答案了呢？当 fd 中的数据不够 count 大小时，read 会返回当前可以读取的字节数？很可惜，答案是否定的。这种行为完全由具体实现来决定。即使同为 socket 文件系统，TCP 套接字的读取操作也会与 UDP 不同。当 TCP 的 fd 的数据不足时，read 操作极可能会阻塞，而不是直接返回。注：TCP 是否阻塞，取决于当前缓存区可用数据多少，要读取的字节数，以及套接字设置的接收低水位大小。

因此在调用 read 的时候，只能根据 read 接口的说明，小心处理所有的情况，而不能主观臆测内核的实现。比如本文中的部分读取情况，阻塞和直接返回两种策略同时存在。

## 1.7 写入文件

Linux 中写入文件操作，最常用的就是 write 函数，其原型如下：

```
ssize_t write(int fd, const void *buf, size_t count);
```

write 尝试从 buf 指向的地址，写入 count 个字节到文件描述符 fd 中，并返回成功写入的字节数，同时将文件偏移向前移动相同的字节数。write 有可能写入比指定 count 少的字节数。

### 1.7.1 write 源码跟踪

write 的源码与 read 的很相似，位于 read\_write.c 中，代码如下：

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
                size_t, count)
```

```

{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    /* 得到file管理结构指针 */
    file = fget_light(fd, &fput_needed);
    if (file) {
        /* 得到当前的文件偏移 */
        loff_t pos = file_pos_read(file);
        /* 利用VFS写入 */
        ret = vfs_write(file, buf, count, &pos);
        /* 更新文件偏移量 */
        file_pos_write(file, pos);
        /* 释放文件管理指针file */
        fput_light(file, fput_needed);
    }

    return ret;
}

```

进入 `vfs_write`，代码如下：

```

ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;

    /* 检查文件是否为写入打开 */
    if (!(file->f_mode & FMODE_WRITE))
        return -EBADF;
    /* 检查文件是否支持打开操作 */
    if (!file->f_op || (!file->f_op->write && !file->f_op->aio_write))
        return -EINVAL;
    /* 检查用户给定的地址范围是否可读取 */
    if (unlikely(!access_ok(VERIFY_READ, buf, count)))
        return -EFAULT;

    /*
     * 验证文件从pos起始是否可以写入count个字节数
     * 并返回可以写入的字节数
     */
    ret = rw_verify_area(WRITE, file, pos, count);
    if (ret >= 0) {
        /* 更新写入字节数 */
        count = ret;
        /*
         * 如果定义write操作，则执行定义的write操作
         * 如果没有定义write操作，则调用do_sync_write——其利用异步
         * aio_write来完成同步的write操作
         */
        if (file->f_op->write)
            ret = file->f_op->write(file, buf, count, pos);
        else
            ret = do_sync_write(file, buf, count, pos);
        if (ret > 0) {

```

```

        /* 写入了一定的字节数，进行通知操作 */
        fsnotify_modify(file);
        /* 增加进程读取字节的统计计数 */
        add_wchar(current, ret);
    }
    /* 增加进程系统调用的统计计数 */
    inc_syscw(current);
}

return ret;
}

```

write 同样有部分写入的情况，这个与 read 类似，都是由具体实现来决定的。在此就不再深入探讨 write 的部分写入的情况了。

### 1.7.2 追加写的实现

前面说过，文件的读写操作都是从当前文件的偏移处开始的。这个文件偏移量保存在文件表中，而每个进程都有一个文件表。那么当多个进程同时写一个文件时，即使对 write 进行了锁保护，在进行串行写操作时，文件依然不可避免地会被写乱。根本原因就在于文件偏移量是进程级别的。

当使用 O\_APPEND 以追加的形式来打开文件时，每次写操作都会先定位到文件末尾，然后再执行写操作。

Linux 下大多数文件系统都是调用 generic\_file\_aio\_write 来实现写操作的。在 generic\_file\_aio\_write 中，有如下代码：

```

mutex_lock(&inode->i_mutex);
blk_start_plug(&plug);
ret = __generic_file_aio_write(iocb, iov, nr_segs, &iocb->ki_pos);
mutex_unlock(&inode->i_mutex);

```

这里有一个关键的语句，就是使用 mutex\_lock 对该文件对应的 inode 进行保护，然后调用 \_\_generic\_file\_aio\_write->generic\_write\_check。其部分代码如下：

```

if (file->f_flags & O_APPEND)
    *pos = i_size_read(inode);

```

上面的代码中，如果发现文件是以追加方式打开的，则将从 inode 中读取到的最新文件大小作为偏移量，然后通过 \_\_generic\_file\_aio\_write 再进行写操作，这样就能保证写操作是在文件末尾追加的。

## 1.8 文件的原子读写

使用 O\_APPEND 可以实现在文件的末尾原子追加新数据，Linux 还提供 pread 和 pwrite 从指定偏移位置读取或写入数据。

它们的实现很简单，代码如下：

```
SYSCALL_DEFINE(pread64)(unsigned int fd, char __user *buf,
                        size_t count, loff_t pos)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    if (pos < 0)
        return -EINVAL;

    file = fget_light(fd, &fput_needed);
    if (file) {
        ret = -ESPIPE;
        if (file->f_mode & FMODE_PREAD)
            ret = vfs_read(file, buf, count, &pos);
        fput_light(file, fput_needed);
    }

    return ret;
}
```

看到这段代码，是不是有一种似曾相识的感觉？让我们再来回顾一下 read 的实现，代码如下所示。

```
/* 得到文件的当前偏移量 */
loff_t pos = file_pos_read(file);
/* 利用vfs进行真正的read */
ret = vfs_read(file, buf, count, &pos);
/* 更新文件偏移量 */
file_pos_write(file, pos);
```

这就是它与 read 的主要区别。pread 不会从文件表中获取当前偏移，而是直接使用用户传递的偏移量，并且在读取完毕后，不会更改当前文件的偏移量。

pwrite 的实现与 pread 类似，在此就不再重复描述了。

## 1.9 文件描述符的复制

Linux 提供了三个复制文件描述符的系统调用，分别为：

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int dup3(int oldfd, int newfd, int flags);
```

其中：

- ❑ dup 会使用一个最小的未用文件描述符作为复制后的文件描述符。
- ❑ dup2 是使用用户指定的文件描述符 newfd 来复制 oldfd 的。如果 newfd 已经是打开的文件描述符，Linux 会先关闭 newfd，然后再复制 oldfd。

❑ 对于 dup3，只有定义了 feature 宏 “\_GNU\_SOURCE” 才可以使用，它比 dup2 多了一个参数，可以指定标志——不过目前仅仅支持 O\_CLOEXEC 标志，可在 newfd 上设置 O\_CLOEXEC 标志。定义 dup3 的原因与 open 类似，可以在进行 dup 操作的同时原子地将 fd 设置为 O\_CLOEXEC，从而避免将文件内容暴露给子进程。

为什么会有 dup、dup2、dup3 这种像兄弟一样的系统调用呢？这是因为随着软件工程的日益复杂，已有的系统调用已经无法满足需求，或者存在安全隐患，这时，就需要内核针对已有问题推出新的接口。

话说在很久以前，程序员在写 daemon 服务程序时，基本上都有这样的流程：首先关闭标准输出 stdout、标准出错 stderr，然后进行 dup 操作，将 stdout 或 stderr 重定向。但是在多线程程序成为主流以后，由于 close 和 dup 操作不是原子的，这就造成了在某些情况下，重定向会失败。因此就引入了 dup2 将 close 和 dup 合为一个系统调用，以保证原子性，然而这依然有问题。大家可以回顾 1.2.2 节中对 O\_CLOEXEC 的介绍。在多线程中进行 fork 操作时，dup2 同样会有让相同的文件描述符暴露的风险，dup3 也就随之诞生了。这三个系统调用看起来有些冗余重复，但实际上它们也是软件工程发展的结果。从这个 dup 的发展过程来看，我们也可以领会到编写健壮代码的不易。正如前文所述，对于一个现代接口，一般都会有一个 flag 标志参数，这样既可以保证兼容性，还可以通过引用新的标志来改善或纠正接口的行为。

下面先看 dup 的实现，如下所示：

```
SYSCALL_DEFINE1(dup, unsigned int, fildes)
{
    int ret = -EBADF;
    /* 必须先得到文件管理结构file，同时也是对描述符fildes的检查 */
    struct file *file = fget_raw(fildes);

    if (file) {
        /* 得到一个未使用的文件描述符 */
        ret = get_unused_fd();
        if (ret >= 0) {
            /* 将文件描述符与file指针关联起来 */
            fd_install(ret, file);
        }
        else
            fput(file);
    }
    return ret;
}
```

然后，再看看 fd\_install 的实现，代码如下所示：

```
void fd_install(unsigned int fd, struct file *file)
{
    struct files_struct *files = current->files;
    struct fdtable *fdt;
    /* 对文件表进行保护 */
    spin_lock(&files->file_lock);
```

```

/* 得到文件表 */
fdt = files_fdttable(files);
BUG_ON(fdt->fd[fd] != NULL);
/* 让文件表中fd对应的指针等于该文件关联结构file */
rcu_assign_pointer(fdt->fd[fd], file);
spin_unlock(&files->file_lock);
}

```

在 `dup` 中调用 `get_unused_fd`，只是得到一个未用的文件描述符，那么如何实现在 `dup` 接口中使用最小的未用文件描述符呢？这就需要回顾 1.4.2 节中总结过的 Linux 文件描述符的选择策略了。

Linux 总是尝试给用户最小的未用文件描述符，所以 `get_unused_fd` 得到的文件描述符始终是最小的可用文件描述符。

在 `fd_install` 中，`fd` 与 `file` 的关联是利用 `fd` 来作为指针数组的索引的，从而让对应的指针指向 `file`。对于 `dup` 来说，这意味着数组中两个指针都指向了同一个 `file`。而 `file` 是进程中真正的管理文件的结构，文件偏移等信息都是保存在 `file` 中的。这就意味着，当使用 `oldfd` 进行读写操作时，无论是 `oldfd` 还是 `newfd` 的文件偏移都会发生变化。

再来看一下 `dup2` 的实现，如下所示：

```

SYSCALL_DEFINE2(dup2, unsigned int, oldfd, unsigned int, newfd)
{
    /* 如果oldfd与newfd相等，这是一种特殊的情况 */
    if (unlikely(newfd == oldfd)) { /* corner case */
        struct files_struct *files = current->files;
        int retval = oldfd;

        /*
         * 检查oldfd的合法性，如果是合法的fd，则直接返回oldfd的值；
         * 如果是不合法的，则返回EBADF
         */
        rcu_read_lock();
        if (!fcheck_files(files, oldfd))
            retval = -EBADF;
        rcu_read_unlock();
        return retval;
    }
    /* 如果oldfd与newfd不同，则利用sys_dup3来实现dup2 */
    return sys_dup3(oldfd, newfd, 0);
}

```

再来查看一下 `dup3` 的实现代码，如下所示：

```

SYSCALL_DEFINE3(dup3, unsigned int, oldfd, unsigned int, newfd, int, flags)
{
    int err = -EBADF;
    struct file * file, *tofree;
    struct files_struct * files = current->files;
    struct fdtable *fdt;

    /* 对标志flags进行检查，支持O_CLOEXEC */

```



```

if ((flags & ~O_CLOEXEC) != 0)
    return -EINVAL;

/* 与dup2不同, 当oldfd与newfd相同的时候, dup3返回错误 */
if (unlikely(oldfd == newfd))
    return -EINVAL;

spin_lock(&files->file_lock);
/* 根据newfd决定是否扩展文件表的大小 */
err = expand_files(files, newfd);
/*
检查oldfd, 如果是非法的, 就直接返回
不过我更倾向于先检查oldfd后扩展文件表, 如果是非法的, 就不需要扩展文件表了
*/
file = fcheck(oldfd);
if (unlikely(!file))
    goto Ebadf;
if (unlikely(err < 0)) {
    if (err == -EMFILE)
        goto Ebadf;
    goto out_unlock;
}

err = -EBUSY;
/* 得到文件表 */
fdt = files_fdt(files);
/* 通过newfd得到对应的file结构 */
tofree = fdt->fd[newfd];
/*
tofree是NULL, 但是newfd已经分配的情况
*/
if (!tofree && FD_ISSET(newfd, fdt->open_fds))
    goto out_unlock;
/* 增加file的引用计数 */
get_file(file);
/* 将文件表newfd对应的指针指向file */
rcu_assign_pointer(fdt->fd[newfd], file);
/*
将newfd加到打开文件的位图中
如果newfd已经是一个合法的fd, 重复设置位图则没有影响;
如果newfd没有打开, 则必须将其加入位图中
那么为什么不新newfd进行检查呢? 因为检查比设置位图更消耗CPU
*/
FD_SET(newfd, fdt->open_fds);
/*
如果flags设置了O_CLOEXEC, 则将newfd加到close_on_exec位图;
如果没有设置, 则清除close_on_exec位图中对应的位
*/
if (flags & O_CLOEXEC)
    FD_SET(newfd, fdt->close_on_exec);
else
    FD_CLR(newfd, fdt->close_on_exec);
spin_unlock(&files->file_lock);

/* 如果tofree不为空, 则需要关闭newfd之前的文件 */

```

```

    if (torelease)
        filp_close(torelease, files);

    return newfd;

Ebadf:
    err = -EBADF;
out_unlock:
    spin_unlock(&files->file_lock);
    return err;
}

```

## 1.10 文件数据的同步

为了提高性能，操作系统会对文件的 I/O 操作进行缓存处理。对于读操作，如果要读取的内容已经存在于文件缓存中，就直接读取文件缓存。对于写操作，会先将修改提交到文件缓存中，在合适的时机或者过一段时间后，操作系统才会将改动提交到磁盘上。

Linux 提供了三个同步接口：

```

void sync(void);
int fsync(int fd);
int fdatasync(int fd);

```

APUE 上说 `sync` 只是让所有修改过的缓存进入提交队列，并不用等待这个工作完成。Linux 手册上则表示从 1.3.20 版本开始，Linux 就会一直等待，直到提交工作完成。

实际情况到底是怎样的呢，让代码告诉我们真相，具体如下：

```

SYSCALL_DEFINE0(sync)
{
    /* 唤醒后台内核线程，将“脏”缓存冲刷到磁盘上 */
    wakeup_flusher_threads(0, WB_REASON_SYNC);
    /*
     * 为什么要调用两次sync_filesystems呢？
     * 这是一种编程技巧，第一次sync_filesystems(0)，参数0表示不等待，可以
     * 迅速地将没有上锁的inode同步。第二次sync_filesystems(1)，参数1表示等待。
     * 对于上锁的inode会等待到解锁，再执行同步，这样可以提高性能。因为第一次操作
     * 中，上锁的inode很可能在第一次操作结束后，就已经解锁，这样就避免了等待
     */
    sync_filesystems(0);
    sync_filesystems(1);
    /*
     * 如果是laptop模式，那么因为此处刚刚做完同步，因此可以停掉后台同步定时器
     */
    if (unlikely(laptop_mode))
        laptop_sync_completion();
    return 0;
}

```

再看一下 `sync_filesystems->iterate_supers->sync_one_sb->__sync_filesystem`，代码如下：

```

static int __sync_filesystem(struct super_block *sb, int wait)

```

```

{
    /*
     * This should be safe, as we require bdi backing to actually
     * write out data in the first place
     */
    if (sb->s_bdi == &noop_backing_dev_info)
        return 0;

    /* 磁盘配额同步 */
    if (sb->s_qcop && sb->s_qcop->quota_sync)
        sb->s_qcop->quota_sync(sb, -1, wait);

    /*
     * 如果wait为true, 则一直等待直到所有的脏inode写入磁盘
     * 如果wait为false, 则启动脏inode回写工作, 但不必等待到结束
     */
    if (wait)
        sync_inodes_sb(sb);
    else
        writeback_inodes_sb(sb, WB_REASON_SYNC);

    /* 如果该文件系统定义了自己的同步操作, 则执行该操作 */
    if (sb->s_op->sync_fs)
        sb->s_op->sync_fs(sb, wait);

    /* 调用block设备的flush操作, 真正地将数据写到设备上 */
    return __sync_blockdev(sb->s_bdev, wait);
}

```

从 sync 的代码实现上看, Linux 的 sync 是阻塞调用, 这里与 APUE 的说明是不一样的。

下面来看看 fsync 与 fdatasync, fsync 只同步 fd 指定的文件, 并且直到同步完成才返回。fdatasync 与 fsync 类似, 但是其只同步文件的实际数据内容, 会影响后面数据操作的元数据。而 fsync 不仅同步数据, 还会同步所有被修改过的文件元数据, 代码如下所示:

```

SYSCALL_DEFINE1(fsync, unsigned int, fd)
{
    return do_fsync(fd, 0);
}

SYSCALL_DEFINE1(fdatasync, unsigned int, fd)
{
    return do_fsync(fd, 1);
}

```

事实上, 真正进行工作的是 do\_fsync, 代码如下所示:

```

static int do_fsync(unsigned int fd, int datasync)
{
    struct file *file;
    int ret = -EBADF;
    /* 得到file管理结构 */
    file = fget(fd);
    if (file) {

```

```

        /* 利用vfs执行sync操作 */
        ret = vfs_fsync(file, datasync);
        fput(file);
    }
    return ret;
}

```

进入 `vfs_fsync->vfs_fsync_range`，代码如下：

```

int vfs_fsync_range(struct file *file, loff_t start, loff_t end, int datasync)
{
    /* 调用具体操作系统的同步操作 */
    if (!file->f_op || !file->f_op->fsync)
        return -EINVAL;
    return file->f_op->fsync(file, start, end, datasync);
}

```

真正执行同步操作的 `fsync` 是由具体的文件系统的操作函数 `file_operations` 决定的。下面选择一个常用的文件系统同步函数 `generic_file_fsync`，代码如下。

```

int generic_file_fsync(struct file *file, loff_t start, loff_t end,
                      int datasync)
{
    struct inode *inode = file->f_mapping->host;
    int err;
    int ret;

    /* 同步该文件缓存中处于start到end范围内的脏页 */
    err = filemap_write_and_wait_range(inode->i_mapping, start, end);
    if (err)
        return err;

    mutex_lock(&inode->i_mutex);
    /* 同步该inode对应的缓存 */
    ret = sync_mapping_buffers(inode->i_mapping);
    /* inode状态没有变化，无需同步，可以直接返回 */
    if (!(inode->i_state & I_DIRTY))
        goto out;
    /* 如果是fdatasync则仅做数据同步，并且若该inode没有影响任何数据方面操作的变化（比如文件长度），则可以直接返回 */
    if (datasync && !(inode->i_state & I_DIRTY_DATASYNC))
        goto out;

    /*
     * 同步inode的元数据
     */
    err = sync_inode_metadata(inode, 1);
    if (ret == 0)
        ret = err;

out:
    mutex_unlock(&inode->i_mutex);
    return ret;
}

```

从上面的代码可以看出，`fdatasync` 的性能会优于 `fsync`。在不需要同步所有元数据的

情况下，选择 `fdatasync` 会得到更好的性能。只有在 `inode` 被设置了 `I_DIRTY_DATASYNC` 标志时，`fdatasync` 才需要同步 `inode` 的元数据。那么 `inode` 何时会被设置 `I_DIRTY_DATASYNC` 这个标志呢？比如使用文件截断 `truncate` 或 `ftruncate` 时；通过在源码中搜索 `I_DIRTY_DATASYNC` 或 `mark_inode_dirty` 时也会给 `inode` 设置该标志位。而调用 `mark_inode_dirty` 的地方就太多了，这里就不一一列举了。



**注意** `sync`、`fsync` 和 `fdatasync` 只能保证 Linux 内核对文件的缓冲被冲刷了，并不能保证数据被真正写到磁盘上，因为磁盘也有自己的缓存。

## 1.11 文件的元数据

1.10 节中我们提到了文件元数据，那么什么是文件的元数据呢？其包括文件的访问权限、上次访问的时间戳、所有者、所有组、文件大小等信息。

### 1.11.1 获取文件的元数据

Linux 环境提供了三个获取文件信息的 API：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

这三个函数都可用于得到文件的基本信息，区别在于 `stat` 得到路径 `path` 所指定的文件基本信息，`fstat` 得到文件描述符 `fd` 指定文件的基本信息，而 `lstat` 与 `stat` 则基本相同，只有当 `path` 是一个链接文件时，`lstat` 得到的是链接文件自己本身的基本信息而不是其指向文件的信息。

所得到的文件基本信息的结果 `struct stat` 的结构如下：

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* inode number */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device ID (if special file) */
    off_t      st_size;     /* total size, in bytes */
    blksize_t  st_blksize;  /* blocksize for file system I/O */
    blkcnt_t   st_blocks;   /* number of 512B blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
};
```

```

        time_t      st_ctime;    /* time of last status change */
};

```

Linux 的 man 手册对 stat 的各个变量做了注释，明确指出了每个变量的意义。唯一需要说明的是 st\_mode，其不仅仅是注释所说的“protection”，即权限管理，同时也用于表示文件类型，比如是普通文件还是目录。

### 1.11.2 内核如何维护文件的元数据

要搞清楚 Linux 如何维护文件的元数据，就需要追踪 stat 的实现，具体代码如下：

```

SYSCALL_DEFINE2(stat, const char __user *, filename,
                struct __old_kernel_stat __user *, statbuf)
{
    struct kstat stat;
    int error;

    /* vfs_stat用于读取文件元数据至stat */
    error = vfs_stat(filename, &stat);
    if (error)
        return error;

    /* 这里仅是从内核的元数据结构stat复制到用户层的数据结构statbuf中 */
    return cp_old_stat(&stat, statbuf);
}

```

进入 vfs\_stat->vfs\_fstatat->vfs\_getattr，代码如下：

```

int vfs_getattr(struct vfsmount *mnt, struct dentry *dentry, struct kstat *stat)
{
    struct inode *inode = dentry->d_inode;
    int retval;

    /* 对获取inode属性操作进行安全性检查 */
    retval = security_inode_getattr(mnt, dentry);
    if (retval)
        return retval;

    /* 如果该文件系统定义了这个inode的自定义操作函数，就执行它 */
    if (inode->i_op->getattr)
        return inode->i_op->getattr(mnt, dentry, stat);

    /* 如果文件系统没有定义inode的操作函数，则执行通用的函数 */
    generic_fillattr(inode, stat);
    return 0;
}

```

不失一般性，也可以通过查看 generic\_fillattr 来进一步了解，代码如下：

```

void generic_fillattr(struct inode *inode, struct kstat *stat)
{
    stat->dev = inode->i_sb->s_dev;
    stat->ino = inode->i_ino;
}

```

```

stat->mode = inode->i_mode;
stat->nlink = inode->i_nlink;
stat->uid = inode->i_uid;
stat->gid = inode->i_gid;
stat->rdev = inode->i_rdev;
stat->size = i_size_read(inode);
stat->atime = inode->i_atime;
stat->mtime = inode->i_mtime;
stat->ctime = inode->i_ctime;
stat->blksize = (1 << inode->i_blkbits);
stat->blocks = inode->i_blocks;
}

```

从这里可以看出，所有的文件元数据均保存在 inode 中，而 inode 是 Linux 也是所有类 Unix 文件系统中的概念。这样的文件系统一般将存储区域分为两类，一类是保存文件对象的元信息数据，即 inode 表；另一类是真正保存文件数据内容的块，所有 inode 完全由文件系统来维护。但是 Linux 也可以挂载非类 Unix 的文件系统，这些文件系统本身没有 inode 的概念，怎么办？Linux 为了让 VFS 有统一的处理流程和方法，就必须要求那些没有 inode 概念的文件系统，根据自己系统的特点——如何维护文件元数据，生成“虚拟的”inode 以供 Linux 内核使用。

### 1.11.3 权限位解析

在 Linux 环境中，文件常见的权限位有 r、w 和 x，分别表示可读、可写和可执行。下面重点解析三个不常用的标志位。

#### 1. SUID 权限位

当文件设置 SUID 权限位时，就意味着无论是谁执行这个文件，都会拥有该文件所有者的权限。passwd 命令正是利用这个特性，来允许普通用户修改自己的密码，因为只有 root 用户才有修改密码文件的权限。当普通用户执行 passwd 命令时，就具有了 root 权限，从而可以修改自己的密码。

以修改文件属性的权限检查代码为例，inode\_change\_ok 用于检查该进程是否有权限修改 inode 节点的属性即文件属性，示例代码如下：

```

int inode_change_ok(const struct inode *inode, struct iattr *attr)
{
    unsigned int ia_valid = attr->ia_valid;

    .....

    /* Make sure a caller can chown. */
    /* 只有在uid和suid都不符合条件的情况下，才会返回权限不足的错误 */
    if ((ia_valid & ATTR_UID) &&
        (current_fsuid() != inode->i_uid ||
         attr->ia_uid != inode->i_uid) && !capable(CAP_CHOWN))
        return -EPERM;

    .....
}

```

## 2. SGID 权限位

SGID 与 SUID 权限位类似，当设置该权限位时，就意味着无论是谁执行该文件，都会拥有该文件所有者所在组的权限。

## 3. Stricky 位

Stricky 位只有配置在目录上才有意义。当目录配置上 sticky 位时，其效果是即使所有的用户都拥有写权限和执行权限，该目录下的文件也只能被 root 或文件所有者删除。

下面来看看内核的实现：

```
static int may_delete(struct inode *dir, struct dentry *victim, int isdir)
{
    .....
    if (check_sticky(dir, victim->d_inode) ||
        IS_APPEND(victim->d_inode) ||
        IS_IMMUTABLE(victim->d_inode) ||
        IS_SWAPFILE(victim->d_inode))
        return -EPERM;
    .....
}
```

在删除文件前，内核要调用 may\_delete 来判断该文件是否可以被删除。在这个函数中，内核通过调用 check\_sticky 来检查文件的 sticky 标志位，其代码如下：

```
static inline int check_sticky(struct inode *dir, struct inode *inode)
{
    /* 得到当前文件访问权限的uid */
    uid_t fsuid = current_fsuid();

    /* 判断上级目录是否设置了sticky标志位 */
    if (!(dir->i_mode & S_ISVTX))
        return 0;
    /* 检查名称空间 */
    if (current_user_ns() != inode_userns(inode))
        goto other_userns;
    /* 检查当前文件的uid是否与当前用户的uid相同 */
    if (inode->i_uid == fsuid)
        return 0;
    /* 检查文件所处目录的uid是否与当前用户的uid相同 */
    if (dir->i_uid == fsuid)
        return 0;

    /* 该文件不属于当前用户 */
other_userns:
    return !ns_capable(inode_userns(inode), CAP_FOWNER);
}
```

当文件所处的目录设置了 sticky 位，即使用户拥有了对应的权限，只要不是目录或文件



的拥有者，就无法删除该文件——除非该用户拥有 CAP\_FOWNER 能力（读者可以通过 man 7 capabilities 来进一步了解 Linux 中的 capabilities。一般只有 root 用户才有这样的能力）。



**说明** 大家可以使用 chmod 来设置文件或目录的权限。

## 1.12 文件截断

### 1.12.1 truncate 与 ftruncate 的简单介绍

Linux 提供了两个截断文件的 API：

```
#include <unistd.h>
#include <sys/types.h>

int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

两者之间的唯一区别在于，truncate 截断的是路径 path 指定的文件，ftruncate 截断的是 fd 引用的文件。

“截断”给人的感觉是将文件变短，即将文件大小缩短至 length 长度。实际上，length 可以大于文件本身的大小，这时文件长度将变为 length 的大小，扩充的内容均被填充为 0。需要注意的是，尽管 ftruncate 使用的是文件描述符，但是其并不会更新当前文件的偏移。

### 1.12.2 文件截断的内核实现

先来看看 truncate 的内核实现，代码如下：

```
SYSCALL_DEFINE2(truncate, const char __user *, path, long, length)
{
    return do_sys_truncate(path, length);
}
```

进入 do\_sys\_truncate，代码如下：

```
static long do_sys_truncate(const char __user *pathname, loff_t length)
{
    struct path path;
    struct inode *inode;
    int error;

    error = -EINVAL;
    /* 长度不能为负数 */
    if (length < 0)
        goto out;

    /* 得到路径结构 */
    error = user_path(pathname, &path);
    if (error)
        goto out;
```

```

inode = path.dentry->d_inode;

error = -EISDIR;
/* 目录不能被截断 */
if (S_ISDIR(inode->i_mode))
    goto dput_and_out;
error = -EINVAL;
/* 不是普通文件不能被截断 */
if (!S_ISREG(inode->i_mode))
    goto dput_and_out;

/* 尝试获得文件系统的写权限 */
error = mnt_want_write(path.mnt);
if (error)
    goto dput_and_out;

/* 检查是否有文件写权限 */
error = inode_permission(inode, MAY_WRITE);
if (error)
    goto mnt_drop_write_and_out;

error = -EPERM;
/* 文件设置了追加属性，则不能被截断 */
if (IS_APPEND(inode))
    goto mnt_drop_write_and_out;

/* 得到inode的写权限 */
error = get_write_access(inode);
if (error)
    goto mnt_drop_write_and_out;

/* 查看是否与文件lease锁相冲突 */
error = break_lease(inode, O_WRONLY);
if (error)
    goto put_write_and_out;

/* 检查是否与文件锁相冲突 */
error = locks_verify_truncate(inode, NULL, length);
if (!error)
    error = security_path_truncate(&path);

/* 如果没有错误，则进行真正的截断 */
if (!error)
    error = do_truncate(path.dentry, length, 0, NULL);

put_write_and_out:
    put_write_access(inode);
mnt_drop_write_and_out:
    mnt_drop_write(path.mnt);
dput_and_out:
    path_put(&path);
out:
    return error;
}

```

再进入 do\_truncate，代码如下：

```
int do_truncate(struct dentry *dentry, loff_t length, unsigned int time_attrs,
                struct file *filp)
{
    int ret;
    struct iattr newattrs;

    if (length < 0)
        return -EINVAL;

    /* 设置要改变的属性，对于截断来说，最重要的是文件长度 */
    newattrs.ia_size = length;
    newattrs.ia_valid = ATTR_SIZE | time_attrs;
    if (filp) {
        newattrs.ia_file = filp;
        newattrs.ia_valid |= ATTR_FILE;
    }

    /*
     * suid权限一定会被去掉
     * 同时设置sgid和xgrp时，sgid权限也会被去掉
     */
    ret = should_remove_suid(dentry);
    if (ret)
        newattrs.ia_valid |= ret | ATTR_FORCE;

    /* 修改inode属性 */
    mutex_lock(&dentry->d_inode->i_mutex);
    ret = notify_change(dentry, &newattrs);
    mutex_unlock(&dentry->d_inode->i_mutex);
    return ret;
}
```

接下来看 ftruncate 的实现，代码如下：

```
SYSCALL_DEFINE2(ftruncate, unsigned int, fd, unsigned long, length)
{
    /* 真正的工作函数do_sys_ftruncate */
    long ret = do_sys_ftruncate(fd, length, 1);
    /* avoid REGPARM breakage on x86: */
    asmlinkage_protect(2, ret, fd, length);
    return ret;
}
```

最后，进入 do\_sys\_ftruncate，代码如下：

```
static long do_sys_ftruncate(unsigned int fd, loff_t length, int small)
{
    struct inode * inode;
    struct dentry *dentry;
    struct file * file;
    int error;

    error = -EINVAL;
```

```

/* 长度检查 */
if (length < 0)
    goto out;
error = -EBADF;
/* 从文件描述符得到file指针 */
file = fget(fd);
if (!file)
    goto out;

/* 如果文件是以O_LARGEFILE选项打开的，则将标志small置为0即假 */
if (file->f_flags & O_LARGEFILE)
    small = 0;

dentry = file->f_path.dentry;
inode = dentry->d_inode;
error = -EINVAL;
/* 如果文件不是普通文件或文件不是写打开，则报错 */
if (!S_ISREG(inode->i_mode) || !(file->f_mode & FMODE_WRITE))
    goto out_putf;

error = -EINVAL;      /* Cannot ftruncate over 2^31 bytes without large file
support */
/* 如果文件不是以O_LARGEFILE打开的话，长度就不能超过MAX_NON_LFS */
if (small && length > MAX_NON_LFS)
    goto out_putf;

error = -EPERM;
/* 如果是追加模式打开的，也不能进行截断 */
if (IS_APPEND(inode))
    goto out_putf;
/* 检查是否有锁冲突 */
error = locks_verify_truncate(inode, file, length);
if (!error)
    error = security_path_truncate(&file->f_path);
if (!error) {
    /* 执行截断操作——前文已经分析过 */
    error = do_truncate(dentry, length, ATTR_MTIME|ATTR_CTIME, file);}
out_putf:
    fput(file);
out:
    return error;
}

```

### 1.12.3 为什么需要文件截断

文件截断时允许指定比原有文件长度更长的值，但更常见的是指定的长度比原有长度短，这主要用于防止文件内容混杂了旧内容的情况。下面以常见的 daemon 程序为例（演示一个文件因不截断而引发的 bug），这种程序往往要将自己的 pid 写入一个 pid 文件中。当 daemon 程序启动的时候，最好是将旧的 pid 文件截断，然后写入新的 pid，不然 pid 文件中可能会保存错误的 pid。

假设当前的 test.pid 文件的内容是上一次的 pid。

```
[fgao@fgao chapter1]#cat test.pid
123456
```

下面的程序是将新的 pid——6789 写入 test.pid 中。

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

int main(void)
{
    int fd = open("test.pid", O_WRONLY);

    write(fd, "6789", sizeof("6789")-1);

    close(fd);

    return 0;
}
```

程序执行完毕，让我们看看 test.pid 的内容：

```
[fgao@fgao chapter1]#cat test.pid
678956
```

这显然不是我们所期望的结果。为了解决这个问题，我们可以在打开文件的同时，指定 O\_TRUNC 标志。

```
int fd = open("test.pid", O_WRONLY | O_TRUNC);
```

或者使用本节介绍的截断 API，代码如下：

```
truncate("test.pid", 0);
int fd = open("test.pid", O_WRONLY);
```

或者用如下代码：

```
int fd = open("test.pid", O_WRONLY);
ftruncate(fd, 0);
```

这样，就能保证旧内容不会与最新写入的内容混杂在一起。

也许有朋友会提出，在上面的例子中写入“6789”时，这样写就不会有问题了：

```
write(fd, "6789", sizeof("6789"));
```

然而结果仍然是错的，其结果为：

```
[fgao@ubuntu chapter1]#cat test.pid
67896
```

这里列举的例子用的是文本文件，如果写入的是一个二进制文件，当不使用文件截断而导致新旧数据混杂在一起时，定位错误将更加困难。所以，在我们的日常编码中，在写入文件，如果并不需要旧数据，那么在打开文件时就要强制截断文件，来提高代码的健壮性。

## 标准 I/O 库

前面的章节介绍的是 Linux 的系统调用。本章将从标准 I/O 库开始讲解 Linux 环境编程中不可或缺的 C 库。在学习和分析标准 I/O 库的同时，与 Linux 的 I/O 系统调用进行比较，可以加深对两者的认识和理解。

### 2.1 stdin、stdout 和 stderr

当 Linux 新建一个进程时，会自动创建 3 个文件描述符 0、1 和 2，分别对应标准输入、标准输出和错误输出。C 库中与文件描述符对应的是文件指针，与文件描述符 0、1 和 2 类似，我们可以直接使用文件指针 stdin、stdout 和 stderr。那么这是否意味着 stdin、stdout 和 stderr 是“自动打开”的文件指针呢？

查看 C 库头文件 stdio.h 中的源码：

```
typedef struct _IO_FILE FILE;

/* Standard streams. */
extern struct _IO_FILE *stdin;      /* Standard input stream. */
extern struct _IO_FILE *stdout;     /* Standard output stream. */
extern struct _IO_FILE *stderr;     /* Standard error output stream. */
#ifdef __STDC__
/* C89/C99 say they're macros. Make them happy. */
#define stdin stdin
#define stdout stdout
#define stderr stderr
#endif
```

从上面的源码可以看出，stdin、stdout 和 stderr 确实是文件指针。而 C 标准要求 stdin、

`stdout` 和 `stderr` 是宏定义，所以在 C 库的代码中又定义了同名宏。

那么 `stdin`、`stdout` 和 `stderr` 又是如何定义的呢？定义代码如下：

```
_IO_FILE *stdin = (FILE *) &_IO_2_1_stdin_;
_IO_FILE *stdout = (FILE *) &_IO_2_1_stdout_;
_IO_FILE *stderr = (FILE *) &_IO_2_1_stderr_;
```

继续查看 `_IO_2_1_stdin_` 等的定义，代码如下：

```
DEF_STDFILE(_IO_2_1_stdin_, 0, 0, _IO_NO_WRITES);
DEF_STDFILE(_IO_2_1_stdout_, 1, &_IO_2_1_stdin_, _IO_NO_READS);
DEF_STDFILE(_IO_2_1_stderr_, 2, &_IO_2_1_stdout_, _IO_NO_READS+_IO_UNBUFFERED);
```

`DEF_STDFILE` 是一个宏定义，用于初始化 C 库中的 `FILE` 结构。这里 `_IO_2_1_stdin_`、`_IO_2_1_stdout_` 和 `_IO_2_1_stderr_` 这三个 `FILE` 结构分别用于文件描述符 0、1 和 2 的初始化，这样 C 库的文件指针就与系统的文件描述符互相关联起来了。大家注意最后的标志位，`stdin` 是不可写的，`stdout` 是不可读的，而 `stderr` 不仅不可读，且没有缓存。

通过上面的分析，可以得到一个结论：`stdin`、`stdout` 和 `stderr` 都是 `FILE` 类型的文件指针，是由 C 库静态定义的，直接与文件描述符 0、1 和 2 相关联，所以应用程序可以直接使用它们。

## 2.2 I/O 缓存引出的趣题

C 库的 I/O 接口对文件 I/O 进行了封装，为了提高性能，其引入了缓存机制，共有三种缓存机制：全缓存、行缓存及无缓存。

- ❑ 全缓存一般用于访问真正的磁盘文件。C 库会为文件访问申请一块内存，只有当文件内容将缓存填满或执行冲刷函数 `flush` 时，C 库才会将缓存内容写入内核中。
- ❑ 行缓存一般用于访问终端。当遇到一个换行符时，就会引发真正的 I/O 操作。需要注意的是，C 库的行缓存也是固定大小的。因此，当缓存已满，即使没有换行符时也会引发 I/O 操作。
- ❑ 无缓存，顾名思义，C 库没有进行任何的缓存。任何 C 库的 I/O 调用都会引发实际的 I/O 操作。

C 库提供了接口，用于修改默认的缓存行为，相关代码如下：

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
void setbuffer(FILE *stream, char *buf, size_t size);
void setlinebuf(FILE *stream);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

下面看一个跟 C 库缓存相关的趣题。

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

```
#include <stdlib.h>

#include <unistd.h>

int main(void)
{
    printf("Hello ");

    if (0 == fork()) {
        printf("child\n");
        return 0;
    }
    printf("parent\n");
    return 0;
}
```

其输出结果是什么？正确的结果是：

```
Hello parent
Hello child
```

或者：

```
Hello child
Hello parent
```

之所以是这样的结果，就是因为背后的行缓存。执行 `printf("Hello")` 时，因为 `printf` 是向标准输出打印的，因此使用的是行缓存。字符串 `Hello` 没有换行符，所以并没有真正的 I/O 输出。当执行 `fork` 时，子进程会完全复制父进程的内存空间，因此字符串 `Hello` 也存在于子进程的行缓存中。故而最后的输出结果中，无论是父进程还是子进程都有 `Hello` 字符串。

## 2.3 fopen 和 open 标志位对比

C 库的 `fopen` 用于打开文件，其内部实现必然要使用 `open` 系统调用。那么 `fopen` 的各个标志位又对应 `open` 的哪些标志位呢？请看表 2-1。

表 2-1 fopen 标志位和 open 标志位对应表

fopen 标志位	open 标志位	用 途
r	O_RDONLY	以只读方式打开文件
r+	O_RDWR	以读写方式打开文件
w	O_WRONLY O_CREAT O_TRUNC	以写方式打开文件；当文件存在时，将其大小截断为 0；当文件不存在时，创建该文件
w+	O_RDWR O_CREAT O_TRUNC	以读写方式打开文件；当文件存在时，将其大小截断为 0；当文件不存在时，创建该文件



(续)

fopen 标志位	open 标志位	用 途
a	O_WRONLY O_APPEND O_CREAT	以追加写的方式打开文件，当文件不存在时，创建该文件
a+	O_RDWR O_APPEND O_CREAT	以追加读写的方式打开文件，当文件不存在时，创建该文件

表 2-1 是 fopen 常用的标志位，实际上 fopen 还有更多的标志位，这也是很多书籍没有涉及的，具体见表 2-2。

表 2-2 更多的 fopen 和 open 标志位对应

fopen 标志位	open 标志位	用 途
c	无	该文件流在 I/O 操作时不能被取消
e	O_CLOEXEC	当进程执行 exec 时，该文件流会自动关闭
m	无	该文件流通过 mmap 来打开或访问，只支持读取操作
x	O_EXCL	在创建文件时，如果文件已经存在，fopen 则会返回失败而不是打开这个文件
b	无	表示打开的文件是二进制流而不是文本流。该标志目前在 Linux 中是无用的

下面进入 glibc 的源码，查看函数 `_IO_new_file_fopen` 来验证上面的结论。

```
_IO_FILE *
_IO_new_file_fopen (fp, filename, mode, is32not64)
{
    _IO_FILE *fp;
    const char *filename;
    const char *mode;
    int is32not64;

    {
        int oflags = 0, omode;
        int read_write;
        int oprot = 0666;
        int i;
        _IO_FILE *result;
#ifdef _LIBC
        const char *cs;
        const char *last_recognized;
#endif

        if (_IO_file_is_open (fp))
            return 0;
        switch (*mode)
        {
            case 'r':
                omode = O_RDONLY;
                read_write = _IO_NO_WRITES;
                break;
```

```

    case 'w':
        omode = O_WRONLY;
        oflags = O_CREAT|O_TRUNC;
        read_write = _IO_NO_READS;
        break;
    case 'a':
        omode = O_WRONLY;
        oflags = O_CREAT|O_APPEND;
        read_write = _IO_NO_READS|_IO_IS_APPENDING;
        break;
    default:
        __set_errno (EINVAL);
        return NULL;
}
#ifdef _LIBC
    last_recognized = mode;
#endif
for (i = 1; i < 7; ++i)
{
    switch (*++mode)
    {
        case '\\0':
            break;
        case '+':
            omode = O_RDWR;
            read_write &= _IO_IS_APPENDING;
#ifdef _LIBC
            last_recognized = mode;
#endif
            continue;
        case 'x':
            oflags |= O_EXCL;
#ifdef _LIBC
            last_recognized = mode;
#endif
            continue;
        case 'b':
#ifdef _LIBC
            last_recognized = mode;
#endif
            continue;
        case 'm':
            fp->_flags2 |= _IO_FLAGS2_MMAP;
            continue;
        case 'c':
            fp->_flags2 |= _IO_FLAGS2_NOTCANCEL;
            continue;
        case 'e':
#ifdef O_CLOEXEC
            oflags |= O_CLOEXEC;
#endif
            fp->_flags2 |= _IO_FLAGS2_CLOEXEC;
            continue;
        default:

```

```

        /* Ignore. */
        continue;
    }
    break;
}
result = _IO_file_open (fp, filename, omode|oflags, oprot, read_write,
                        is32not64);

```

上面的源代码非常简单，很容易理解。每个 mode 都是 switch 语句的一个 case，oflags 就是要传给 open 的标志位，这就验证了前文的结论。

## 2.4 fdopen 与 fileno

Linux 提供了文件描述符，而 C 库又提供了文件流。在平时的工作中，有时候需要在两者之间进行切换，因此 C 库提供了两个 API：

```

#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
int fileno(FILE *stream);

```

fdopen 用于从文件描述符 fd 生成一个文件流 FILE，而 fileno 则用于从文件流 FILE 得到对应的文件描述符。

查看 fdopen 的实现，其基本工作是创建一个新的文件流 FILE，并建立文件流 FILE 与描述符的对应关系。我们以 fileno 的简单实现，来了解文件流 FILE 与文件描述符 fd 的关系。——因为该函数代码较长，在此就不罗列 C 库的代码了。代码如下：

```

int fileno (_IO_FILE* fp)
{
    CHECK_FILE (fp, EOF);

    if (!(fp->_flags & _IO_IS_FILEBUF) || _IO_fileno (fp) < 0)
    {
        __set_errno (EBADF);
        return -1;
    }

    return _IO_fileno (fp);
}

#define _IO_fileno(FP) ((FP)->_fileno)

```

从 fileno 的实现基本上就可以得知文件流与文件描述符的对应关系。文件流 FILE 保存了文件描述符的值。当从文件流转换到文件描述符时，可以直接通过当前 FILE 保存的值 \_fileno 得到 fd。而从文件描述符转换到文件流时，C 库返回的都是一个重新申请的文件流 FILE，且这个 FILE 的 \_fileno 保存了文件描述符。

因此无论是 fdopen 还是 fileno，关闭文件时，都要使用 fclose 来关闭文件，而不是用 close。因为只有采用此方式，fclose 作为 C 库函数，才会释放文件流 FILE 占用的内存。

## 2.5 同时读写的痛苦

前面介绍过内核的文件描述符实现。在内核中，每一个文件描述符 `fd` 都对应了一个文件管理结构 `struct file`——用于维护该文件描述符的信息，如偏移量等。在第 1 章对 `read` 和 `write` 的源码分析中，可以发现每一次系统调用的 `read` 和 `write` 成功返回后，文件的偏移量都会被更新。

因此，如果程序对同一个文件描述符进行读写操作的话，肯定会得到非期望的结果，示例代码如下：

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

int main(void)
{
    char buf[20];
    int ret;

    FILE *fp = fopen("./tmp.txt", "w+");
    if (!fp) {
        printf("Fail to open file\n");
        return -1;
    }

    ret = fwrite("123", sizeof("123"), 1, fp);
    printf("we write %d member\n", ret);

    memset(buf, 0, sizeof(buf));
    ret = fread(buf, 1, 1, fp);
    printf("We read %s, ret is %d\n", buf, ret);

    fwrite("456", sizeof("456"), 1, fp);

    fclose(fp);

    return 0;
}
```

上面的代码中，利用 `fopen` 的读写模式打开了一个文件流，先写入一个字符串“123”，然后读取一个字节，再写入一个字符串“456”。

大家想想输出结果会是什么呢？`fread` 读取的字符又会是什么呢？是否为“1”呢？请看下面的结果：

```
[fgao@ubuntu chapter2]# ./a.out
we write 1 member
We read , ret is 0
```

为什么 `fread` 什么都没有读取到，返回值是 0 呢？这是因为上面的代码中，`fwrite` 和 `fread` 操作的是同一个文件指针 `fp`，也就是对应的是同一个文件描述符。第一次 `fwrite` 后，

在 tmp.txt 中写入了字符串“123”，同时文件偏移为 3，也就是到了文件尾。进行 fread 操作时，既然操作的是同一个文件描述符，自然会共享同一个文件偏移，那么，从文件尾自然读取不到任何数据。

## 2.6 ferror 的返回值

ferror 用于告诉用户 C 库的文件流 FILE 是否有错误发生。当有错误发生时，ferror 返回非零值，反之则返回 0。那么 ferror 是否会返回不同的错误呢？让我们来看看 ferror 的源码。

```
weak_alias (_IO_ferror, ferror)
int _IO_ferror (fp)
    _IO_FILE* fp;
{
    int result;
    /* 检查文件流的有效性，失败则返回 EOF */
    CHECK_FILE (fp, EOF);
    _IO_flockfile (fp);
    result = _IO_ferror_unlocked (fp);
    _IO_funlockfile (fp);
    return result;
}
```

进入 \_IO\_ferror\_unlocked，代码如下：

```
#define _IO_ferror_unlocked(__fp) (((__fp)->_flags & _IO_ERR_SEEN) != 0)

#define _IO_ERR_SEEN 0x20
```

从源码上可以看出 ferror 有两个返回值：

- ❑ 当文件流 FILE\* fp 非法时，返回 EOF (-1)。
- ❑ 当文件流 FILE\* fp 前面的操作发生错误时，返回 1。

并且由于文件流的错误只是使用一个标志位 \_IO\_ERR\_SEEN 来表示的，因此 ferror 的返回值就不可能针对不同的错误返回不同的值了。

## 2.7 clearerr 的用途

2.6 节中的 ferror 用于检测文件流是否有错误发生，而 clearerr 用于清除文件流的文件结束位和错误位。

查看 clearerr 的实现，代码如下：

```
#define clearerr_unlocked(x) clearerr (x)

void
clearerr_unlocked (fp)
```

```

    FILE *fp;
{
    CHECK_FILE (fp, /*nothing*/);
    _IO_clearerr (fp);
}

#define _IO_clearerr(FP) ((FP)->_flags &= ~(_IO_ERR_SEEN|_IO_EOF_SEEN))

```

可见，`clearerr` 可以清除文件流中的文件结尾标志和错误标志。

但是清除错误标志又有什么用处呢？按照某些资料上的描述，当文件流读到文件尾时，文件流会被设置上 EOF 标志。如果不使用 `clearerr` 清除 EOF 标志，即使有新的数据，也无法读取成功。

让我们写个程序来验证一下：

```

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

int main(void)
{
    FILE *fp = fopen("./tmp.txt", "r");
    if (!fp) {
        printf("Fail to fopen\n");
        return -1;
    }

    while (1) {
        int c = getc(fp);

        if (feof(fp)) {
            printf("reach feof\n");
        }

        return 0;
    }
}

```

为了满足前面所说的测试情况，我们使用 `gdb` 来控制程序，代码如下：

```

31             int c = getc(fp);
(gdb)
33             if (feof(fp)) {
(gdb) n
34                 printf("reach feof\n");

```

现在，文件流 `fp` 已经读到了文件尾，被设置上了 EOF 标志。接下来向 `tmp.txt` 追加一个字母 ‘a’。

```
[fgao@fgao chapter3]#echo "a" >> tmp.txt
```

继续 `gdb`，`getc` 仍然可以继续读取，并获得新数据。

```
(gdb) n
31             int c = getc(fp);
(gdb)
33             if (feof(fp)) {
(gdb) p c
$1 = 97
(gdb) p /c c
$2 = 97 'a'
```

继续下一步：

```
(gdb) n
34             printf("reach feof\n");
```

我们可以发现虽然此时文件流 `fp` 仍然是被设置了 EOF 标志，但是依然能够成功读取数据。这与某些资料的描述不符，这就应对了那句老话“尽信书不如无书”，对于一些资料的结论，不要完全相信，而是要通过自己的实践来验证。

下面回到 `glibc` 的源码，查看 `_IO_getc`，从代码中了解为什么是这样的结果。

```
int
_IO_getc (fp)
    FILE *fp;
{
    int result;
    /* 检查fp */
    CHECK_FILE (fp, EOF);
    _IO_acquire_lock (fp);
    result = _IO_getc_unlocked (fp);
    _IO_release_lock (fp);
    return result;
}

/*
只有定义了IO_DEBUG, CHECK_FILE才会检查_IO_file_flags标志,
当其不为0时, 则返回错误值。对于fgetc即为EOF
*/
#ifdef IO_DEBUG
# define CHECK_FILE(FILE, RET) \
    if ((FILE) == NULL) { MAYBE_SET_EINVAL; return RET; } \
    else { COERCE_FILE(FILE); \
        if (((FILE)->_IO_file_flags & _IO_MAGIC_MASK) != _IO_MAGIC) \
            { MAYBE_SET_EINVAL; return RET; }}
#else
# define CHECK_FILE(FILE, RET) COERCE_FILE (FILE)
#endif
```

从 `glibc` 的源码中可以发现，文件流 `FILE` 的错误标志位只有在打开 `IO_DEBUG` 的情况下才会对后面的 I/O 调用产生影响：在有错误标志位的时候，后面的 I/O 调用都会直接返回 EOF。而一般情况下，`IO_DEBUG` 这个宏是没有定义的。

## 2.8 小心 fgetc 和 getc

fgetc 和 getc 是两个定义得很不友好的函数，其函数名中的 getc 很容易让使用者误以为其返回值是 char 字符。实际上两个函数的接口定义如下：

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
```

两者的返回值都是 int 类型。为什么要用 int 类型作为返回值呢？因为当文件流读到文件尾时，需要返回 EOF 值。C99 标准中规定了 EOF 为一个 int 类型的负数常量，并没有规定具体的值。在 glibc 中，EOF 被定义为 -1 且 char 为有符号数。但是不能排除某些实现将 EOF 定义为其他负值，甚至可能因为不遵守 C99 标准，EOF 的值有可能超过 char 的表示范围。因此，为了代码的健壮性和可移植性，在使用 fgetc 和 getc 时，应使用 int 类型的变量保存其返回值。

## 2.9 注意 fread 和 fwrite 的返回值

fread 和 fwrite 的声明代码如下：

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

这两个函数原型很容易让人产生误解。当看到返回值类型为 size\_t 时，人们很有可能理解为 fread 和 fwrite 会返回成功读取或写入的字节数，然而实际上其返回的是成功读取或写入的个数，即有多少个 size 大小的对象被成功读取或写入了。而参数 nmemb 则用于指示 fread 或 fwrite 要执行的对象个数。

看看下面的示例代码：

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

int main(void)
{
    const char str[] = "123456789";
    FILE *fp = fopen("tmp.txt", "w");
    size_t size = fwrite(str, strlen(str), 1, fp);

    printf("size is %d\n", size);

    fclose(fp);
```



```
    return 0;
}
```

这段代码的输出为：

```
size is 1
```

结果并不是写入的字符串长度 9，而是返回写入的对象个数 1。其原因是参数 `ptr` 指示的是要写入对象的地址，`size` 为每个对象的字节数，`nmemb` 为有多少个要写入的对象。

将上面的代码稍微变换一下，将 `fwrite` 的语句改为：

```
size_t size = fwrite(str, 1, strlen(str), fp);
```

这时程序的输出就变为：

```
size is 9
```

其原因在于，参数 `size` 表示每个对象的字节数是 1 字节，`nmemb` 表示要写入 9 个对象，因此返回值就变为 9 了。

## 2.10 创建临时文件

在项目中经常会需要生成临时文件，用于保存临时数据，创建管道文件、Unix 域 `socket` 等。为了不与已有的文件同名，或者避免与其他临时文件相冲突，有些朋友可能会选择利用进程 `id`、时间戳等来生成临时文件名。其实，C 库已经提供了生成临时文件的接口。下面对生成临时文件的各种方法进行分析对比。先来看看 `tmpnam` 方式，代码如下：

```
#include <stdio.h>

char *tmpnam(char *s);
```

`tmpnam` 会返回一个目前系统不存在的临时文件名。当 `s` 为 `NULL` 时，返回的文件名保存在一个静态的缓存中，因此再次调用 `tmpnam` 时，新生成的文件名会覆盖上一次的结果。当 `s` 不为 `NULL` 时，生成的临时文件名会保存在 `s` 中，因此要求 `s` 至少要有 C 库规定的 `L_tmpnam` 大小。C 库同时还规定 `tmpnam` 产生的临时文件的路径以 `P_tmpdir` 开头——glibc 中 `P_tmpdir` 定义为 `/tmp`。

从上面的描述中可以清楚地发现 `tmpnam` 的缺点：

- ❑ 当 `s` 为 `NULL` 时，`tmpnam` 不是线程安全的。
- ❑ `tmpnam` 生成的临时文件名，必须位于固定的路径下 (`/tmp`)。
- ❑ 使用 `tmpnam` 创建临时文件不是一个原子行为，需要先生成临时文件名，然后调用其他 I/O 函数创建文件。这有可能会在创建文件时，该文件已经存在。

再来看看 `tmpfile` 方式：

```
#include <stdio.h>

FILE *tmpfile(void);
```

`tmpfile` 返回一个以读写模式打开的、唯一的临时文件流指针。当文件指针关闭或程序正常结束时，该临时文件会被自动删除。

`tmpfile` 直接返回临时的文件流指针——这个自然避免了 `tmpnam` 中潜在的线程安全问题，同时还避免了将生成文件名和创建文件分为两个步骤来执行的行为。那么 `tmpfile` 是否真的实现了原子地创建临时文件？让我们看一下 `tmpfile` 的实现，代码如下：

```
FILE *
tmpfile (void)
{
    char buf[FILENAME_MAX];
    int fd;
    FILE *f;

    if (__path_search (buf, FILENAME_MAX, NULL, "tmpf", 0))
        return NULL;
    int flags = 0;
#ifdef FLAGS
    flags = FLAGS;
#endif
    fd = __gen_tempname (buf, 0, flags, __GT_FILE);
    if (fd < 0)
        return NULL;

    /* Note that this relies on the UNIX semantics that
       a file is not really removed until it is closed. */
    (void) __unlink (buf);

    if ((f = __fdopen (fd, "w+b")) == NULL)
        __close (fd);

    return f;
}
```

乍一看，`tmpfile` 是通过 `__path_search` 先产生临时文件名，然后再创建该文件，最后通过文件句柄生成文件流指针。这样的过程看上去好像并不是原子的。下面，让我们深入到 `__gen_tempname` 中一探究竟。

```
case __GT_FILE:
    fd = __open (tmpl,
                (flags & ~O_ACCMODE)
                | O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    break;
```

在创建临时文件时，C 库使用了 `open` 函数的 `O_CREAT` 和 `O_EXCL` 标志组合，这点保证了文件的原子性创建，从而使 `tmpfile` 创建临时文件的行为是原子的。但 `tmpfile` 也有一个缺点，与 `tmpnam` 相同，这个临时文件只能生成在固定的路径下（`/tmp`），并且其有可能

因为文件名称冲突而失败返回 NULL。

那么，有没有可以给临时文件指定目录的方法呢？下面请看 mkstemp，代码如下：

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

```
int mkstemp(char *template);
```

mkstemp 会根据 template 创建并打开一个独一无二的临时文件。template 的最后 6 个字符必须是“XXXXXX”。glibc 库会生成一个独一无二的后缀来替换“XXXXXX”，因此要求 template 必须是可以修改的。

mkstemp 执行成功后会返回创建的临时文件的文件描述符，失败时则返回 -1。下面看一下 mkstemp 的实现。

```
int #mkstemp (template)
    char *template;
{
    return __gen_tempname (template, 0, 0, __GT_FILE);
}
```

进入 \_\_gen\_tempname 后：

```
int #__gen_tempname (char *tmpl, int suffixlen, int flags, int kind)
{
    int len;
    char *XXXXXX;
    static uint64_t value;
    uint64_t random_time_bits;
    unsigned int count;
    int fd = -1;
    int save_errno = errno;
    struct_stat64 st;

#define ATTEMPTS_MIN (62 * 62 * 62)
    /* The number of times to attempt to generate a temporary file. To
       conform to POSIX, this must be no smaller than TMP_MAX. */
    #if ATTEMPTS_MIN < TMP_MAX
        unsigned int attempts = TMP_MAX;
    #else
        unsigned int attempts = ATTEMPTS_MIN;
    #endif

    /* 检查template的合法性，检查长度及结尾的XXXXXX字符 */
    len = strlen (tmpl);
    if (len < 6 + suffixlen || memcmp (&tmpl[len - 6 - suffixlen], "XXXXXX", 6))
    {
        __set_errno (EINVAL);
        return -1;
    }

    /* 得到结尾XXXXXX起始位置 */
    XXXXXX = &tmpl[len - 6 - suffixlen];
```

```

/* 得到“随机”数据 */
#ifdef RANDOM_BITS
    RANDOM_BITS (random_time_bits);
#else
    #if HAVE_GETTIMEOFDAY || _LIBC
    {
        struct timeval tv;
        __gettimeofday (&tv, NULL);
        random_time_bits = ((uint64_t) tv.tv_usec << 16) ^
            tv.tv_sec;
    }
    #else
        random_time_bits = time (NULL);
    #endif
#endif
/* 根据上面的伪随机数和进程pid生成value */
value += random_time_bits ^ __getpid ();
/*
根据value得到唯一的临时文件名，如有重复则加上7777继续。
最多重复attempts次。
*/
for (count = 0; count < attempts; value += 7777, ++count)
{
    uint64_t v = value;

    /*
letters是26个英文大小写加上10个阿拉伯数字，为62个大小的字符数组。因此使用62作为除数，
以得到随机字符。
*/
    XXXXXX[0] = letters[v % 62];
    v /= 62;
    XXXXXX[1] = letters[v % 62];
    v /= 62;
    XXXXXX[2] = letters[v % 62];
    v /= 62;
    XXXXXX[3] = letters[v % 62];
    v /= 62;
    XXXXXX[4] = letters[v % 62];
    v /= 62;
    XXXXXX[5] = letters[v % 62];

    switch (kind)
    {
        case __GT_FILE:
            /* 这是mkstemp的情况，利用O_CREAT|O_EXCL创建唯一文件 */
            fd = __open (tmpl,
                (flags & ~O_ACCMODE)
                | O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
            break;
    }

    if (fd >= 0)
    {
        /* 成功创建了文件，恢复原来的errno，并返回创建的文件描述符fd */
        __set_errno (save_errno);
    }
}

```

```

        return fd;
    }
    else if (errno != EEXIST) {
        /* 如失败的原因不是因为文件已经存在的时候，则直接返回。*/
        return -1;
    }
    /* 如果是其他原因，则会重新生成新的文件名，并再次尝试重建 */
}

/* 将errno设置为EEXIST，即文件已经存在*/
__set_errno (EEXIST);
return -1;
}

```

综上所述，在需要使用临时文件时，不推荐使用 `tmpnam`，而要用 `tmpfile` 和 `mkstemp`。前者的局限在于不能指定路径，并且在文件名称冲突时会返回失败。后者可以由调用者来指定路径，并且在文件名称冲突时，会自动重新生成并重试。

除了上面介绍的几种方法，Linux 环境还提供了这些接口的一些变种：`tempnam`、`mkostemp`、`mkstemps` 等，分别对其原始形态进行了扩展，详细区别可以直接查看 Linux 手册。