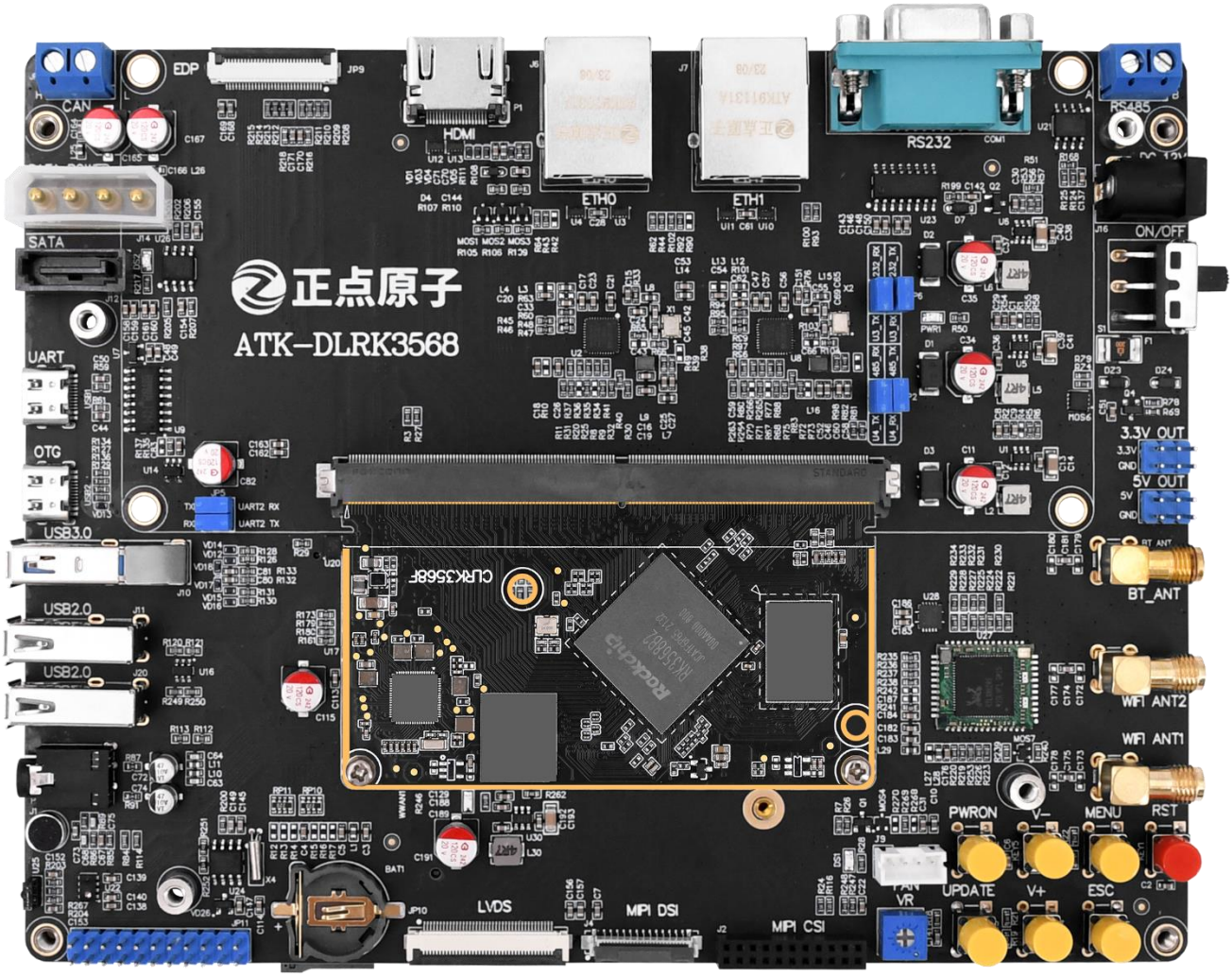
Linux C 应用编程参考手册 V1.0

-正点原子 ATK-DLRK3568 开发板文档



正点原子公司名称 ：广州市星翼电子科技有限公司

原子哥在线教学平台：www.yuanzige.com

开源电子网 / 论坛 ：http://www.openedv.com/forum.php

正点原子淘宝店铺 ：https://openedv.taobao.com

正点原子官方网站 ：www.alientek.com

正点原子 B 站视频 ：https://space.bilibili.com/394620890

电话：020-38271790 传真：020-36773971

请关注正点原子公众号，资料发布更新我们会通知。

请下载原子哥 APP，数千讲视频免费学习，更快更流畅。

扫码关注正点原子公众号

扫码下载“原子哥”APP

文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初始版本	邓涛	邓涛	2023.06.06

目录

前言

首先欢迎大家阅读本文档，本文档为正点原子 Linux 团队所编写的《Linux C 应用编程参考手册》，基于 Linux 系统的 C 语言应用编程学习指南，主要讲解 Linux C 语言应用编程基础知识，包括文件 IO 操作、文件高级 IO、标准 IO、文件属性、系统信息、进程、线程、信号以及线程同步等内容。本文档定义为基础入门文档，其中并不会对所讲解之内容进行深入剖析，适用于 Linux 应用编程初学者，如果您已有多年的 Linux 应用编程经验，熟悉各种应用场合的 Linux 应用开发，那本文档并不适合您阅读。对于 Linux 应用编程小白，如果您对 Linux 应用编程感兴趣亦或想将来从事一份与之相关的工作，那么本文档是一个不错的选择！

虽然本文档定义为基础入门文档，但是对读者也有一定的要求，当然要求并不过分，只需要大家有一定的 C 语言编程基础、熟练使用 Linux 操作系统（譬如 Ubuntu 系统）即可！

对于 Linux 应用编程，市面上有很多相关书籍，其中有一些笔者认为写的非常好的书籍！这里给大家推荐三本，分别为：《**UNIX 环境高级编程**》、《**Linux/UNIX 系统编程手册**》（分为上册和下册）、《**UNIX 网络编程**》；这三本书籍都非常不错，内容涵盖广、知识点讲解到位、深入剖析，不管是初学者还是拥有多年编程经验的工程师，都可以拿来阅读！

最后感谢大家阅读本书，您对我们的支持与信任，是我们永往直前的信心和勇气。另外，正点原子为大家提供了一个技术交流平台（开源电子网）：http://www.openedv.com/forum.php，如果您在学习过程中遇到了问题或者存在一些疑问，可以通过该平台与众多 Linux 开发者一起交流、讨论；同时，由于笔者知识水平有限，文中错漏与不足之处在所难免，恳请各位读者不吝赐教。

第一章 应用编程概念

对于大多数首次接触 Linux 应用编程的读者来说，可能对应用编程（也可称为系统编程）这个概念并不太了解，所以在正式学习 Linux 应用编程之前，笔者有必要向大家介绍这些简单基本的概念，从整体上认识到应用编程为何物？与驱动编程、裸机编程有何不同？

了解本章所介绍的内容是掌握应用编程的先决条件，所以本章主要内容便是对 Linux 应用编程进行一个简单地介绍，让读者对此有一个基本的认识。

本章将会讨论如下主题内容。

- 何为系统调用；
- 何为库函数；
- 应用程序的 main()函数；
- 应用程序开发环境的介绍。

1.1 系统调用

系统调用（system call）其实是 Linux 内核提供给应用层的应用编程接口（API），是 Linux 应用层进入内核的入口。不止 Linux 系统，所有的操作系统都会向应用层提供系统调用，应用程序通过系统调用来使用操作系统提供的各种服务。

通过系统调用，Linux 应用程序可以请求内核以自己的名义执行某些事情，譬如打开磁盘中的文件、读写文件、关闭文件以及控制其它硬件外设。

通过系统调用 API，应用层可以实现与内核的交互，其关系可通过下图简单描述：

图 1.1.1 内核、系统调用与应用程序

内核提供了一系列的服务、资源、支持一系列功能，应用程序通过调用系统调用 API 函数来使用内核提供的服务、资源以及各种各样的功能，如果大家接触过其它操作系统编程，想必对此并不陌生，譬如 Windows 应用编程，操作系统内核一般都会向应用程序提供应用编程接口 API，否则我们将我无法使用操作系统。

应用编程与裸机编程、驱动编程有什么区别？

在学习应用编程之前，相信大家都有过软件开发经验，譬如 51、STM32 等单片机软件开发、以及嵌入式 Linux 硬件平台下的驱动开发等，51、STM32 这类单片机的软件开发通常是裸机程序开发，并不会涉及到操作系统的概念，那应用编程与裸机编程以及驱动开发有什么区别呢？

就拿嵌入式 Linux 硬件平台下的软件开发来说，我们大可将编程分为三种，分别为裸机编程、Linux 驱动编程以及 Linux 应用编程。首先对于裸机编程这个概念来说很好理解，一般把没有操作系统支持的编程环境称为裸机编程环境，譬如单片机上的编程开发，编写直接在硬件上运行的程序，没有操作系统支持；狭义上 Linux 驱动编程指的是基于内核驱动框架开发驱动程序，驱动开发工程师通过调用 Linux 内核提供的接口完成设备驱动的注册，驱动程序负责底层硬件操作相关逻辑，如果学习过 Linux 驱动开发的读者，想必对此并不陌生；而 Linux 应用编程（系统编程）则指的是基于 Linux 操作系统的应用编程，在应用程序中通过调用系统调用 API 完成应用程序的功能和逻辑，应用程序运行于操作系统之上。通常在操作系统下有两种不同的状态：内核态和用户态，应用程序运行在用户态、而内核则运行在内核态。

关于应用编程这个概念，以上给大家解释得很清楚了，笔者以实现点亮一个 LED 功能为例，给大家简单地说明三者之间的区别，LED 裸机程序如下所示：

示例代码 1.1.1 LED 裸机程序

```
static void led_on(void)
{
```

```
    /* 点亮 LED 硬件操作代码 */
}

static void led_off(void)
{
    /* 熄灭 LED 硬件操作代码 */
}

int main(void)
{
    /* 用户逻辑 */
    for ( ; ; ) {

        led_on();          //点亮 LED
        delay();           //延时
        led_off();         //熄灭 LED
        delay();           //延时
    }
}
```

可以看到在裸机程序当中，LED 硬件操作代码与用户逻辑代码全部都是在同一个源文件（同一个工程）中实现的，硬件操作代码与用户逻辑代码没有隔离，没有操作系统支持，代码编译之后直接在硬件平台运行，俗称“裸跑”。我们再来看一个 Linux 系统下的 LED 驱动程序示例代码，如下所示：

示例代码 1.1.2 Linux 下 LED 驱动程序

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/of_gpio.h>
#include <linux/delay.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>

static void led_on(void)
{
    /* 点亮 LED 硬件操作代码 */
}

static void led_off(void)
{
    /* 熄灭 LED 硬件操作代码 */
}

static int led_open(struct inode *inode, struct file *filp)
{
    /* 打开设备时需要做的事情 */
```

```
}

static ssize_t led_write(struct file *filp, const char __user *buf,
                         size_t size, loff_t *offt)
{
    int flag;

    /* 获取应用层 write 的数据,存放在 flag 变量 */
    if (copy_from_user(&flag, buf, size))
        return -EFAULT;

    /* 判断用户写入的数据,如果是 0 则熄灭 LED,如果是非 0 则点亮 LED */
    if (flag)
        led_on();
    else
        led_off();

    return 0;
}

static int led_release(struct inode *inode, struct file *filp)
{
    /* 关闭设备时需要做的事情 */
}

static struct file_operations led_fops = {
    .owner    = THIS_MODULE,
    .open     = led_open,
    .write    = led_write,
    .release  = led_release,
};

static int led_probe(struct platform_device *pdev)
{
    /* 驱动加载时需要做的事情 */
}

static int led_remove(struct platform_device *pdev)
{
    /* 驱动卸载时需要做的事情 */
}

static const struct of_device_id led_of_match[] = {
```

```
    { .compatible = "alientek,led", },
    { /* sentinel */ },
};
MODULE_DEVICE_TABLE(of, led_of_match);

static struct platform_driver led_driver = {
    .driver = {
        .owner          = THIS_MODULE,
        .name           = "led",
        .of_match_table = led_of_match,
    },
    .probe  = led_probe,
    .remove = led_remove,
};
module_platform_driver(led_driver);

MODULE_DESCRIPTION("LED Driver");
MODULE_LICENSE("GPL");
```

以上并不是一个完整的 LED 驱动代码，如果没有接触过 Linux 驱动开发的读者，看不懂也没有关系，并无大碍，此驱动程序使用了最基本的字符设备驱动框架编写而成，非常简单；led_fops 对象中提供了 open、write、release 方法，当应用程序调用 open 系统调用打开此 LED 设备时会执行到 led_open 函数，当调用 close 系统调用关闭 LED 设备时会执行到 led_release 函数，而调用 write 系统调用时会执行到 led_write 函数，此驱动程序的设定是当应用层调用 write 写入 0 时熄灭 LED，write 写入非 0 时点亮 LED。

驱动程序属于内核的一部分，当操作系统启动的时候会加载驱动程序，可以看到 LED 驱动程序中仅仅实现了点亮/熄灭 LED 硬件操作相关逻辑代码，应用程序可通过 write 这个系统调用 API 函数控制 LED 亮灭；接下来我们看看 Linux 系统下的 LED 应用程序示例代码，如下所示：

示例代码 1.1.3 Linux 下 LED 应用程序

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

int main(int argc, char **argv)
{
    int fd;
    int data;

    fd = open("/dev/led", O_WRONLY);//打开 LED 设备(假定 LED 的设备文件为/dev/led)
    if (0 > fd)
        return -1;

    for ( ; ; ) {
```

```
        data = 1;
        write(fd, &data, sizeof(data)); //写 1 点亮 LED
        sleep(1);           //延时 1 秒

        data = 0;
        write(fd, &data, sizeof(data)); //写 0 熄灭 LED
        sleep(1);           //延时 1 秒
    }

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

此应用程序也非常简单，仅只需实现用户逻辑代码即可，循环点亮、熄灭 LED，并不需要实现硬件操作相关，示例代码中调用了 open、write、close 这三个系统调用 API 接口，open 和 close 分别用于打开/关闭 LED 设备，write 写入数据传给 LED 驱动，传入 0 熄灭 LED，传入非 0 点亮 LED。

LED 应用程序与 LED 驱动程序是分隔、分离的，它们单独编译，它们并不是整合在一起的，应用程序运行在操作系统之上，有操作系统支持，应用程序处于用户态，而驱动程序处于内核态，与纯粹的裸机程序存在着质的区别。Linux 应用开发与驱动开发是两个不同的方向，将来在工作当中也会负责不同的任务、解决不同的问题。

关于本小节系统调用概念相关的内容就介绍到这里了，接下来向大家介绍库函数。

1.2 库函数

前面给大家介绍了系统调用，系统调用是内核直接向应用层提供的应用编程接口，譬如 open、write、read、close 等，关于这些系统调用后面会给大家进行详细介绍。编写应用程序除了使用系统调用之外，我们还可以使用库函数，本小节来聊一聊库函数。

库函数也就是 C 语言库函数，C 语言库是应用层使用的一套函数库，在 Linux 下，通常以动态（.so）库文件的形式提供，存放在根文件系统/lib 目录下，C 语言库函数构建于系统调用之上，也就是说库函数其实是由系统调用封装而来的，当然也不能完全这么说，原因在于有些库函数并不调用任何系统调用，譬如一些字符串处理函数 strlen()、strcat()、memcpy()、memset()、strchr()等等；而有些库函数则会使用系统调用来帮它完成实际的操作，譬如库函数 fopen 内部调用了系统调用 open()来帮它打开文件、库函数 fread()就利用了系统调用 read()来完成读文件操作、fwrite()就利用了系统调用 write()来完成写文件操作。

Linux 系统内核提供了一系列的系统调用供应用层使用，我们直接使用系统调用就可以了呀，那为何还要设计出库函数呢？事实上，有些系统调用使用起来并不是很方便，于是就出现了 C 语言库，这些 C 语言库函数的设计是为了提供比底层系统调用更为方便、更为好用、且更具有可移植性的调用接口。

来看一看它们之间的区别：

- 库函数是属于应用层，而系统调用是内核提供给应用层的编程接口，属于系统内核的一部分；
- 库函数运行在用户空间，调用系统调用会由用户空间（用户态）陷入到内核空间（内核态）；
- 库函数通常是有缓存的，而系统调用是无缓存的，所以在性能、效率上，库函数通常要优于系统调用；
- 可移植性：库函数相比于系统调用具有更好的可移植性，通常对于不同的操作系统，其内核向应用层提供的系统调用往往都是不同，譬如系统调用的定义、功能、参数列表、返回值等往往都是不一样的；而对于 C 语言库函数来说，由于很多操作系统都实现了 C 语言库，C 语言库在不同的操作

系统之间其接口定义几乎是一样的，所以库函数在不同操作系统之间相比于系统调用具有更好的可移植性。

以上便上它们之间一个大致的区别，从实现者的角度来看，系统调用与库函数之间有根本的区别，但从用户使用角度来看，其区别并不重要，它们都是 C 语言函数。在实际应用编程中，库函数和系统调用都会使用到，所以对于我们来说，直接把它们当做是 C 函数即可，知道你自己调用的函数是系统调用还是库函数即可，不用太过于区分它们之间的差别。

所以应用编程简单点来说就是：开发 Linux 应用程序，通过调用内核提供的系统调用或使用 C 库函数来开发具有相应功能的应用程序。

1.3 标准 C 语言函数库

在 Linux 系统下，使用的 C 语言库为 GNU C 语言函数库（也叫作 glibc，其网址为 http://www.gnu.org/software/libc/），作为 Linux 下的标准 C 语言函数库。

进入到 http://www.gnu.org/software/libc/网址，如下所示：

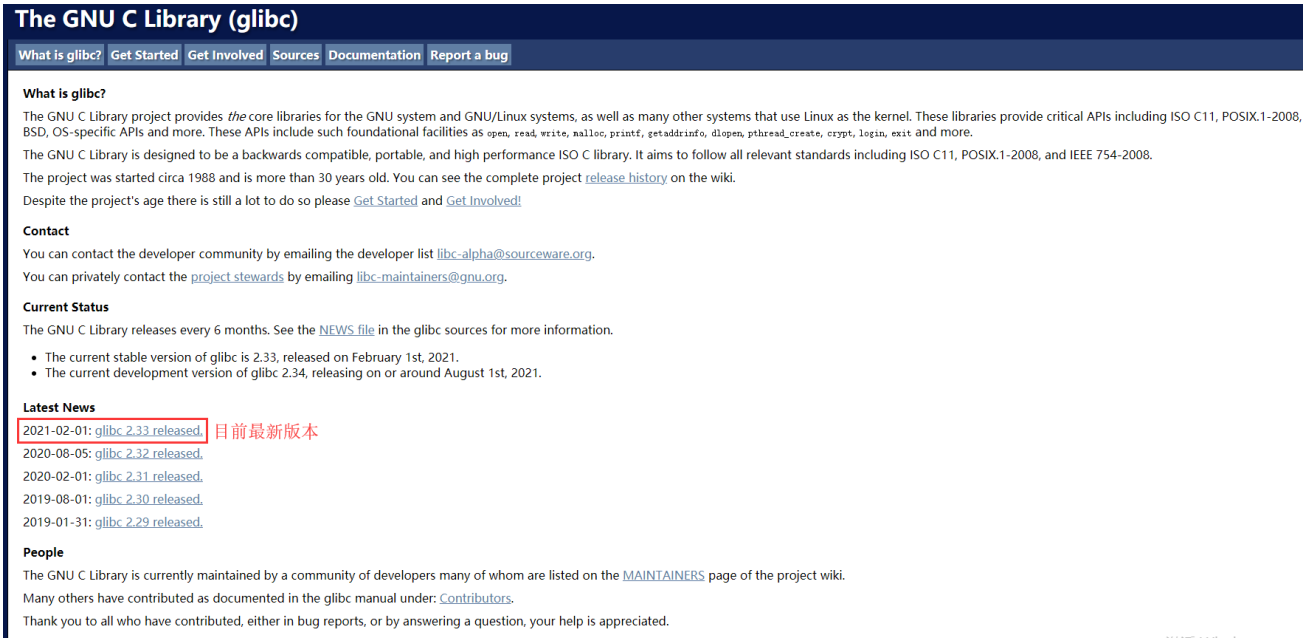


图 1.3.1 glibc 官网

点击上面的 Sources 选项可以查看它的源码实现：

图 1.3.2 获取源码的方式

glibc 源码的获取方式很简单，直接直接从 git 仓库下载，也可以通过 ftp 下载，如果大家有兴趣、或者想要了解某一个库函数它的具体实现，那么就可以获取到它源码来进行分析，好了，这里就不再多说了！

确定 Linux 系统的 glibc 版本

前面提到过了，C 语言库是以动态库文件的形式提供的，通常存放在/lib 目录，它的命名方式通常是 libc.so.6，不过这个是一个软链接文件，它会链接到真正的库文件。

进入到 Ubuntu 系统的/lib 目录下，笔者使用的 Ubuntu 版本为 20.04，在我的/lib 目录下并没有发现 libc.so.6 这个文件，其实是在/lib/x86_64-linux-gnu 目录下，进入到该目录：

```
tgg@tgg-virtual-machine:/lib/x86_64-linux-gnu$
tgg@tgg-virtual-machine:/lib/x86_64-linux-gnu$ ls -lh libc.so.6
lrwxrwxrwx 1 root root 12 2月  25  2022 libc.so.6 -> libc-2.31.so
tgg@tgg-virtual-machine:/lib/x86_64-linux-gnu$
```

图 1.3.3 libc.so.6 文件

可以看到 libc.so.6 链接到了 libc-2.31.so 库文件，2.31 表示的就是这个 glibc 库的版本号为 2.31。除此之外，我们还可以直接运行该共享库来获取到它的信息，如下所示：

```
tgg@tgg-virtual-machine:/lib/x86_64-linux-gnu$
tgg@tgg-virtual-machine:/lib/x86_64-linux-gnu$ ./libc.so.6
GNU C Library (Ubuntu GLIBC 2.31-0ubuntu9.7) stable release version 2.31
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 9.3.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
tgg@tgg-virtual-machine:/lib/x86_64-linux-gnu$
```

图 1.3.4 确定 glibc 版本号

从打印信息可以看到，笔者所使用的 Ubuntu 系统对应的 glibc 版本号为 2.31。

1.4 main 函数

对学习过 C 语言编程的读者来说，譬如单片机编程、Windows 应用编程等，main 函数想必大家再熟悉不过了，很多编程开发都是以 main 函数作为程序的入口函数，同样在 Linux 应用程序中，main 函数也是作

为应用程序的入口函数存在，main 函数的形参一般会有两种写法，如果执行应用程序无需传参，则可以写成如下形式：

示例代码 1.4.1 main 函数写法之无传参

```
int main(void)
{
    /* 代码 */
}
```

如果在执行应用程序的时候需要向应用程序传递参数，则写法如下：

示例代码 1.4.2 main 函数写法之有传参

```
int main(int argc, char **argv)
{
    /* 代码 */
}
```

argc 形参表示传入参数的个数，包括应用程序自身路径和程序名，譬如运行当前目录下的 hello 可执行文件，并且传入参数，如下所示：

```
./hello 112233
```

那么此时参数个数为 2，并且这些参数都是作为字符串的形式传递给 main 函数:

argv[0]等于"./hello"

argv[1]等于"112233"

有传参时 main 函数的写法并不只有这一种，只是这种写法最常用，对于其它的写法，后面学习过程中如果遇到了再给大家进行讲解，这里暂时先不去管。

1.5 本书使用的开发环境

本书以 Ubuntu 操作系统作为应用编程实操环境，在学习之前，您需要先安装好 Ubuntu 操作系统，建议使用虚拟机进行安装，Ubuntu 版本建议与本书保持一致（Ubuntu 20.04）；除此之外，您还需要在 Ubuntu 系统下安装一款 IDE 软件（除非您喜欢用 vi），用于编辑代码，推荐使用 vscode；vscode 是一款简化且高效的代码编辑器、免费而且功能强大，支持第三方插件、有完善的插件生态，插件功能种类繁多。

关于 Ubuntu 系统的安装以及 vscode 软件的安装与使用，本书不做介绍。

第二章 文件 I/O 基础

本章给大家介绍 Linux 应用编程中最基础的知识，即文件 I/O（Input、Outout），文件 I/O 指的是对文件的输入/输出操作，说白了就是对文件的读写操作；Linux 下一切皆文件，文件作为 Linux 系统设计思想的核心理念，在 Linux 系统下显得尤为重要，所以对文件的 I/O 操作既是基础也是最重要的部分。

本章将向大家介绍 Linux 系统下文件描述符的概念，随后会逐一讲解构成通用 I/O 模型的系统调用，譬如打开文件、关闭文件、从文件中读取数据和向文件中写入数据以及这些系统调用涉及的参数等内容。

本章将会讨论如下主题内容。

- 文件描述符的概念；
- 打开文件 open()、关闭文件 close()；
- 写文件 write()、读文件 read()；
- 文件读写位置偏移量。

2.1 一个简单的文件 IO 示例

本章主要介绍文件 IO 操作相关系统调用，一个通用的 IO 模型通常包括打开文件、读写文件、关闭文件这些基本操作，主要涉及到 4 个函数：open()、read()、write()以及 close()，我们先来看一个简单地文件读写示例，应用程序代码如下所示：

示例代码 2.1.1 一个简单地文件 IO 示例

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

int main(void)
{
    char buff[1024];
    int fd1, fd2;
    int ret;

    /* 打开源文件 src_file(只读方式) */
    fd1 = open("./src_file", O_RDONLY);
    if (-1 == fd1)
        return fd1;

    /* 打开目标文件 dest_file(只写方式) */
    fd2 = open("./dest_file", O_WRONLY);
    if (-1 == fd2) {
        ret = fd2;
        goto out1;
    }

    /* 读取源文件 1KB 数据到 buff 中 */
    ret = read(fd1, buff, sizeof(buff));
    if (-1 == ret)
        goto out2;

    /* 将 buff 中的数据写入目标文件 */
    ret = write(fd2, buff, sizeof(buff));
    if (-1 == ret)
        goto out2;

    ret = 0;

out2:
    /* 关闭目标文件 */
```

```
    close(fd2);

out1:
    /* 关闭源文件 */
    close(fd1);
    return ret;
}
```

这段代码非常简单明了，代码所要实现的功能在注释当中已经描述得很清楚了，从源文件 src_file 中读取 1KB 数据，然后将其写入到目标文件 dest_file 中（这里假设当前目录下这两个文件都是存在的）；在进行读写操作之前，首先调用 open 函数将源文件和目标文件打开，成功打开之后再调用 read 函数从源文件中读取 1KB 数据，然后再调用 write 函数将这 1KB 数据写入到目标文件中，至此，文件读写操作就完成了，读写操作完成之后，最后调用 close 函数关闭源文件和目标文件。

接下来我们给大家详细介绍这些函数以及相关的内容。

2.2 文件描述符

调用 open 函数会有一个返回值，譬如示例代码 2.1.1 中的 fd1 和 fd2，这是一个 int 类型的数据，在 open 函数执行成功的情况下，会返回一个非负整数，该返回值就是一个文件描述符（file descriptor），这说明文件描述符是一个非负整数；对于 Linux 内核而言，所有打开的文件都会通过文件描述符进行索引。

当调用 open 函数打开一个现有文件或创建一个新文件时，内核会向进程返回一个文件描述符，用于指代被打开的文件，所有执行 IO 操作的系统调用都是通过文件描述符来索引到对应的文件，譬如示例代码 2.1.1 中，当调用 read/write 函数进行文件读写时，会将文件描述符传送给 read/write 函数，所以在代码中，fb1 就是源文件 src_file 被打开时所对应的文件描述符，而 fd2 则是目标文件 dest_file 被打开时所对应的文件描述符。

一个进程可以打开多个文件，但是在 Linux 系统中，一个进程可以打开的文件数是有限制，并不是可以无限制打开很多的文件，大家想一想便可以知道，打开的文件是需要占用内存资源的，文件越大、打开的文件越多那占用的内存就越多，必然会对整个系统造成很大的影响，如果超过进程可打开的最大文件数限制，内核将会发送警告信号给对应的进程，然后结束进程；在 Linux 系统下，我们可以通过 ulimit 命令来查看进程可打开的最大文件数，用法如下所示：

```
ulimit -n
```

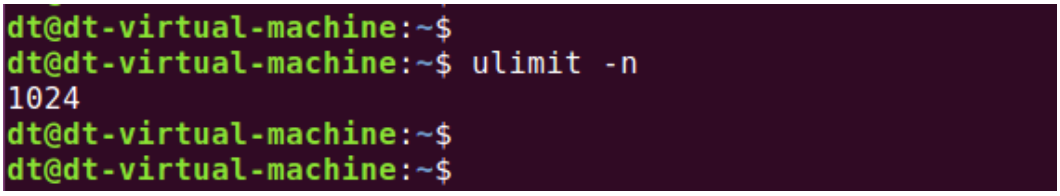


图 2.2.1 查看进程可打开的最大文件数

该最大值默认情况下是 1024，也就意味着一个进程最多可以打开 1024 个文件，当然这个限制数其实是可以设置的，这个就先不给大家介绍了，当然除了进程有最大文件数限制外，其实对于整个 Linux 系统来说，也有最大限制，那么关于这些问题，如果后面的章节内容中涉及到了再给大家进行介绍。

所以对于一个进程来说，文件描述符是一种有限资源，文件描述符是从 0 开始分配的，譬如说进程中第一个被打开的文件对应的文件描述符是 0、第二个文件是 1、第三个文件是 2、第 4 个文件是 3……以此类推，所以由此可知，文件描述符数字最大值为 1023（0~1023）。每一个被打开的文件在同一个进程中都有一个唯一的文件描述符，不会重复，如果文件被关闭后，它对应的文件描述符将会被释放，那么这个文件描述符将可以再次分配给其它打开的文件、与对应的文件绑定起来。

每次给打开的文件分配文件描述符都是从最小的没有被使用的文件描述符（0~1023）开始，当之前打开的文件被关闭之后，那么它对应的文件描述符会被释放，释放之后也就成为了一个没有被使用的文件描述符了。

当我们在程序中，调用 open 函数打开文件的时候，分配的文件描述符一般都是从 3 开始，这里大家可能要问了，上面不是说从 0 开始的吗，确实是如此，但是 0、1、2 这三个文件描述符已经默认被系统占用了，分别分配给了系统标准输入（0）、标准输出（1）以及标准错误（2），关于这个问题，这里不便给大家说太多，毕竟这是后面的内容，这里只是给大家提一下，后面遇到了再具体讲解。

Tips：Linux 系统下，一切皆文件，也包括各种硬件设备，使用 open 函数打开任何文件成功情况下便会返回对应的文件描述符 fd。每一个硬件设备都会对应于 Linux 系统下的某一个文件，把这类文件称为设备文件。所以设备文件对应的其实是某一硬件设备，应用程序通过对设备文件进行读写等操作、来使用、操控硬件设备，譬如 LCD 显示器、串口、音频、键盘等。

标准输入一般对应的是键盘，可以理解为 0 便是打开键盘对应的设备文件时所得到的文件描述符；标准输出一般指的是 LCD 显示器，可以理解为 1 便是打开 LCD 设备对应的设备文件时所得到的文件描述符；而标准错误一般指的也是 LCD 显示器。

2.3 open 打开文件

在 Linux 系统中要操作一个文件，需要先打开该文件，得到文件描述符，然后再对文件进行相应的读写操作（或其他操作），最后在关闭该文件；open 函数用于打开文件，当然除了打开已经存在的文件之外，还可以创建一个新的文件，函数原型如下所示：

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

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

在 Linux 系统下，可以通过 man 命令（也叫 man 手册）来查看某一个 Linux 系统调用的帮助信息，man 命令可以将该系统调用的详细信息显示出来，譬如函数功能介绍、函数原型、参数、返回值以及使用该函数所需包含的头文件等信息；man 更像是一份帮助手册，所以也把它称为 man 手册，当我们需要查看某个系统调用的功能介绍、使用方法时，不用在上网到处查找，直接通过 man 命令便可以搞定，man 命令用法如下所示：

```
man 2 open          #查看 open 函数的帮助信息
```

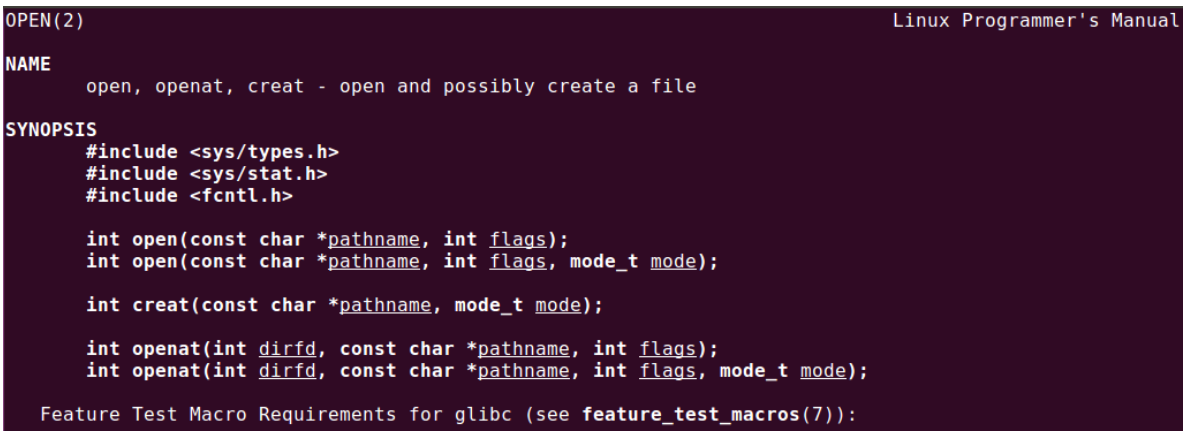


图 2.3.1 查看 open 函数帮助信息

Tips：man 命令后面跟着两个参数，数字 2 表示系统调用，man 命令除了可以查看系统调用的帮助信息外，还可以查看 Linux 命令（对应数字 1）以及标准 C 库函数（对应数字 3）所对应的帮助信息；最后一个参数 open 表示需要查看的系统调用函数名。

由于篇幅有限，此截图只是其中一部分内容，从图中可知，open 函数有两种原型？这是为什么呢？关于这个问题笔者一开始也不理解，大家都知道 C 语言是不支持重载的，那既然这样，只有一种解释了，那就是可变参函数；对于 C 语言中的可变参函数，对此不了解的朋友可以自行百度，本文档不作说明！

所以由此可知，在应用程序中调用 open 函数即可传入 2 个参数（pathname、flags）、也可传入 3 个参数（pathname、flags、mode），但是第三个参数 mode 需要在第二个参数 flags 满足条件时才会有效，稍后将对此进行说明；从图 2.3.1 可知，在应用程序中使用 open 函数时，需要包含 3 个头文件“#include <sys/types.h>”、“#include <sys/stat.h>”、“#include <fcntl.h>”。

函数参数和返回值含义如下：

pathname：字符串类型，用于标识需要打开或创建的文件，可以包含路径（绝对路径或相对路径）信息，譬如："./src_file"（当前目录下的 src_file 文件）、"/home/dengtao/hello.c"等；如果 pathname 是一个符号链接，会对其进行解引用。

flags：调用 open 函数时需要提供的标志，包括文件访问模式标志以及其它文件相关标志，这些标志使用宏定义进行描述，都是常量，open 函数提供了非常多的标志，我们传入 flags 参数时既可以单独使用某一个标志，也可以通过位或运算（|）将多个标志进行组合。这些标志介绍如下：

表 2.3.1 open 函数 flags 参数值介绍

标志	用途	说明
O_RDONLY	以只读方式打开文件	这三个是文件访问权限标志，传入的 flags 参数中必须要包含其中一种标志，而且只能包含一种，打开的文件只能按照这种权限来操作，譬如使用了 O_RDONLY 标志，就只能对文件进行读取操作，不能写操作。
O_WRONLY	以只写方式打开文件	
O_RDWR	以可读可写方式打开文件	
O_CREAT	如果 pathname 参数指向的文件不存在则创建此文件	使用此标志时，调用 open 函数需要传入第 3 个参数 mode，参数 mode 用于指定新建文件的访问权限，稍后将对此进行说明。 **open 函数的第 3 个参数只有在使用了 O_CREAT 或 O_TMPFILE 标志时才有效。**
O_DIRECTORY	如果 pathname 参数指向的不是一个目录，则调用 open 失败	
O_EXCL	此标志一般结合 O_CREAT 标志一起使用，用于专门创建文件。 在 flags 参数同时使用到了 O_CREAT 和 O_EXCL 标志的情况下，如果 pathname 参数指向的文件已经存在，则 open 函数返回错误。	可以用于测试一个文件是否存在，如果不存在则创建此文件，如果存在则返回错误，这使得测试和创建两者成为一个原子操作；关于原子操作，在后面的内容当中将会对此进行说明。

O_NOFOLLOW	如果 pathname 参数指向的是一个符号链接，将不对其进行解引用，直接返回错误。	不加此标志情况下，如果 pathname 参数是一个符号链接，会对其进行解引用。

以上给大家介绍了一些比较常用的标志，open 函数的 flags 标志并不止这些，还有很多标志这里并没有给大家进行介绍，譬如 O_APPEND、O_ASYNC、O_DSYNC、O_NOATIME、O_NONBLOCK、O_SYNC 以及 O_TRUNC 等，对于这些没有提及到的标志，在后面学习过程中，也会给大家慢慢介绍。对于初学者来说，我们需要把表 2.3.1 中所列出的这些标志给弄明白、理解它们的作用和含义。

Tips：不同内核版本所支持的 flags 标志是存在差别的，譬如说新版本内核所支持的标志可能在老版本是不支持的，亦或者老版本支持的标志在新版本已经被取消、替代，man 手册中对一些标志是从哪个版本开始支持的有简单地说明，读者可以自行阅读！

前面我们说过，flags 参数时既可以单独使用某一个标志，也可以通过位或运算（|）将多个标志进行组合，譬如：

```
open("./src_file", O_RDONLY)                    //单独使用某一个标志
open("./src_file", O_RDONLY | O_NOFOLLOW)       //多个标志组合
```

mode：此参数用于指定新建文件的访问权限，只有当 flags 参数中包含 O_CREAT 或 O_TMPFILE 标志时才有效（O_TMPFILE 标志用于创建一个临时文件）。权限对于文件来说是一个很重要的属性，那么在 Linux 系统中，我们可以通过 touch 命令新建一个文件，此时文件会有一个默认的权限，如果需要修改文件权限，可通过 chmod 命令对文件权限进行修改，譬如在 Linux 系统下我们可以使用"ls -l"命令来查看到文件所对应的权限。

当我们调用 open 函数去新建一个文件时，也需要指定该文件的权限，而 mode 参数便用于指定此文件的权限，接下来看看我们该如何通过 mode 参数来表示文件的权限，首先 mode 参数的类型是 mode_t，这是一个 u32 无符号整形数据，权限表示方法如下所示：

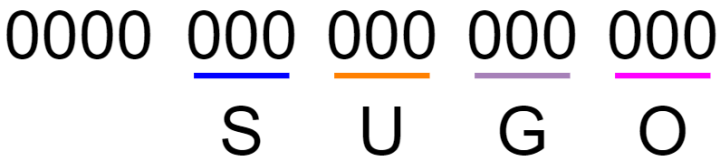


图 2.3.2 mode 权限表示方法

我们从低位从上看，每 3 个 bit 位分为一组，分别表示：

O---这 3 个 bit 位用于表示其他用户的权限；

G---这 3 个 bit 位用于表示同组用户（group）的权限，即与文件所有者有相同组 ID 的所有用户；

U---这 3 个 bit 位用于表示文件所属用户的权限，即文件或目录的所属者；

S---这 3 个 bit 位用于表示文件的特殊权限，文件特殊权限一般用的比较少，这里就不给大家细讲了。

关于什么是文件所属用户、同组用户以及其他用户，这些都是 Linux 操作系统相关的基础知识，相信大家都理解这些概念；3 个 bit 位中，按照 rwx 顺序来分配权限位（特殊权限除外），最高位（权值为 4）表示读权限，为 1 时表示具有读权限，为 0 时没有读权限；中间位（权值为 2）表示写权限，为 1 时表示具有写权限，为 0 时没有写权限；最低位（权值为 1）表示执行权限，为 1 时表示具有可执行权限，为 0 时没有执行权限。接下来我们举几个例子（特殊权限这里暂时不管，其 S 字段全部为 0）：

最高权限表示方法：111111111（二进制表示）、777（八进制表示）、511（十进制表示）；

最高权限这里意味着所有用户对此文件都具有读权限、写权限以及执行权限。

111000000（二进制表示）：表示文件所属者具有读、写、执行权限，而同组用户和其他用户不具有任何权限；

100100100（二进制表示）：表示文件所属者、同组用户以及其他用户都具有读权限，但都没有写、执行权限。

Tips：open 函数 O_RDONLY、O_WRONLY 以及 O_RDWR 这三个标志表示以什么方式去打开文件，譬如以只写方式打开（open 函数得到的文件描述符只能对文件进行写操作，不能读）、以只读方式打开（open 函数得到的文件描述符只能对文件进行读操作，不能写）、以可读可写方式打开（open 函数得到的文件描述符可对文件进行读和写操作）；与文件权限之间的联系，只有用户对该文件具有相应权限时，才可以使用对应的标志去打开文件，否则会打开失败！譬如，我们的程序对该文件只有只读权限，那么执行 open 函数使用 O_RDWR 或 O_WRONLY 标志将会失败。关于文件权限等相关问题，将会在 4.1 中给大家介绍。

关于文件权限表示方法的问题，以上就给大家介绍这么多，在实际编程中，我们可以直接使用 Linux 中已经定义好的宏，不同的宏定义表示不同的权限，如下所示：

表 2.3.2 open 函数文件权限宏

宏定义	说明
S_IRUSR	允许文件所属者读文件
S_IWUSR	允许文件所属者写文件
S_IXUSR	允许文件所属者执行文件
S_IRWXU	允许文件所属者读、写、执行文件
S_IRGRP	允许同组用户读文件
S_IWGRP	允许同组用户写文件
S_IXGRP	允许同组用户执行文件
S_IRWXG	允许同组用户读、写、执行文件
S_IROTH	允许其他用户读文件
S_IWOTH	允许其他用户写文件
S_IXOTH	允许其他用户执行文件
S_IRWXO	允许其他用户读、写、执行文件
S_ISUID S_ISGID S_ISVTX	set-user-ID（特殊权限） set-group-ID（特殊权限） sticky（特殊权限）

这些宏既可以单独使用，也可以通过位或运算将多个宏组合在一起，譬如：

```
S_IRUSR | S_IWUSR | S_IROTH
```

返回值：成功将返回文件描述符，文件描述符是一个非负整数；失败将返回-1。

以上就把 open 函数相关的基础知识给大家介绍完了，包括函数返回值、参数等信息，当然在后面的章节内容中，我们还会更加深入地给大家讲解 open 函数相关的知识点；接下来我们看一些 open 函数的简答使用示例。

open 函数使用示例

(1)使用 open 函数打开一个已经存在的文件（例如当前目录下的 app.c 文件），使用只读方式打开：

```
int fd = open("./app.c", O_RDONLY)
if (-1 == fd)
    return fd;
```

(2)使用 open 函数打开一个已经存在的文件（例如当前目录下的 app.c 文件），使用可读可写方式打开：

```
int fd = open("./app.c", O_RDWR)
if (-1 == fd)
    return fd;
```

(3)使用 open 函数打开一个指定的文件（譬如/home/dengtao/hello），使用可读可写方式,如果该文件是一个符号链接文件，则不对其进行解引用，直接返回错误：

```
int fd = open("/home/dengtao/hello", O_RDWR | O_NOFOLLOW);
if (-1 == fd)
    return fd;
```

(4)使用 open 函数打开一个指定的文件（譬如/home/dengtao/hello），如果该文件不存在则创建该文件，创建该文件时，将文件权限设置如下：

文件所属者拥有读、写、执行权限；

同组用户与其他用户只有读权限。

使用可读可写方式打开：

```
int fd = open("/home/dengtao/hello", O_RDWR | O_CREAT, S_IRWXU | S_IRGRP | S_IROTH);
if (-1 == fd)
    return fd;
```

2.4 write 写文件

调用 write 函数可向打开的文件写入数据，其函数原型如下所示（可通过"man 2 write"查看）：

```
#include <unistd.h>

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

首先使用 write 函数需要先包含 unistd.h 头文件。

函数参数和返回值含义如下：

fd：文件描述符。关于文件描述符，前面已经给大家进行了简单地讲解，这里不再重述！我们需要将进行写操作的文件所对应的文件描述符传递给 write 函数。

buf：指定写入数据对应的缓冲区。

count：指定写入的字节数。

返回值：如果成功将返回写入的字节数（0 表示未写入任何字节），如果此数字小于 count 参数，这不是错误，譬如磁盘空间已满，可能会发生这种情况；如果写入出错，则返回-1。

对于普通文件（我们一般操作的大部分文件都是普通文件，譬如常见的文本文件、二进制文件等），不管是读操作还是写操作，一个很重要的问题是：从文件的哪个位置开始进行读写操作？也就是 IO 操作所对应的位置偏移量，读写操作都是从文件的当前位置偏移量处开始，当然当前位置偏移量可以通过 lseek 系统调用进行设置，关于此函数后面再讲；默认情况下当前位置偏移量一般是 0，也就是指向了文件起始位置，当调用 read、write 函数读写操作完成之后，当前位置偏移量也会向后移动对应字节数，譬如当前位置偏移量为 1000 个字节处，调用 write()写入或 read()读取 500 个字节之后，当前位置偏移量将会移动到 1500 个字节处。

2.5 read 读文件

调用 read 函数可从打开的文件中读取数据，其函数原型如下所示（可通过"man 2 read"查看）：

```
#include <unistd.h>

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

首先使用 read 函数需要先包含 unistd.h 头文件。

函数参数和返回值含义如下：

fd：文件描述符。与 write 函数的 fd 参数意义相同。

buf：指定用于存储读取数据的缓冲区。

count：指定需要读取的字节数。

返回值：如果读取成功将返回读取到的字节数，实际读取到的字节数可能会小于 count 参数指定的字节数，也有可能会为 0，譬如进行读操作时，当前文件位置偏移量已经到了文件末尾。实际读取到的字节数少于要求读取的字节数，譬如在到达文件末尾之前有 30 个字节数据，而要求读取 100 个字节，则 read 读取成功只能返回 30；而下一次再调用 read 读，它将返回 0（文件末尾）。

2.6 close 关闭文件

可调用 close 函数关闭一个已经打开的文件，其函数原型如下所示（可通过"man 2 close"查看）：

```
#include <unistd.h>

int close(int fd);
```

首先使用 close 函数需要先包含 unistd.h 头文件，当我们对文件进行 IO 操作完成之后，后续不再对文件进行操作时，需要将文件关闭。

函数参数和返回值含义如下：

fd：文件描述符，需要关闭的文件所对应的文件描述符。

返回值：如果成功返回 0，如果失败则返回-1。

除了使用 close 函数显式关闭文件之外，在 Linux 系统中，当一个进程终止时，内核会自动关闭它打开的所有文件，也就是说在我们的程序中打开了文件，如果程序终止退出时没有关闭打开的文件，那么内核会自动将程序中打开的文件关闭。很多程序都利用了这一功能而不显式地用 close 关闭打开的文件。

显式关闭不再需要的文件描述符往往是良好的编程习惯，会使代码在后续修改时更具有可读性，也更可靠，进而言之，文件描述符是有限资源，当不再需要时必须将其释放、归还于系统。

2.7 lseek

对于每个打开的文件，系统都会记录它的读写位置偏移量，我们也把这个读写位置偏移量称为读写偏移量，记录了文件当前的读写位置，当调用 read()或 write()函数对文件进行读写操作时，就会从当前读写位置偏移量开始进行数据读写。

读写偏移量用于指示 read()或 write()函数操作时文件的起始位置，会以相对于文件头部的位置偏移量来表示，文件第一个字节数据的位置偏移量为 0。

当打开文件时，会将读写偏移量设置为指向文件开始位置处，以后每次调用 read()、write()将自动对其进行调整，以指向已读或已写数据后的下一字节，因此，连续的调用 read()和 write()函数将使得读写按顺序递增，对文件进行操作。我们先来看看 lseek 函数的原型，如下所示（可通过"man 2 lseek"查看）：

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

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

首先调用 lseek 函数需要包含<sys/types.h>和<unistd.h>两个头文件。

函数参数和返回值含义如下：

fd：文件描述符。

offset：偏移量，以字节为单位。

whence：用于定义参数 offset 偏移量对应的参考值，该参数为下列其中一种（宏定义）：

- SEEK_SET：读写偏移量将指向 offset 字节位置处（从文件头部开始算）；

- SEEK_CUR：读写偏移量将指向当前位置偏移量 + offset 字节位置处，offset 可以为正、也可以为负，如果是正数表示往后偏移，如果是负数则表示往前偏移；
- SEEK_END：读写偏移量将指向文件末尾 + offset 字节位置处，同样 offset 可以为正、也可以为负，如果是正数表示往后偏移、如果是负数则表示往前偏移。

返回值：成功将返回从文件头部开始算起的位置偏移量（字节为单位），也就是当前的读写位置；发生错误将返回-1。

使用示例：

(1)将读写位置移动到文件开头处：

```
off_t off = lseek(fd, 0, SEEK_SET);
if (-1 == off)
    return -1;
```

(2)将读写位置移动到文件末尾：

```
off_t off = lseek(fd, 0, SEEK_END);
if (-1 == off)
    return -1;
```

(3)将读写位置移动到偏移文件开头 100 个字节处：

```
off_t off = lseek(fd, 100, SEEK_SET);
if (-1 == off)
    return -1;
```

(4)获取当前读写位置偏移量：

```
off_t off = lseek(fd, 0, SEEK_CUR);
if (-1 == off)
    return -1;
```

函数执行成功将返回文件当前读写位置。

2.8 练习

本章给大家介绍了文件 IO 中的基础知识，包括文件 IO 常用到的系统调用 open()、read()、write()、close()以及 lseek()，当然关于这些系统调用所涉及到的知识点并没有讲完，本章所介绍的都是一些基础知识内容，在后面的学习过程中，我们会一一给大家深入讲解文件 IO 中的一些问题。

学完本章内容后，我们来做一些简单地文件 IO 编程练习，以巩固、提高大家所学知识内容，以下笔者给大家准备了 4 个简单地实战编程例子，大家可以尝试自己独立完成。

简单地编程实战例子

(1)打开一个已经存在的文件（例如 src_file），使用只读方式；然后打开一个新建文件（例如 dest_file），使用只写方式，新建文件的权限设置如下：

文件所属者拥有读、写、执行权限；

同组用户与其他用户只有读权限。

从 src_file 文件偏移头部 500 个字节位置开始读取 1Kbyte 字节数据，然后将读取出来的数据写入到 dest_file 文件中，从文件开头处开始写入，1Kbyte 字节大小，操作完成之后使用 close 显式关闭所有文件，然后退出程序。

(2)通过 open 函数判断文件是否存在（例如 test_file），并将判断结果显示出来。

(3)新建一个文件（例如 new_file），新建文件的权限设置为：

文件所属者拥有读、写、执行权限；

同组用户与其他用户只有读权限。

使用只写方式打开文件，将文件前 1Kbyte 字节数据填充为 0x00，将下 1Kbyte 字节数据填充为 0xFF，操作完成之后显式关闭文件，退出程序。

(4)打开一个已经存在的文件（例如 test_file），通过 lseek 函数计算该文件的大小，并打印出来。

以上就是本章内容给大家整理的几个简单地编程实战例子，大家可以根据本章所需知识内容将这几个例子独立完成，在本文档配套的资料包中，笔者会给出相应的示例代码，建议大家先自己独立思考，如果实在不知再参考笔者给出的示例代码。

vscode 编写代码

vscode 是一款非常好用的代码编辑器，我们可以在 Ubuntu 系统下使用 vscode 软件进行的代码编写，首先在 Ubuntu 系统下创建一个文件夹作为 vscode 的工作目录，譬如笔者在家目录下创建了一个名为 vscode_ws 的目录，将其作为 vscode 的工作目录，如下所示：

```
dt@dt-virtual-machine:~$ cd ~/
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ mkdir vscode_ws
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ cd vscode_ws/
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ pwd
/home/dt/vscode_ws
dt@dt-virtual-machine:~/vscode_ws$
```

图 2.8.1 创建 vscode 工作目录

在 Ubuntu 系统下打开 vscode 软件，如下所示：

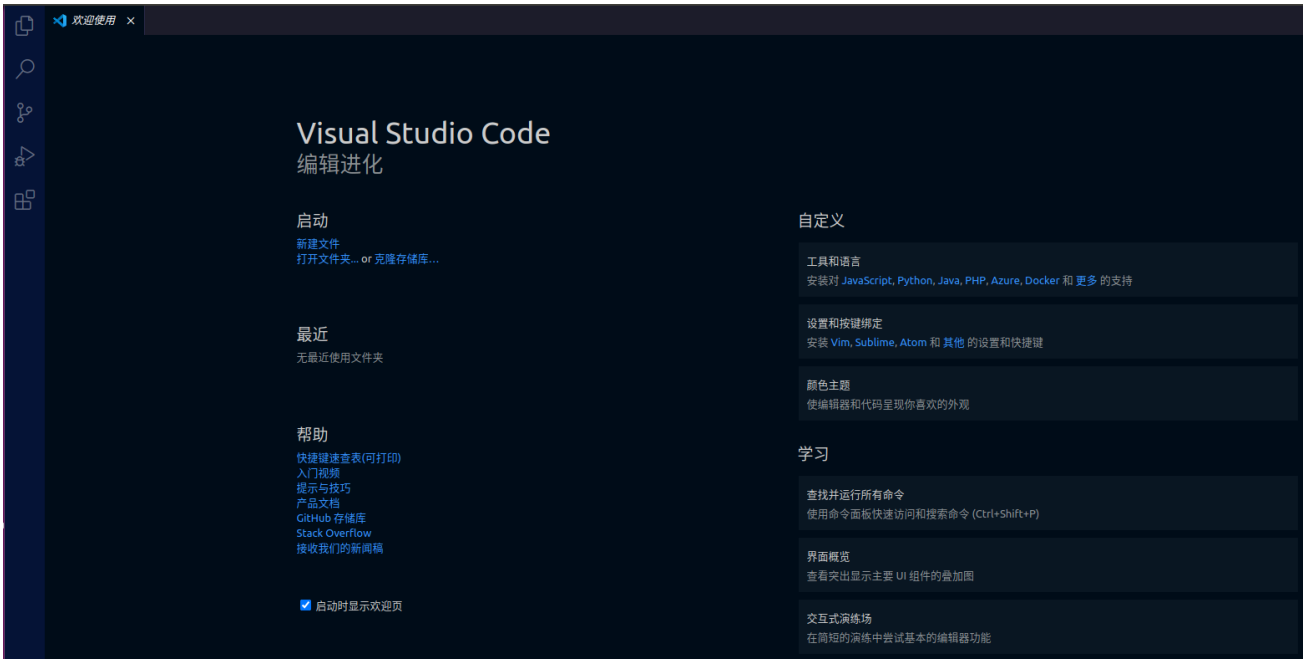


图 2.8.2 打开 vscode 软件

选择上边菜单栏，点击“文件--->打开文件夹”，在弹出来的页面中选择需要打开的文件夹，这里我们选择前面创建好的 vscode_ws 目录，如下所示：

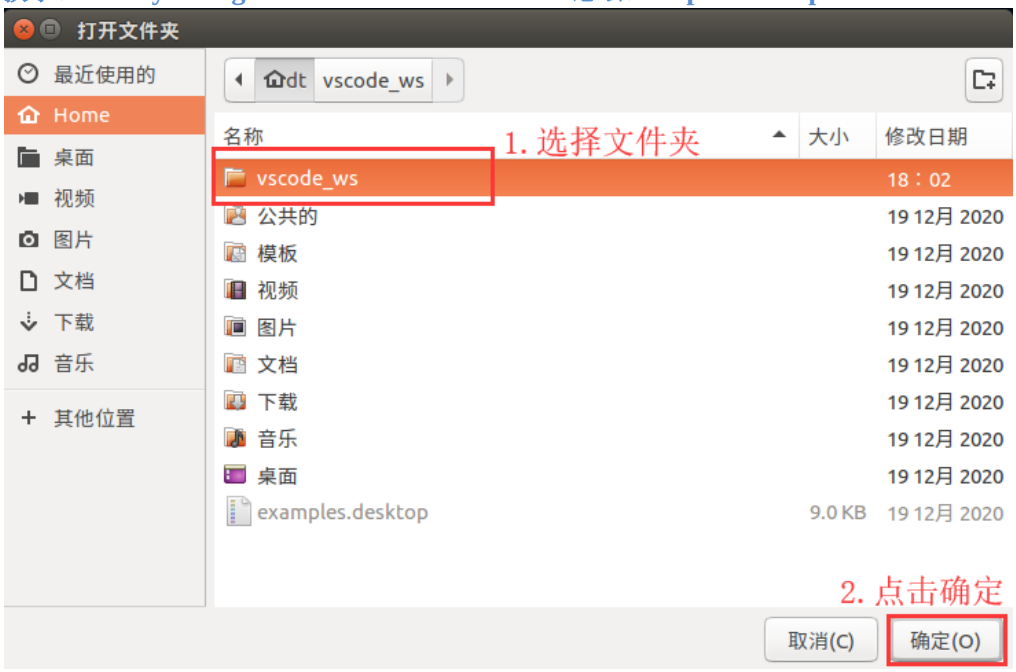


图 2.8.3 打开工作目录

接下来在工作目录下创建一个文件夹作为本章编程实战例程源码存放目录，直接在 vscode 软件进行创建即可，这里笔者将其命名为 1_chapter，如下所示：

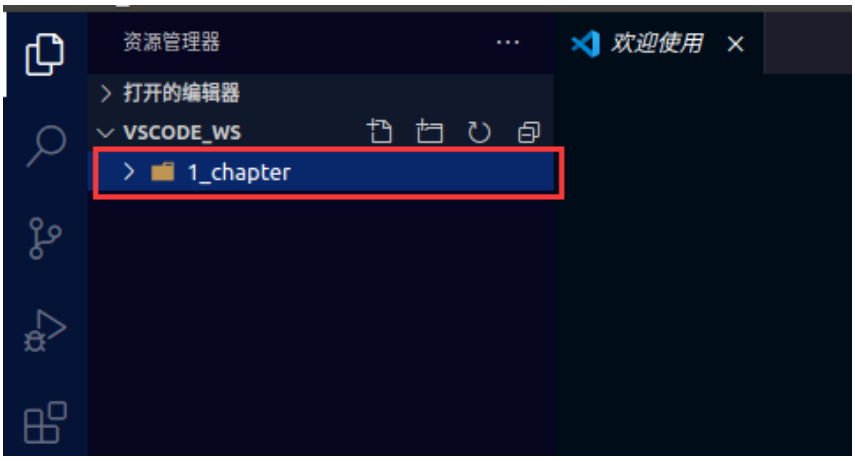


图 2.8.4 在工作目录下创建一个文件夹

接下来在 1_chapter 目录下创建一个.c 源文件，将其命名为 testApp_1.c，此源文件对应上面第一个(1)编程实战例子，如下所示：

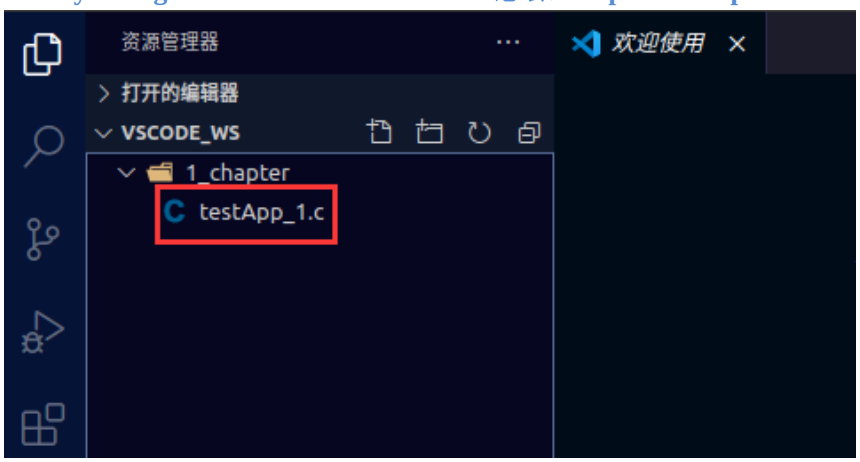


图 2.8.5 创建 C 程序源文件

点击打开 testApp_1.c 文本编辑页面，接下来就可以在 testApp_1.c 源文件中编写第一个实战例子对应的源代码了，以下是笔者给大家提供的一份示例代码，大家可以进行参考，但还是希望大家能够根据本章所学知识独立完成代码编写：

示例代码 2.8.1 编程实战例子 1

```
/*****************************************************************
 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
 文件名 : testApp_1.c
 作者 : 邓涛
 版本 : V1.0
 描述 :
 论坛 : www.openedv.com
 日志 : 初版 V1.0 2021/01/05 创建
 *****************************************************************/

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

int main(void)
{
    char buffer[1024];
    int fd1, fd2;
    int ret;

    /* 打开 src_file 文件 */
    fd1 = open("./src_file", O_RDONLY);
    if (-1 == fd1) {
        printf("Error: open src_file failed!\n");
        return -1;
```

```
    }

    /* 新建 dest_file 文件并打开 */
    fd2 = open("./dest_file", O_WRONLY | O_CREAT | O_EXCL,
                    S_IRWXU | S_IRGRP | S_IROTH);
    if (-1 == fd2) {
        printf("Error: open dest_file failed!\n");
        ret = -1;
        goto err1;
    }

    /* 将 src_file 文件读写位置移动到偏移文件头 500 个字节处 */
    ret = lseek(fd1, 500, SEEK_SET);
    if (-1 == ret)
        goto err2;

    /* 读取 src_file 文件数据，大小 1KByte */
    ret = read(fd1, buffer, sizeof(buffer));
    if (-1 == ret) {
        printf("Error: read src_file filed!\n");
        goto err2;
    }

    /* 将 dest_file 文件读写位置移动到文件头 */
    ret = lseek(fd2, 0, SEEK_SET);
    if (-1 == ret)
        goto err2;

    /* 将 buffer 中的数据写入 dest_file 文件，大小 1KByte */
    ret = write(fd2, buffer, sizeof(buffer));
    if (-1 == ret) {
        printf("Error: write dest_file failed!\n");
        goto err2;
    }

    printf("OK: test successful\n");
    ret = 0;

err2:
    close(fd2);

err1:
    close(fd1);
```

```
    return ret;
}
```

此代码中用到了库函数 printf，是一个格式化输出函数，用于将格式化打印信息输出显示到屏幕上（终端），使用此函数需要包含标准 I/O 库头文件<stdio.h>，对于有 C 语言编程经验的读者来说，如果对该函数的使用方法并不了解，可以查看 4.8.1 小节内容。

编译源码文件

代码编写完成之后，接下来我们需要对源代码进行编译，在 vscode 中点击上边菜单“终端--->新终端”打开一个终端，打开的终端将会在下面显示出来，如下所示：

图 2.8.6 在 vscode 中打开终端

vscode 提供了终端功能，这样就可以不使用 Ubuntu 自带的终端了，非常的方便、不需要在 Ubuntu 终端和 vscode 软件之间进行切换，进入到 1_chapter 目录下，如下所示：

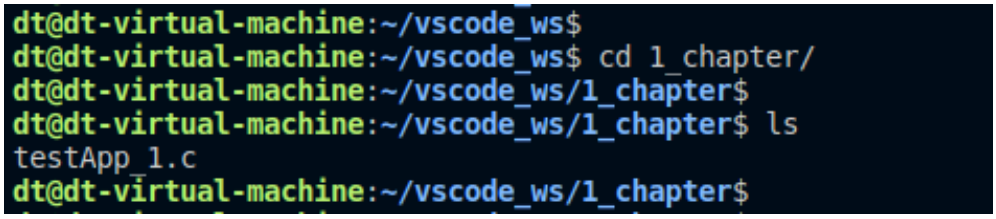


图 2.8.7 进入 1_chapter 目录

在该目录下我们直接使用 Ubuntu 系统提供的 gcc 编译器对源文件进行编译，编译生成一个可在 Ubuntu 系统下运行的可执行文件，执行如下命令：

```
gcc -o testApp_1 testApp_1.c
```

gcc 是 Ubuntu 系统下所使用的 C 语言编译器，-o 选项指定编译生成的可执行文件的名字，testApp_1.c 表示需要进行编译的 C 语言源文件，gcc 命令后面可以携带很多的编译相关的选项，这里不给大家介绍这个内容，这不是本文档我们需要去了解的重点，如果后面有机会可以给大家介绍下。编译完成之后会生成对应的可执行文件，如下所示：

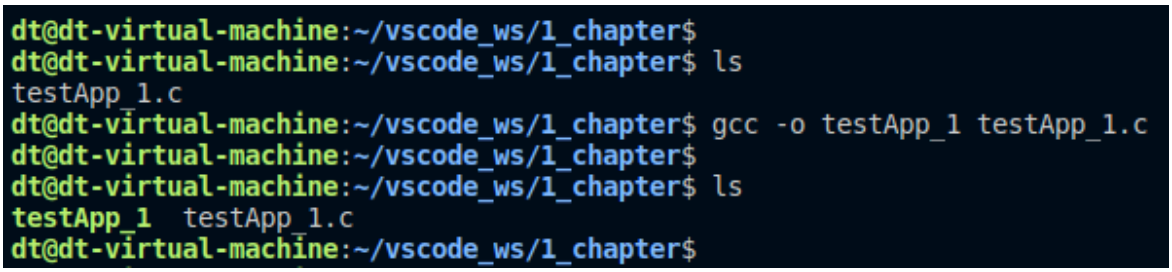


图 2.8.8 编译源文件

运行可执行文件测试

编译完成之后，接下来可以运行测试了，首先我们先准备一个文件 src_file，这个是例子要求需要打开一个已经存在的文件，文件大小大于或等于 1Kbyte，这里笔者已经将 src_file 文件放置到 1_chapter 目录下了，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls -l
总用量 28
-rw-rw-r-- 1 dt dt 8920 1月    5 18:46 src_file
-rwxrwxr-x 1 dt dt 8920 1月    5 18:46 testApp_1
-rw-rw-r-- 1 dt dt 1583 1月    5 17:37 testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 2.8.9 准备好一个测试需要用到的文件

直接在当前目录下运行 testApp_1 可执行文件：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ./testApp_1
OK: test successful
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls
dest_file  src_file  testApp_1  testApp_1.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 2.8.10 运行 testApp_1 可执行文件

运行成功之后会生成 dest_file 文件，这就是 testApp_1.c 源码中使用 open 创建的新文件，通过查看它的权限可知与源码中设置的权限是相同的，文件大小为 1Kbyte，其中这 1Kbyte 字节数据是从 src_file 文件中读取过来的。

剩下的几个例子希望大家自己独立完成。

Tips：最后再给大家说一点，就是关于函数返回值的问题，我们可以发现，本章给大家所介绍的这些函数都是有返回值的，其实不管是 Linux 应用编程 API 函数，还是驱动开发中所使用到的函数，基本上都是有返回值的，返回值的作用就是告诉开发人员此函数执行的一个状态是怎样的，执行成功了还是失败了，在 Linux 系统下，绝大部分的函数都是返回 0 作为函数调用成功的标识、而返回负数（譬如-1）表示函数调用失败，如果大家学习过驱动开发，想必对此并不陌生，所以很多时候可以使用如下的方式来判断函数执行成功还是失败：

```
if (func()) {
    //执行失败
} else {
    //执行成功
}
```

当然以上说的是大部分情况，并不是所有函数都是这样设计，所以呢，这里笔者也给大家一个建议，自己在进行编程开发的时候，自定义函数也可以使用这样的一种方法来设计你的函数返回值，不管是裸机程序亦或是 Linux 应用程序、驱动程序。

第三章 深入探究文件 I/O

经过上一章内容的学习，相信各位读者对 Linux 系统应用编程中的基础文件 I/O 操作有了一定的认识和理解了，能够独立完成一些简单地文件 I/O 编程问题，如果你的工作中仅仅只是涉及到一些简单文件读写操作相关的问题，其实上一章的知识内容已经够你使用了。

当然作为大部分读者来说，我相信你不会止步于此、还想学习更多的知识内容，那本章笔者将会同各位读者一起，来深入探究文件 I/O 中涉及到的一些问题、原理以及所对应的解决方法，譬如 Linux 系统下文件是如何进行管理的、调用函数返回错误该如何处理、open 函数的 O_APPEND、O_TRUNC 标志以及等相关问题。

好了，废话不多说，开始本章的学习吧，加油！

本章将会讨论如下主题内容。

- 对 Linux 下文件的管理方式进行简单介绍；
- 函数返回错误的处理；
- 退出程序 exit()、_Exit()、_exit()；
- 空洞文件的概念；
- open 函数的 O_APPEND 和 O_TRUNC 标志；
- 多次打开同一文件；
- 复制文件描述符；
- 文件共享介绍；
- 原子操作与竞争冒险；
- 系统调用 fcntl()和 ioctl()介绍；
- 截断文件；

3.1 Linux 系统如何管理文件

3.1.1 静态文件与 inode

文件在没有被打开的情况下一般都是存放在磁盘中的，譬如电脑硬盘、移动硬盘、U 盘等外部存储设备，文件存放在磁盘文件系统中，并且以一种固定的形式进行存放，我们把他们称为静态文件。

文件储存在硬盘上，硬盘的最小存储单位叫做“扇区”（Sector），每个扇区储存 512 字节（相当于 0.5KB），操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个“块”（block）。这种由多个扇区组成的“块”，是文件存取的最小单位。“块”的大小，最常见的是 4KB，即连续八个 sector 组成一个 block。

所以由此可以知道，静态文件对应的数据都是存储在磁盘设备不同的“块”中，那么问题来了，我们在程序中调用 open 函数是如何找到对应文件的数据存储“块”的呢，难道仅仅通过指定的文件路径就可以实现？这里我们就来简单地聊一聊这内部实现的过程。

我们的磁盘在进行分区、格式化的时候会将其分为两个区域，一个是数据区，用于存储文件中的数据；另一个是 inode 区，用于存放 inode table（inode 表），inode table 中存放的是一个一个的 inode（也成为 inode 节点），不同的 inode 就可以表示不同的文件，每一个文件都必须对应一个 inode，inode 实质上是一个结构体，这个结构体中有很多的元素，不同的元素记录了文件了不同信息，譬如文件字节大小、文件所有者、文件对应的读/写/执行权限、文件时间戳（创建时间、更新时间等）、文件类型、文件数据存储的 block（块）位置等等信息，如图 3.1.1 中所示（这里需要注意的是，文件名并不是记录在 inode 中，这个问题后面章节内容再给大家讲）。

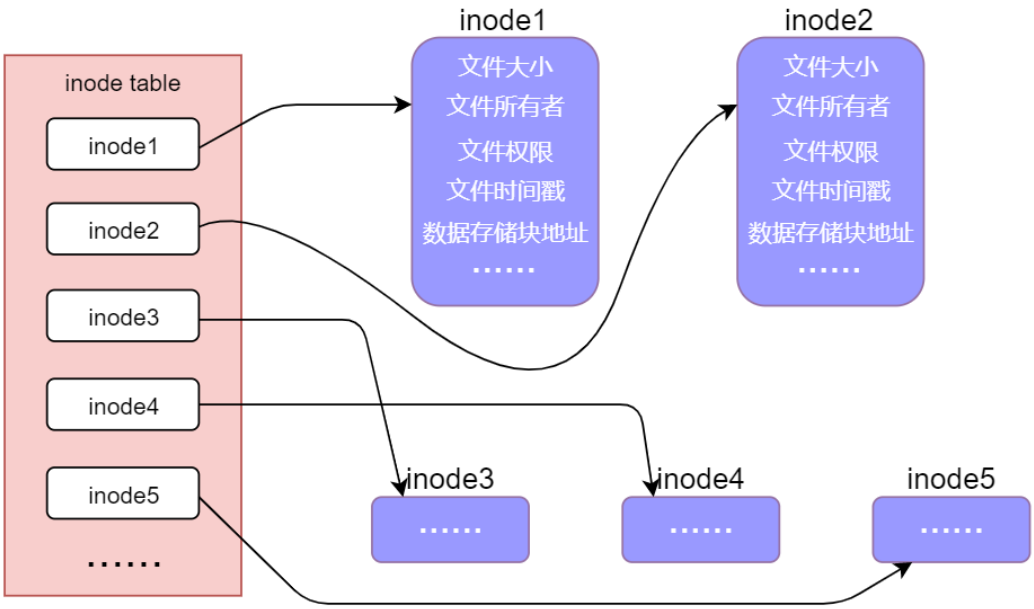


图 3.1.1 inode table 与 inode

所以由此可知，inode table 表本身也需要占用磁盘的存储空间。每一个文件都有唯一的一个 inode，每一个 inode 都有一个与之相对应的数字编号，通过这个数字编号就可以找到 inode table 中所对应的 inode。在 Linux 系统下，我们可以通过"ls -i"命令查看文件的 inode 编号，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls -il
总用量 16
3701769 -rw-rw-r-- 1 dt dt 1583 1月    5 19:54 testApp_1.c
3701836 -rw-rw-r-- 1 dt dt  709 1月    5 20:07 testApp_2.c
3701854 -rw-rw-r-- 1 dt dt 1381 1月    5 20:23 testApp_3.c
3702154 -rw-rw-r-- 1 dt dt  800 1月    5 20:33 testApp_4.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 3.1.2 ls 查看文件的 inode 编号

上图中 ls 打印出来的信息中，每一行前面的一个数字就表示了对应文件的 inode 编号。除此之外，还可以使用 stat 命令查看，用法如下：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ stat testApp_1.c
  文件：'testApp_1.c'
  大小：1583            块：8          IO 块：4096   普通文件
设备：801h/2049d        Inode: 3701769     硬链接：1
权限：(0664/-rw-rw-r--)  Uid: ( 1000/      dt)   Gid: ( 1000/      dt)
最近访问：2021-01-05 20:07:30.925816725 +0800
最近更改：2021-01-05 19:54:20.012237341 +0800
最近改动：2021-01-05 19:54:20.012237341 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 3.1.3 stat 查看 inode 编号

由以上的介绍大家可以联系到实际操作中，譬如我们在 Windows 下进行 U 盘格式化的时候会有一个“快速格式化”选项，如下所示：

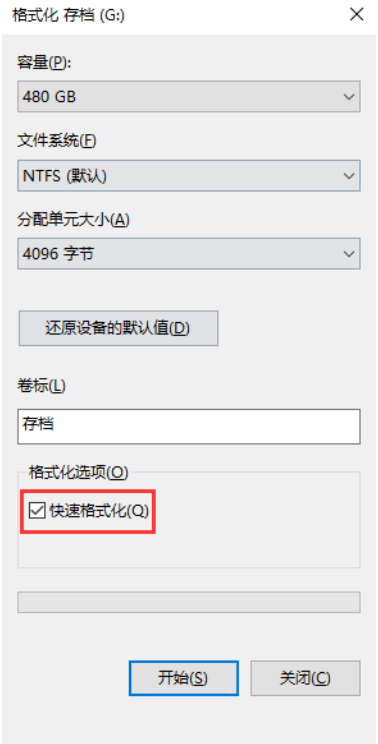


图 3.1.4 Windows 下格式化磁盘

如果勾选了“快速格式化”选项，在进行格式化操作的时候非常的快，而如果不勾选此选项，直接使用普通格式化方式，将会比较慢，那说明这两种格式化方式是存在差异的，其实快速格式化只是删除了 U 盘

中的 inode table 表，真正存储文件数据的区域并没有动，所以使用快速格式化的 U 盘，其中的数据是可以被找回来的。

通过以上介绍可知，打开一个文件，系统内部会将这个过程分为三步：

1) 系统找到这个文件名所对应的 inode 编号；
2) 通过 inode 编号从 inode table 中找到对应的 inode 结构体；
3) 根据 inode 结构体中记录的信息，确定文件数据所在的 block，并读出数据。

3.1.2 文件打开时的状态

当我们调用 open 函数去打开文件的时候，内核会申请一段内存（一段缓冲区），并且将静态文件的数据内容从磁盘这些存储设备中读取到内存中进行管理、缓存（也把内存中的这份文件数据叫做动态文件、内核缓冲区）。打开文件后，以后对这个文件的读写操作，都是针对内存中这一份动态文件进行相关的操作，而并不是针对磁盘中存放的静态文件。

当我们对动态文件进行读写操作后，此时内存中的动态文件和磁盘设备中的静态文件就不同步了，数据的同步工作由内核完成，内核会在之后将内存这份动态文件更新（同步）到磁盘设备中。由此我们也可以联系到实际操作中，譬如说：

- 打开一个大文件的时候会比较慢；
- 文档写了一半，没记得保存，此时电脑因为突然停电直接掉电关机了，当重启电脑后，打开编写的文档，发现之前写的内容已经丢失。

想必各位读者在工作当中都遇到过这种问题吧，通过上面的介绍，就解释了为什么会出现这种问题。好，我们再来说一下，为什么要这样设计？

因为磁盘、硬盘、U 盘等存储设备基本都是 Flash 块设备，因为块设备硬件本身有读写限制等特征，块设备是以一块一块为单位进行读写的（一个块包含多个扇区，而一个扇区包含多个字节），一个字节的改动也需要将该字节所在的 block 全部读取出来进行修改，修改完成之后再写入块设备中，所以导致对块设备的读写操作非常不灵活；而内存可以按字节为单位来操作，而且可以随机操作任意地址数据，非常地很灵活，所以对于操作系统来说，会先将磁盘中的静态文件读取到内存中进行缓存，读写操作都是针对这份动态文件，而不是直接去操作磁盘中的静态文件，不但操作不灵活，效率也会下降很多，因为内存的读写速率远比磁盘读写快得多。

在 Linux 系统中，内核会为每个进程（关于进程的概念，这是后面的内容，我们可以简单地理解为一个运行的程序就是一个进程，运行了多个程序那就是存在多个进程）设置一个专门的数据结构用于管理该进程，譬如用于记录进程的状态信息、运行特征等，我们把这个称为进程控制块（Process control block，缩写 PCB）。

PCB 数据结构体中有一个指针指向了文件描述符表（File descriptors），文件描述符表中的每一个元素索引到对应的文件表（File table），文件表也是一个数据结构体，其中记录了很多文件相关的信息，譬如文件状态标志、引用计数、当前文件的读写偏移量以及 i-node 指针（指向该文件对应的 inode）等，进程打开的所有文件对应的文件描述符都记录在文件描述符表中，每一个文件描述符都会指向一个对应的文件表，其示意图如下所示：

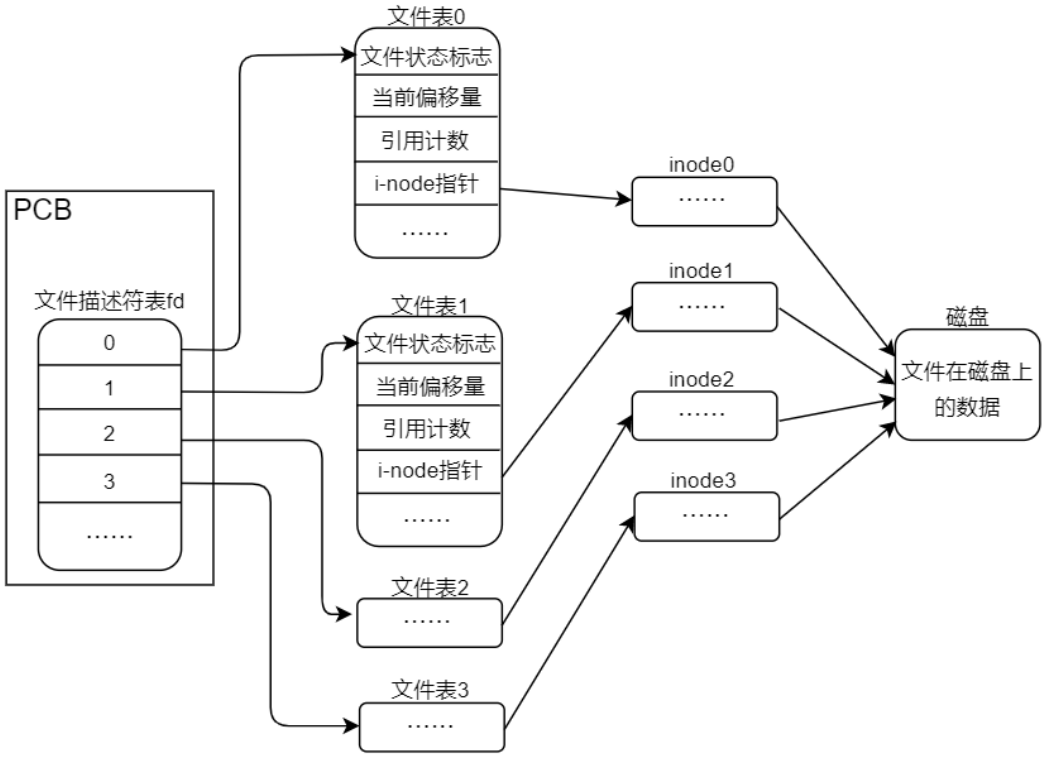


图 3.1.5 文件描述符表、文件表以及 inode 之间的关系

前面给大家介绍了 inode，inode 数据结构体中的元素会记录该文件的数据存储的 block（块），也就是说可以通过 inode 找到文件数据存在在磁盘设备中的那个位置，从而把文件数据读取出来。

以上就是本小节给大家介绍到所有内容了，上面给大家所介绍的内容后面的学习过程中还会用到，虽然这些理论知识对大家的编程并没有什么影响，但是会帮助大家理解文件 IO 背后隐藏的一些理论知识，其实这些理论知识还是非常浅薄的、只是一个大概的认识，其内部具体的实现是比较复杂的，当然这个不是我们学习 Linux 应用编程的重点，操作系统已经帮我们完成了这些具体的实现，我们要做的仅仅只是调用操作系统提供 API 函数来完成自己的工作。

好了，废话不多说，我们接着看下一小节内容。

3.2 返回错误处理与 errno

在上一章节中，笔者给大家编写了很多的示例代码，大家会发现这些示例代码会有一个共同的特点，那就是当判断函数执行失败后，会调用 return 退出程序，但是对于我们来说，我们并不知道为什么会出错，什么原因导致此函数执行失败，因为执行出错之后它们的返回值都是-1。

难道我们真的就不知道错误原因了吗？其实不然，在 Linux 系统下对常见的错误做了一个编号，每一个编号都代表着每一种不同的错误类型，当函数执行发生错误的时候，操作系统会将这个错误所对应的编号赋值给 errno 变量，每一个进程（程序）都维护了自己的 errno 变量，它是程序中的全局变量，该变量用于存储就近发生的函数执行错误编号，也就意味着下一次的错误码会覆盖上一次的错误码。所以由此可知道，当程序中调用函数发生错误的时候，操作系统内部会通过设置程序的 errno 变量来告知调用者究竟发生了什么错误！

errno 本质上是一个 int 类型的变量，用于存储错误编号，但是需要注意的是，并不是执行所有的系统调用或 C 库函数出错时，操作系统都会设置 errno，那我们如何确定一个函数出错时系统是否会设置 errno 呢？其实这个通过 man 手册便可以查到，譬如以 open 函数为例，执行"man 2 open"打开 open 函数的帮助信息，找到函数返回值描述段，如下所示：

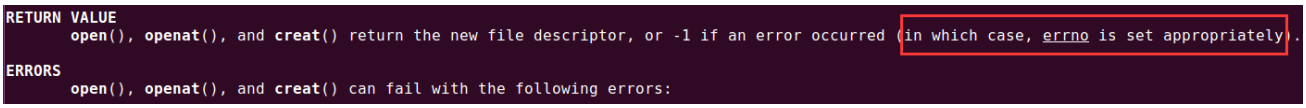


图 3.2.1 查看返回值描述信息

从图中红框部分描述文字可知，当函数返回错误时会设置 errno，当然这里是以 open 函数为例，其它的系统调用也可以这样查找，大家可以自己试试！

在我们的程序当中如何去获取系统所维护的这个 errno 变量呢？只需要在我们程序当中包含<errno.h>头文件即可，你可以直接认为此变量就是在<errno.h>头文件中的申明的，好，我们来测试下：

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

int main(void)
{
    printf("%d\n", errno);
    return 0;
}
```

以上的这段代码是不会报错的，大家可以自己试试！

3.2.1 strerror 函数

前面给大家说到了 errno 变量，但是 errno 仅仅只是一个错误编号，对于开发者来说，即使拿到了 errno 也不知道错误为何？还需要对比源码中对此编号的错误定义，可以说非常不友好，这里介绍一个 C 库函数 strerror()，该函数可以将对应的 errno 转换成适合我们查看的字符串信息，其函数原型如下所示（可通过"man 3 strerror"命令查看，注意此函数是 C 库函数，并不是系统调用）：

```
#include <string.h>

char *strerror(int errnum);
```

首先调用此函数需要包含头文件<string.h>。

函数参数和返回值如下：

errnum：错误编号 errno。

返回值：对应错误编号的字符串描述信息。

测试

接下来我们测试下，测试代码如下：

示例代码 3.2.1 strerror 测试代码

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

int main(void)
```

```
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        printf("Error: %s\n", strerror(errno));
        return -1;
    }

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

编译源代码，在 Ubuntu 系统下运行测试下，在当前目录下并不存在 test_file 文件，测试打印结果如下：

```
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ gcc -o test test.c
dt@dt-virtual-machine:~/vscode_ws$ ls
1_chapter  test  test.c
dt@dt-virtual-machine:~/vscode_ws$ ./test
Error: No such file or directory
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$
```

图 3.2.2 strerror 测试结果

从打印信息可以知道，strerror 返回的字符串是"No such file or directory"，所以从打印信息可知，我们就可以很直观的知道 open 函数执行的错误原因是文件不存在！

3.2.2 perror 函数

除了 strerror 函数之外，我们还可以使用 perror 函数来查看错误信息，一般用的最多的还是这个函数，调用此函数不需要传入 errno，函数内部会自己去获取 errno 变量的值，调用此函数会直接将错误提示字符串打印出来，而不是返回字符串，除此之外还可以在输出的错误提示字符串之前加入自己的打印信息，函数原型如下所示（可通过"man 3 perror"命令查看）：

```
#include <stdio.h>

void perror(const char *s);
```

需要包含<stdio.h>头文件。

函数参数和返回值含义如下：

s：在错误提示字符串信息之前，可加入自己的打印信息，也可不加，不加则传入空字符串即可。

返回值：void 无返回值。

测试

接下来我们进行测试，测试代码如下所示：

示例代码 3.2.2 perror 测试代码

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

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

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        return -1;
    }

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

编译源代码，在 Ubuntu 系统下运行测试下，在当前目录下并不存在 test_file 文件，测试打印结果如下：

```
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ gcc -o test test.c
dt@dt-virtual-machine:~/vscode_ws$ ls
1_chapter  test  test.c
dt@dt-virtual-machine:~/vscode_ws$ ./test
open error: No such file or directory
dt@dt-virtual-machine:~/vscode_ws$
```

图 3.2.3 perror 测试结果

从打印信息可以知道，perror 函数打印出来的错误提示字符串是"No such file or directory"，跟 strerror 函数返回的字符串信息一样，"open error"便是我们附加的打印信息，而且从打印信息可知，perror 函数会在附加信息后面自动加入冒号和空格以区分。

以上给大家介绍了 strerror、perror 两个 C 库函数，都是用于查看函数执行错误时对应的提示信息，大家用哪个函数都可以，这里笔者推荐大家使用 perror，在实际的编程中这个函数用的还是比较多的，当然除了这两个之外，其它其它一些类似功能的函数，这里就不再给大家介绍了，意义不大！

3.3 exit、_exit、_Exit

当程序在执行某个函数出错的时候，如果此函数执行失败会导致后面的步骤不能在进行下去时，应该在出错时终止程序运行，不应该让程序继续运行下去，那么如何退出程序、终止程序运行呢？有过编程经验的读者都知道使用 return，一般原则程序执行正常退出 return 0，而执行函数出错退出 return -1，前面我们所编写的示例代码也是如此。

在 Linux 系统下，进程（程序）退出可以分为正常退出和异常退出，注意这里说的异常并不是执行函数出现了错误这种情况，异常往往更多的是一种不可预料的系统异常，可能是执行了某个函数时发生的、也有可能是收到了某种信号等，这里我们只讨论正常退出的情况。

在 Linux 系统下，进程正常退出除了可以使用 return 之外，还可以使用 exit()、_exit()以及_Exit()，下面我们分别介绍。

3.3.1 _exit()和_Exit()函数

main 函数中使用 return 后返回，return 执行后把控制权交给调用函数，结束该进程。调用_exit()函数会清除其使用的内存空间，并销毁其在内核中的各种数据结构，关闭进程的所有文件描述符，并结束进程、将控制权交给操作系统。_exit()函数原型如下所示：

```
#include <unistd.h>

void _exit(int status);
```

调用函数需要传入 status 状态标志，0 表示正常结束、若为其它值则表示程序执行过程中检测到有错误发生。使用示例如下：

示例代码 3.3.1 _exit()使用示例

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

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        _exit(-1);
    }

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

用法很简单，大家可以自行测试！

_Exit()函数原型如下所示：

```
#include <stdlib.h>

void _Exit(int status);
```

_exit()和_Exit()两者等价，用法作用是一样的，这里就不再讲了，需要注意的是这 2 个函数都是系统调用。

3.3.2 exit()函数

exit()函数_exit()函数都是用来终止进程的，exit()是一个标准 C 库函数，而_exit()和_Exit()是系统调用。执行 exit()会执行一些清理工作，最后调用_exit()函数。exit()函数原型如下：

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

```
void exit(int status);
```

该函数是一个标准 C 库函数，使用该函数需要包含头文件<stdlib.h>，该函数的用法和_exit()/_Exit()是一样的，这里就不再多说了。

本小节就给大家介绍了 3 中终止进程的方法：

- main 函数中运行 return；
- 调用 Linux 系统调用_exit()或_Exit()；
- 调用 C 标准库函数 exit()。

不管你用哪一种都可以结束进程，但还是推荐大家使用 exit()，其实关于 return、exit、_exit/_Exit()之间的区别笔者在上面只是给大家简单地描述了一下，甚至不太确定我的描述是否正确，因为笔者并不太多去关心其间的差异，对这些概念的描述会比较模糊、笼统，如果大家看不明白可以自己百度搜索相关的内容，当然对于初学者来说，不太建议大家去查找这些东西，至少对你现阶段来说，意义不是很大。好，本小节就介绍这么多，我们接着学习下一小节的内容。

3.4 空洞文件

3.4.1 概念

什么是空洞文件（hole file）？在上一章内容中，笔者给大家介绍了 lseek()系统调用，使用 lseek 可以修改文件的当前读写位置偏移量，此函数不但可以改变位置偏移量，并且还允许文件偏移量超出文件长度，这是什么意思呢？譬如有一个 test_file，该文件的大小是 4K（也就是 4096 个字节），如果通过 lseek 系统调用将该文件的读写偏移量移动到偏移文件头部 6000 个字节处，大家想一想会怎样？如果笔者没有提前告诉大家，大家觉得不能这样操作，但事实上 lseek 函数确实可以这样操作。

接下来使用 write()函数对文件进行写入操作，也就是说此时将是从偏移文件头部 6000 个字节处开始写入数据，也就意味着 4096~6000 字节之间出现了一个空洞，因为这部分空间并没有写入任何数据，所以形成了空洞，这部分区域就被称为文件空洞，那么相应的该文件也被称为空洞文件。

文件空洞部分实际上并不会占用任何物理空间，直到在某个时刻对空洞部分进行写入数据时才会为它分配对应的空间，但是空洞文件形成时，逻辑上该文件的大小是包含了空洞部分的大小的，这点需要注意。

那说了这么多，空洞文件有什么用呢？空洞文件对多线程共同操作文件是及其有用的，有时候我们创建一个很大的文件，如果单个线程从头开始依次构建该文件需要很长的时间，有一种思路就是将文件分为多段，然后使用多线程来操作，每个线程负责其中一段数据的写入；这个有点像我们现实生活当中施工队修路的感觉，比如说修建一条高速公路，单个施工队修筑会很慢，这个时候可以安排多个施工队，每一个施工队负责修建其中一段，最后将他们连接起来。

来看一下实际中空洞文件的两个应用场景：

- 在使用迅雷下载文件时，还未下载完成，就发现该文件已经占据了全部文件大小的空间，这也是空洞文件；下载时如果没有空洞文件，多线程下载时文件就只能从一个地方写入，这就不能发挥多线程的作用了；如果有了空洞文件，可以从不同的地址同时写入，就达到了多线程的优势；
- 在创建虚拟机时，你给虚拟机分配了 100G 的磁盘空间，但其实系统安装完成之后，开始也不过只用了 3、4G 的磁盘空间，如果一开始就把 100G 分配出去，资源是很大的浪费。

关于空洞文件，这里就介绍这么多，上述描述当中多次提到了线程这个概念，关于线程这是后面的内容，这里先不给大家讲。

3.4.2 实验测试

这里我们进行相关的测试，新建一个文件把它做成空洞文件，示例代码如下所示：

示例代码 3.4.1 空洞文件测试代码

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

int main(void)
{
    int fd;
    int ret;
    char buffer[1024];
    int i;

    /* 打开文件 */
    fd = open("./hole_file", O_WRONLY | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件读写位置移动到偏移文件头 4096 个字节(4K)处 */
    ret = lseek(fd, 4096, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    /* 初始化 buffer 为 0xFF */
    memset(buffer, 0xFF, sizeof(buffer));

    /* 循环写入 4 次，每次写入 1K */
    for (i = 0; i < 4; i++) {

        ret = write(fd, buffer, sizeof(buffer));
        if (-1 == ret) {
            perror("write error");
            goto err;
        }
    }
```

```
    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

示例代码中，我们使用 open 函数新建了一个文件 hole_file，在 Linux 系统中，新建文件大小是 0，也就是没有任何数据写入，此时使用 lseek 函数将读写偏移量移动到 4K 字节处，再使用 write 函数写入数据 0xFF，每次写入 1K，一共写入 4 次，也就是写入了 4K 数据，也就意味着该文件前 4K 是文件空洞部分，而后 4K 数据才是真正写入的数据。

接下来进行编译测试，首先确保当前文件目录下不存在 hole_file 文件，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
hole_file  testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -lh hole_file
-rw-r--r-- 1 dt dt 8.0K 1月    8 10:10 hole_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ du -h hole_file
4.0K    hole_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

示例代码 3.4.2 空洞文件测试结果

使用 ls 命令查看到空洞文件的大小是 8K，使用 ls 命令查看到的大小是文件的逻辑大小，自然是包括了空洞部分大小和真实数据部分大小；当使用 du 命令查看空洞文件时，其大小显示为 4K，du 命令查看到的大小是文件实际占用存储块的大小。

本小节内容就讲解完了，最后再向各位抛出一个问题：若使用 read 函数读取文件空洞部分，读取出来的将会是什么？关于这个问题大家可以先思考下，至于结果是什么，笔者这里便不给出答案了，大家可以自己动手编写代码进行测试以得出结论。

3.5 O_APPEND 和 O_TRUNC 标志

在上一章给大家讲解 open 函数的时候介绍了一些 open 函数的 flags 标志，譬如 O_RDONLY、O_WRONLY、O_CREAT、O_EXCL 等，本小节再给大家介绍两个标志，分别是 O_APPEND 和 O_TRUNC，接下来对这两个标志分别进行介绍。

3.5.1 O_TRUNC 标志

O_TRUNC 这个标志的作用非常简单，如果使用了这个标志，调用 open 函数打开文件的时候会将文件原本的内容全部丢弃，文件大小变为 0；这里我们直接测试即可！测试代码如下所示：

示例代码 3.5.1 O_TRUNC 标志测试

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

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

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_WRONLY | O_TRUNC);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

在当前目录下有一个文件 test_file，测试代码中使用了 O_TRUNC 标志打开该文件，代码中仅仅只是打开该文件，之后调用 close 关闭了文件，并没有对其进行读写操作，接下来编译运行来看看测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 16
-rw-rw-r-- 1 dt dt  318 1月   8 16:06 testApp.c
-rw-rw-r-- 1 dt dt 8760 1月   8 16:06 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 16
-rwxrwxr-x 1 dt dt 8760 1月   8 16:07 testApp
-rw-rw-r-- 1 dt dt  318 1月   8 16:06 testApp.c
-rw-rw-r-- 1 dt dt    0 1月   8 16:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.5.1 O_TRUNC 测试结果

在测试之前 test_file 文件中是有数据的，文件大小为 8760 个字节，执行完测试程序后，再使用 ls 命令查看文件大小时发现 test_file 大小已经变成了 0，也就是说明文件之前的内容已经全部被丢弃了。这就是 O_TRUNC 标志的作用了，大家可以自己动手试试。

3.5.2 O_APPEND 标志

接下里聊一聊 O_APPEND 标志，如果 open 函数携带了 O_APPEND 标志，调用 open 函数打开文件，当每次使用 write()函数对文件进行写操作时，都会自动把文件当前位置偏移量移动到文件末尾，从文件末尾开始写入数据，也就是意味着每次写入数据都是从文件末尾开始。这里我们直接进行测试，测试代码如下所示：

示例代码 3.5.2 O_APPEND 标志测试

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

int main(void)
{
    char buffer[16];
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("./test_file", O_RDWR | O_APPEND);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 初始化 buffer 中的数据 */
    memset(buffer, 0x55, sizeof(buffer));

    /* 写入数据: 写入 4 个字节数据 */
    ret = write(fd, buffer, 4);
    if (-1 == ret) {
        perror("write error");
        goto err;
    }

    /* 将 buffer 缓冲区中的数据全部清 0 */
    memset(buffer, 0x00, sizeof(buffer));

    /* 将位置偏移量移动到距离文件末尾 4 个字节处 */
    ret = lseek(fd, -4, SEEK_END);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    /* 读取数据 */
```

```
    ret = read(fd, buffer, 4);
    if (-1 == ret) {
        perror("read error");
        goto err;
    }

    printf("0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[1],
                buffer[2], buffer[3]);
    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

测试代码中会去打开当前目录下的 test_file 文件，使用可读可写方式，并且使用了 O_APPEND 标志，前面笔者给大家提到过，open 打开一个文件，默认的读写位置偏移量会处于文件头，但测试代码中使用了 O_APPEND 标志，如果 O_APPEND 确实能生效的话，也就意味着调用 write 函数会从文件末尾开始写；代码中写入了 4 个字节数据，都是 0x55，之后,使用 lseek 函数将位置偏移量移动到距离文件末尾 4 个字节处，读取 4 个字节（也就是读取文件最后 4 个字节数据），之后将其打印出来，如果上面笔者的描述正确的话，打印出来的数据就是我们写入的数据，如果 O_APPEND 不能生效，则打印出来数据就不会是 0x55，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0x55 0x55 0x55 0x55
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.5.2 O_APPEND 标志测试结果

从上面打印信息可知，读取出来的数据确实等于 0x55，说明 O_APPEND 标志确实有作用，当调用 write() 函数写文件时，会自动把文件当前位置偏移量移动到文件末尾。

当然，本小节内容还并没有结束，这其中还涉及到一些细节问题需要大家注意，首先第一点，O_APPEND 标志并不会影响读文件，当读取文件时，O_APPEND 标志并不会影响读位置偏移量，即使使用了 O_APPEND 标志，读文件位置偏移量默认情况下依然是文件头，关于这个问题大家可以自己进行测试，编程是一个实践性很强的工作，有什么不能理解的问题，可以自己编写程序进行测试。

大家可能会想到使用 lseek 函数来改变 write()时的写位置偏移量，其实这种做法并不会成功，这就是笔者给大家提的第二个细节，使用了 O_APPEND 标志，即使是通过 lseek 函数也是无法修改写文件时对应的位置偏移量（注意笔者这里说的是写文件，并不包括读），写入数据依然是从文件末尾开始，lseek 并不会该变写位置偏移量，这个问题测试方法很简单，也就是在 write 之前使用 lseek 修改位置偏移量，这里笔者就不再给大家测试了，我还是那句话，编程是一个实践性很强的工作，大家只需要把示例代码 3.5.2 进行简单地修改即可！

其实关于第二点细节原因很简单，当执行 write()函数时，检测到 open 函数携带了 O_APPEND 标志，所以在 write 函数内部会自动将写位置偏移量移动到文件末尾，当然这里也只是笔者的一个简单地猜测，至于是不是这样，笔者也无从考证。

到这里本小节的内容就暂时介绍完了，为什么说是“暂时”？因为后面的内容中还会聊到 O_APPEND 标志，最后笔者再给大家出一个小问题，大家可以自己动手测试。

- 当 open 函数同时携带了 O_APPEND 和 O_TRUNC 两个标志时会有什么作用？

3.6 多次打开同一个文件

大家看到这个小节标题可能会有疑问，同一个文件还能被多次打开？事实确实如此，同一个文件可以被多次打开，譬如在一个进程中多次打开同一个文件、在多个不同的进程中打开同一个文件，那么这些操作都是被允许的。本小节就来探讨下多次打开同一个文件会有一些什么现象以及相应的细节问题？

3.6.1 验证一些现象

- **一个进程内多次 open 打开同一个文件，那么会得到多个不同的文件描述符 fd，同理在关闭文件的时候也需要调用 close 依次关闭各个文件描述符。**

针对这个问题，我们编写测试代码进行测试，如下所示：

示例代码 3.6.1 多次打开同一个文件测试代码 1

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

int main(void)
{
    int fd1, fd2, fd3;
    int ret;

    /* 第一次打开文件 */
    fd1 = open("./test_file", O_RDWR);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 第二次打开文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
```

```
    }

    /* 第三次打开文件 */
    fd3 = open("./test_file", O_RDWR);
    if (-1 == fd3) {
        perror("open error");
        ret = -1;
        goto err2;
    }

    /* 打印出 3 个文件描述符 */
    printf("%d   %d   %d\n", fd1, fd2, fd3);

    close(fd3);
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

上述示例代码中，通过 3 次调用 open 函数对 test_file 文件打开了 3 次，每一个调用传参一样，最后将 3 次得到的文件描述符打印出来，在当前目录下存在 test_file 文件，接下来编译测试，看看结果如何：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
6  7  8
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.1 打印文件描述符

从打印结果可知，三次调用 open 函数得到的文件描述符分别为 6、7、8，通过任何一个文件描述符对文件进行 IO 操作都是可以的，但是需要注意是，调用 open 函数打开文件使用的是什么权限，则返回的文件描述符就拥有什么权限，文件 IO 操作完成之后，在结束进程之前需要使用 close 关闭各个文件描述符。

在图 3.6.1 中，细心的读者可能会发现，调用 open 函数得到的最小文件描述符是 6，在上一章节内容中给大家提到过，程序中分配得到的最小文件描述符一般是 3，但这里竟然是 6！这是为何？其实这个问题跟 vscode 有关，说明 3、4、5 这 3 个文件描述符已经被 vscode 软件对应的进程所占用了，而当前这里执行 testApp 文件是在 vscode 软件提供的终端下进行的，所以 vscode 可以认为是 testApp 进程的父进程，相反，testApp 进程便是 vscode 进程的子进程，子进程会继承父进程的文件描述符。关于子进程和父进程这些都是后面的内容，这里暂时不给大家进行介绍，这是只是给大家简单地解释一下，免得大家误会！

其实可以直接在 Ubuntu 系统的 Terminal 终端执行 testApp，这时你会发现打印出来的文件描述符分别是 3、4、5，这里就不给大家演示了。

- **一个进程内多次 open 打开同一个文件，在内存中并不会存在多份动态文件。**

当调用 open 函数的时候，会将文件数据（文件内容）从磁盘等块设备读取到内存中，将文件数据在内存中进行维护，内存中的这份文件数据我们就把它称为动态文件！这是前面给大家介绍的内容，这里再简单地提一下。这里出现了一个问题：如果同一个文件被多次打开，那么该文件所对应的动态文件是否在内存中也存在多份？也就是说，多次打开同一个文件是否会将其文件数据多次拷贝到内存中进行维护？

关于这个问题，各位读者可以简单地思考一下，这里我们直接编写代码进行测试，测试代码如下所示：

示例代码 3.6.2 多次打开同一个文件测试代码 2

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

int main(void)
{
    char buffer[4];
    int fd1, fd2;
    int ret;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* 通过 fd1 文件描述符写入 4 个字节数据 */
    buffer[0] = 0x11;
    buffer[1] = 0x22;
```

```
    buffer[2] = 0x33;
    buffer[3] = 0x44;

    ret = write(fd1, buffer, 4);
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }

    /* 将读写位置偏移量移动到文件头 */
    ret = lseek(fd2, 0, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err2;
    }

    /* 读取数据 */
    memset(buffer, 0x00, sizeof(buffer));
    ret = read(fd2, buffer, 4);
    if (-1 == ret) {
        perror("read error");
        goto err2;
    }

    printf("0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[1],
                buffer[2], buffer[3]);

    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

当前目录下不存在 test_file 文件，上述代码中，第一次调用 open 函数新建并打开 test_file 文件，第二次调用 open 函数再次打开它，新建文件时，文件大小为 0；首先通过文件描述符 fd1 写入 4 个字节数据（0x11/0x22/0x33/0x44），从文件头开始写；然后再通过文件描述符 fd2 读取 4 个字节数据，也是从文件头开始读取。假如，内存中只有一份动态文件，那么读取得到的数据应该就是 0x11、0x22、0x33、0x44，如果存在多份动态文件，那么通过 fd2 读取的是与它对应的动态文件中的数据，那就不是 0x11、0x22、0x33、0x44，而是读取出 0 个字节数据，因为它的文件大小是 0。

接下来进行编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0x11 0x22 0x33 0x44
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.2 测试结果 2

上图中打印显示读取出来的数据是 0x11/0x22/0x33/0x44，所以由此可知，即使多次打开同一个文件，内存中也只有一份动态文件。

- **一个进程内多次 open 打开同一个文件，不同文件描述符所对应的读写位置偏移量是相互独立的。**

同一个文件被多次打开，会得到多个不同的文件描述符，也就意味着会有多个不同的文件表，而文件读写偏移量信息就记录在文件表数据结构中，所以从这里可以推测不同的文件描述符所对应的读写偏移量是相互独立的，并没有关联在一起，并且文件表中 i-node 指针指向的都是同一个 inode，如下图所示：

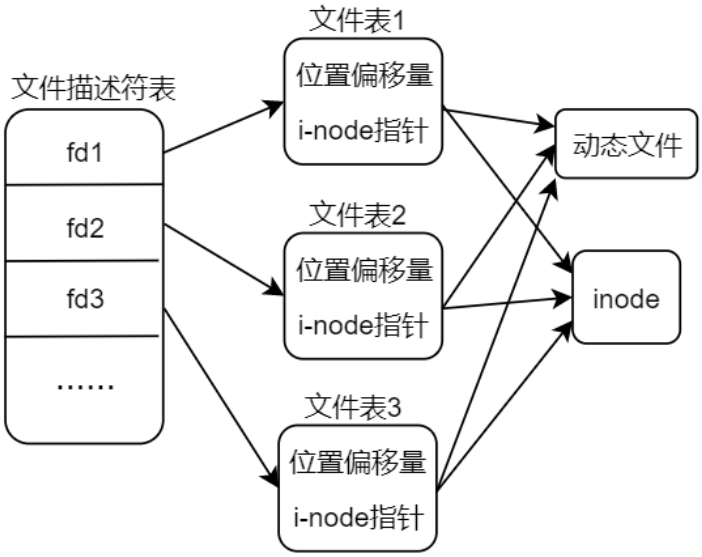


图 3.6.3 多次打开同一个文件--文件描述符表、文件表以及 inode 之间的关系

测试的方法很简单，只需在示例代码 3.6.2 中简单地修改即可，将 lseek 函数调用去掉，然后在编译测试，如果读出的数据依然是 0x11/0x22/0x33/0x44，则表示第三点结论成立，这里不再给大家演示。

Tips：多个不同的进程中调用 open()打开磁盘中的同一个文件，同样在内存中也只是维护了一份动态文件，多个进程间共享，它们有各自独立的文件读写位置偏移量。

动态文件何时被关闭呢？当文件的引用计数为 0 时，系统会自动将其关闭，同一个文件被打开多次，文件表中会记录该文件的引用计数，如图 3.1.5 所示，引用计数记录了当前文件被多少个文件描述符 fd 关联。

3.6.2 多次打开同一文件进行读操作与 O_APPEND 标志

重复打开同一个文件，进行写操作，譬如一个进程中两次调用 open 函数打开同一个文件，分别得到两个文件描述符 fd1 和 fd2，使用这两个文件描述符对文件进行写入操作，那么它们是分别写（各从各的位置偏移量开始写）还是接续写（一个写完，另一个接着后面写）？其实这个问题，3.6.1 小节中已经给出了答

案，因为这两个文件描述符所对应的读写位置偏移量是相互独立的，所以是分别写，接下来我们还是编写代码进行测试，测试代码如下所示：

示例代码 3.6.3 多次打开同一个文件测试代码 3

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

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* buffer 数据初始化 */
    buffer1[0] = 0x11;
    buffer1[1] = 0x22;
    buffer1[2] = 0x33;
    buffer1[3] = 0x44;

    buffer2[0] = 0xAA;
    buffer2[1] = 0xBB;
    buffer2[2] = 0xCC;
```

```
    buffer2[3] = 0xDD;

    /* 循环写入数据 */
    for (i = 0; i < 4; i++) {

        ret = write(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("write error");
            goto err2;
        }

        ret = write(fd2, buffer2, sizeof(buffer2));
        if (-1 == ret) {
            perror("write error");
            goto err2;
        }
    }

    /* 将读写位置偏移量移动到文件头 */
    ret = lseek(fd1, 0, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err2;
    }

    /* 读取数据 */
    for (i = 0; i < 8; i++) {

        ret = read(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("read error");
            goto err2;
        }

        printf("%x%x%x%x", buffer1[0], buffer1[1],
                buffer1[2], buffer1[3]);
    }

    printf("\n");
    ret = 0;
err2:
    close(fd2);
```

```
err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

示例代码 3.6.3 中，重复两次打开 test_file 文件，分别得到两个文件描述符 fd1、fd2；首先通过 fd1 写入 4 个字节数据（0x11、0x22、0x33、0x44）到文件中，接着再通过 fd2 写入 4 个字节数据（0xaa、0xbb、0xcc、0xdd）到文件中，循环写入 4 此；最后再将写入的数据读取出来，将其打印到终端。如果它们是分别写，那么读取出来的数据就应该是 aabbccdd……，因为通过 fd1 写入的数据被 fd2 写入的数据给覆盖了；如果它们是接续写，那么读取出来的数据应该是 11223344aabbccdd……，接下里我们编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
aabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.4 测试结果 3

从打印结果可知，它们确实是分别写。如果想要实现接续写，也就是当通过 fd1 写入完成之后，通过 fd2 写入的数据是接在 fd1 写入的数据之后，那么该怎么做呢？当然可以写入数据之前通过 lseek 函数将文件偏移量移动到文件末尾，如果是这样做，会存在一些问题，关于这个问题后面再给大家介绍；这里我们给大家介绍使用 O_APPEND 标志来解决这个问题，也就是将分别写更改为接续写。

前面给大家介绍了 open 函数的 O_APPEND 标志，当 open 函数使用 O_APPEND 标志，在使用 write 函数进行写入操作时，会自动将偏移量移动到文件末尾，也就是每次写入都是从文件末尾开始；这里结合本小节的内容，我们再来讨论 O_APPEND 标志，在多次打开同一个文件进行写操作时，使用 O_APPEND 标志会有什么样的效果，接下来进行测试：

示例代码 3.6.4 多次打开同一个文件测试代码 4

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

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL | O_APPEND,
```

```
                S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR | O_APPEND);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* buffer 数据初始化 */
    buffer1[0] = 0x11;
    buffer1[1] = 0x22;
    buffer1[2] = 0x33;
    buffer1[3] = 0x44;

    buffer2[0] = 0xAA;
    buffer2[1] = 0xBB;
    buffer2[2] = 0xCC;
    buffer2[3] = 0xDD;

    /* 循环写入数据 */
    for (i = 0; i < 4; i++) {

        ret = write(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("write error");
            goto err2;
        }

        ret = write(fd2, buffer2, sizeof(buffer2));
        if (-1 == ret) {
            perror("write error");
            goto err2;
        }
    }

    /* 将读写位置偏移量移动到文件头 */
    ret = lseek(fd1, 0, SEEK_SET);
```

```
    if (-1 == ret) {
        perror("lseek error");
        goto err2;
    }

    /* 读取数据 */
    for (i = 0; i < 8; i++) {

        ret = read(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("read error");
            goto err2;
        }

        printf("%x%x%x%x", buffer1[0], buffer1[1],
                buffer1[2], buffer1[3]);
    }

    printf("\n");
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

示例代码 3.6.4 仅仅只是在示例代码 3.6.3 的基础上，open 函数添加了 O_APPEND 标志，其它内容并没有动过，接下来编译测试。

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
11223344aabbccdd11223344aabbccdd11223344aabbccdd11223344aabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.5 测试结果 4

从打印出来的数据可知，加入了 O_APPEND 标志后，分别写已经变成了接续写。关于 O_APPEND 标志还涉及到一个原子操作的问题，后面再给大家介绍，本小节内容到此！

3.7 复制文件描述符

在 Linux 系统中，open 返回得到的文件描述符 fd 可以进行复制，复制成功之后可以得到一个新的文件描述符，使用新的文件描述符和旧的文件描述符都可以对文件进行 IO 操作，复制得到的文件描述符和旧的

文件描述符拥有相同的权限，譬如使用旧的文件描述符对文件有读写权限，那么新的文件描述符同样也具有读写权限；在 Linux 系统下，可以使用 dup 或 dup2 这两个系统调用对文件描述符进行复制，本小节就给大家介绍这两个函数的用法以及它们之间的区别。

复制得到的文件描述符与旧的文件描述符都指向了同一个文件表，假设 fd1 为原文件描述符，fd2 为复制得到的文件描述符，如下图所示：

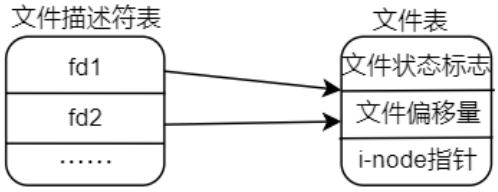


图 3.7.1 指向同一个文件表

因为复制得到的文件描述符与旧的文件描述符指向的是同一个文件表，所以可知，这两个文件描述符的属性是一样，譬如对文件的读写权限、文件状态标志、文件偏移量等，所以从这里也可知道“复制”的含义实则是复制文件表。同样，在使用完毕之后也需要使用 close 来关闭文件描述符。

3.7.1 dup 函数

dup 函数用于复制文件描述符，此函数原型如下所示（可通过"man 2 dup"命令查看）：

```
#include <unistd.h>

int dup(int oldfd);
```

首先使用此函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

oldfd：需要被复制的文件描述符。

返回值：成功时将返回一个新的文件描述符，由操作系统分配，分配置原则遵循文件描述符分配原则；如果复制失败将返回-1，并且会设置 errno 值。

测试

由前面的介绍可知，复制得到的文件描述符与原文件描述符都指向同一个文件表，所以它们的文件读写偏移量是一样的，那么是不是可以在不使用 O_APPEND 标志的情况下，通过文件描述符复制来实现接续写，接下来我们编写一个程序进行测试，测试代码如下所示：

示例代码 3.7.1 dup 函数测试代码

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

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
```

```
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 复制文件描述符 */
    fd2 = dup(fd1);
    if (-1 == fd2) {
        perror("dup error");
        ret = -1;
        goto err1;
    }

    printf("fd1: %d\nfd2: %d\n", fd1, fd2);

    /* buffer 数据初始化 */
    buffer1[0] = 0x11;
    buffer1[1] = 0x22;
    buffer1[2] = 0x33;
    buffer1[3] = 0x44;

    buffer2[0] = 0xAA;
    buffer2[1] = 0xBB;
    buffer2[2] = 0xCC;
    buffer2[3] = 0xDD;

    /* 循环写入数据 */
    for (i = 0; i < 4; i++) {

        ret = write(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("write error");
            goto err2;
        }

        ret = write(fd2, buffer2, sizeof(buffer2));
        if (-1 == ret) {
```

```
                perror("write error");
                goto err2;
            }
        }

        /* 将读写位置偏移量移动到文件头 */
        ret = lseek(fd1, 0, SEEK_SET);
        if (-1 == ret) {
            perror("lseek error");
            goto err2;
        }

        /* 读取数据 */
        for (i = 0; i < 8; i++) {

            ret = read(fd1, buffer1, sizeof(buffer1));
            if (-1 == ret) {
                perror("read error");
                goto err2;
            }

            printf("%x%x%x%x", buffer1[0], buffer1[1],
                    buffer1[2], buffer1[3]);
        }

        printf("\n");
        ret = 0;
err2:
        close(fd2);

err1:
        /* 关闭文件 */
        close(fd1);
        exit(ret);
}
```

测试代码中，我们使用了 dup 系统调用复制了文件描述符 fd1，得到另一个新的文件描述符 fd2，分别通过 fd1 和 fd2 对文件进行写操作，最后读取写入的数据来判断是分别写还是接续写，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 7
11223344aabbccdd11223344aabbccdd11223344aabbccdd11223344aabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.7.2 dup 测试结果

由打印信息可知，fd1 等于 6，复制得到的新的文件描述符为 7（遵循 fd 分配原则），打印出来的数据显示为接续写，所以可知，通过复制文件描述符可以实现接续写。

3.7.2 dup2 函数

dup 系统调用分配的文件描述符是由系统分配的，遵循文件描述符分配原则，并不能自己指定一个文件描述符，这是 dup 系统调用的一个缺陷；而 dup2 系统调用修复了这个缺陷，可以手动指定文件描述符，而不需要遵循文件描述符分配原则，当然在实际的编程工作中，需要根据自己的情况来进行选择。

dup2 函数原型如下所示（可以通过"man 2 dup2"命令查看）：

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

同样使用该命令也需要包含<unistd.h>头文件。

函数参数和返回值含义如下：

oldfd：需要被复制的文件描述符。

newfd：指定一个文件描述符（需要指定一个当前进程没有使用到的文件描述符）。

返回值：成功时将返回一个新的文件描述符，也就是手动指定的文件描述符 newfd；如果复制失败将返回-1，并且会设置 errno 值。

测试

接下来编写一个简单地测试程序，如下所示：

示例代码 3.7.2 dup2 函数测试代码

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

int main(void)
{
    int fd1, fd2;
    int ret;

    /* 创建新文件 test_file 并打开 */
```

```
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 复制文件描述符 */
    fd2 = dup2(fd1, 100);
    if (-1 == fd2) {
        perror("dup error");
        ret = -1;
        goto err1;
    }

    printf("fd1: %d\nfd2: %d\n", fd1, fd2);
    ret = 0;

    close(fd2);
err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

测试代码使用 dup2 函数复制文件描述符 fd1，指定新的文件描述符为 100，复制成功之后将其打印出来，结果如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 100
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.7.3 dup2 函数测试结果

由打印信息可知，复制得到的文件描述符 fd2 等于 100，正是我们在 dup2 函数中指定的文件描述符。本小节的内容到这里结束了，最后再强调一点，文件描述符并不是只能复制一次，实际上可以对同一个文件描述符 fd 调用 dup 或 dup2 函数复制多次，得到多个不同的文件描述符。

3.8 文件共享

什么是文件共享？所谓文件共享指的是同一个文件（譬如磁盘上的同一个文件，对应同一个 inode）被多个独立的读写体同时进行 IO 操作。多个独立的读写体大家可以将其简单地理解为对应于同一个文件的多个不同的文件描述符，譬如多次打开同一个文件所得到的多个不同的 fd，或使用 dup()（或 dup2）函数复制得到的多个不同的 fd 等。

同时进行 IO 操作指的是一个读写体操作文件尚未调用 close 关闭的情况下，另一个读写体去操作文件，前面给大家编写的示例代码中就已经涉及到了文件共享的内容了，譬如 3.6 小节中编写的示例代码中，同一个文件对应两个不同的文件描述符 fd1 和 fd2，当使用 fd1 对文件进行写操作之后，并没有关闭 fd1，而此时使用 fd2 对文件再进行写操作，这其实就是一种文件共享。

文件共享的意义有很多，多用于多进程或多线程编程环境中，譬如我们可以通过文件共享的方式来实现多个线程同时操作同一个大文件，以减少文件读写时间、提升效率。

文件共享的核心是：如何制造出多个不同的文件描述符来指向同一个文件。其实方法在上面的内容中都已经给大家介绍过了，譬如多次调用 open 函数重复打开同一个文件得到多个不同的文件描述符、使用 dup()或 dup2()函数对文件描述符进行复制以得到多个不同的文件描述符。

常见的三种文件共享的实现方式

(1)同一个进程中多次调用 open 函数打开同一个文件，各数据结构之间的关系如下图所示：

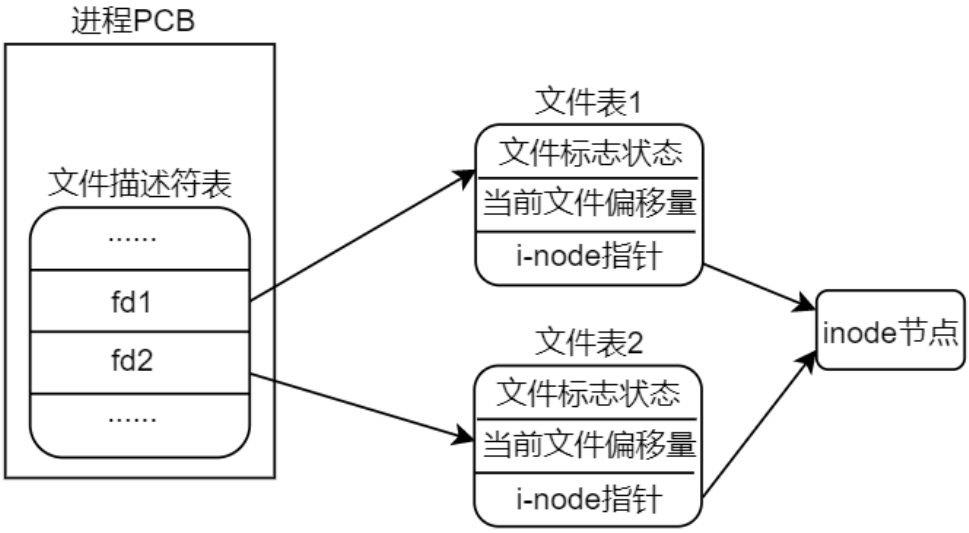


图 3.8.1 同一进程多次 open 打开同一文件各数据结构关系图

这种情况非常简单，多次调用 open 函数打开同一个文件会得到多个不同的文件描述符，并且多个文件描述符对应多个不同的文件表，所有的文件表都索引到了同一个 inode 节点，也就是磁盘上的同一个文件。

(2)不同进程中分别使用 open 函数打开同一个文件，其数据结构关系图如下所示：

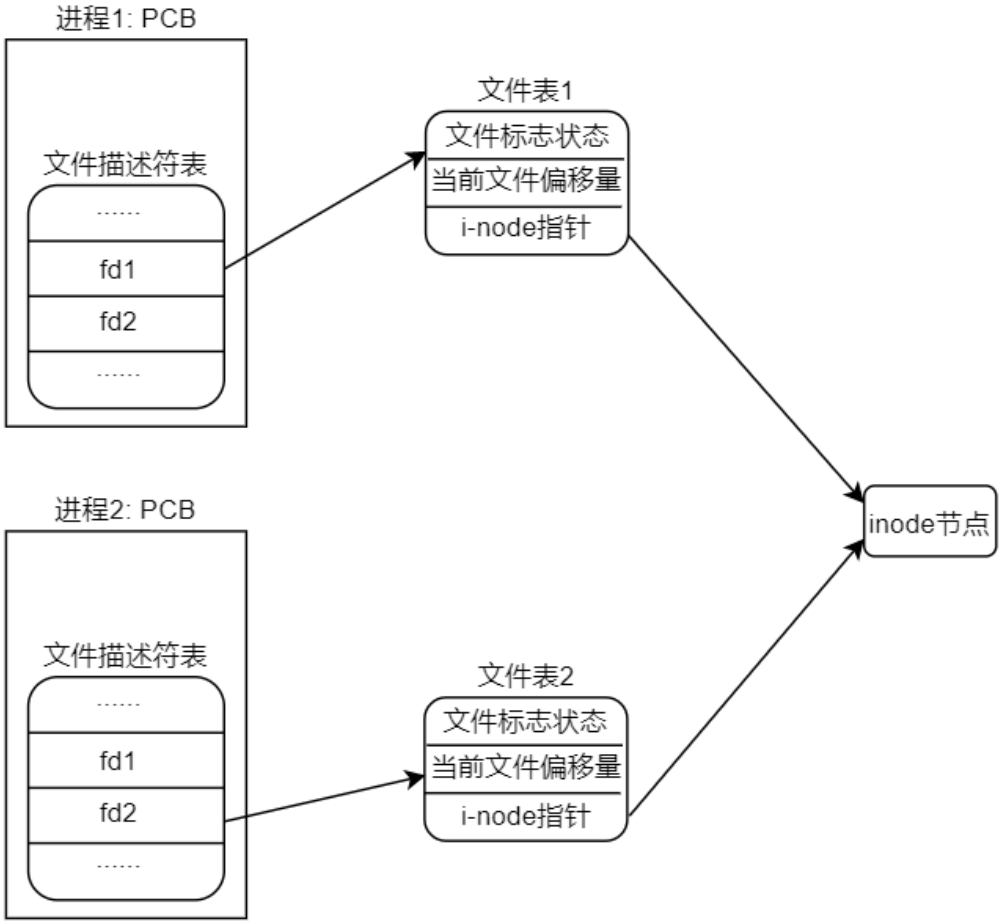


图 3.8.2 不同进程 open 打开同一文件数据结构关系图

进程 1 和进程 2 分别是运行在 Linux 系统上两个独立的进程（理解为两个独立的程序），在他们各自的程序中分别调用 open 函数打开同一个文件，进程 1 对应的文件描述符为 fd1，进程 2 对应的文件描述符为 fd2，fd1 指向了进程 1 的文件表 1，fd2 指向了进程 2 的文件表 2；各自的文件表都索引到了同一个 inode 节点，从而实现共享文件。

(3)同一个进程中通过 dup（dup2）函数对文件描述符进行复制，其数据结构关系如下图所示：

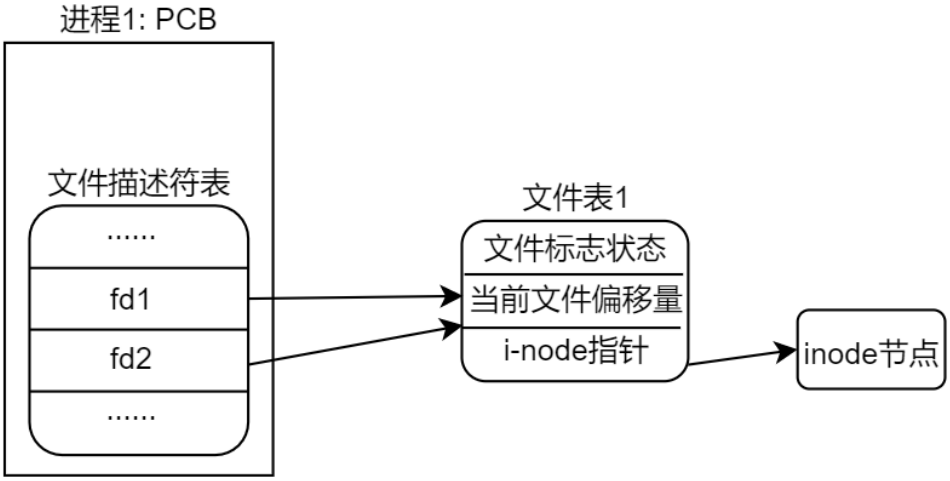


图 3.8.3 dup 复制文件描述符实现文件共享

这种方式上一小节已经给大家进行了详细讲解，这里不再重述！

对于文件共享，存在着竞争冒险，这个是需要大家关注的，下一小节将会向大家介绍。除此之外，我们还需要关心的是文件共享时，不同的读写体之间是分别写还是接续写，这些细节问题大家都要搞清楚。

3.9 原子操作与竞争冒险

Linux 是一个多任务、多进程操作系统，系统中往往运行着多个不同的进程、任务，多个不同的进程就有可能对同一个文件进行 IO 操作，此时该文件便是它们的共享资源，它们共同操作着同一份文件；操作系统级编程不同于大家以前接触的裸机编程，裸机程序中不存在进程、多任务这种概念，而在 Linux 系统中，我们必须要留意到多进程环境下可能会导致的竞争冒险。

3.9.1 竞争冒险简介

本小节给大家竞争冒险这个概念，如果学习过 Linux 驱动开发的读者对这些概念应该并不陌生，也就意味着竞争冒险不但存在于 Linux 应用层、也存在于 Linux 内核驱动层。

假设有两个独立的进程 A 和进程 B 都对同一个文件进行追加写操作（也就是在文件末尾写入数据），每一个进程都调用了 open 函数打开了该文件，但未使用 O_APPEND 标志，此时，各数据结构之间的关系如图 3.8.2 所示。每个进程都有它自己的进程控制块 PCB，有自己的文件表（意味着有自己独立的读写位置偏移量），但是共享同一个 inode 节点（也就是对应同一个文件）。假定此时进程 A 处于运行状态，B 未处于等待运行状态，进程 A 调用了 lseek 函数，它将进程 A 的该文件当前位置偏移量设置为 1500 字节处（假设这里是文件末尾），刚好此时进程 A 的时间片耗尽，然后内核切换到了进程 B，进程 B 执行 lseek 函数，也将其对该文件的当前位置偏移量设置为 1500 个字节处（文件末尾）。然后进程 B 调用 write 函数，写入了 100 个字节数据，那么此时在进程 B 中，该文件的当前位置偏移量已经移动到了 1600 字节处。B 进程时间片耗尽，内核又切换到了进程 A，使进程 A 恢复运行，当进程 A 调用 write 函数时，是从进程 A 的该文件当前位置偏移量（1500 字节处）开始写入，此时文件 1500 字节处已经不再是文件末尾了，如果还从 1500 字节处写入就会覆盖进程 B 刚才写入到该文件中的数据。

其上述假设工作流程图如下图所示：

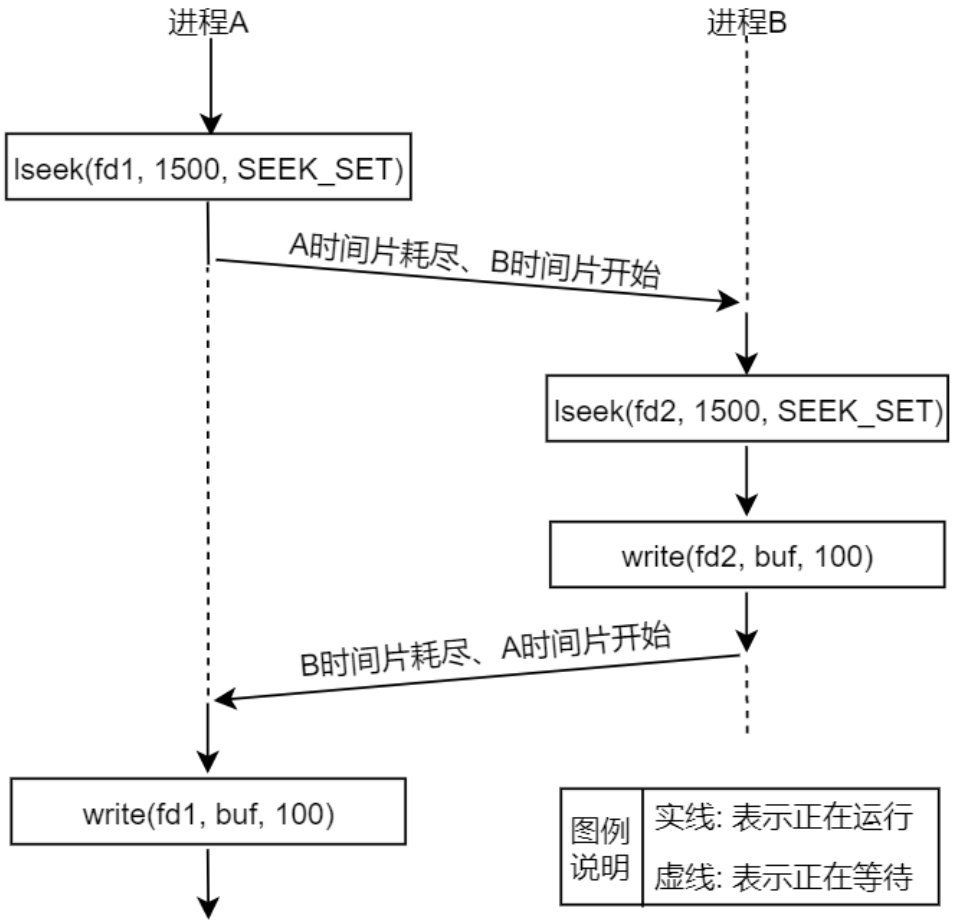


图 3.9.1 假设中的 AB 进程工作流程

以上给大家所描述的这样一种情形就属于竞争状态（也成为竞争冒险），操作共享资源的两个进程（或线程），其操作之后的所得到的结果往往是不可预期的，因为每个进程（或线程）去操作文件的顺序是不可预期的，即这些进程获得 CPU 使用权的先后顺序是不可预期的，完全由操作系统调配，这就是所谓的竞争状态。

既然存在竞争状态，那么该如何规避或消除这种状态呢？接下来给大家介绍原子操作。

3.9.2 原子操作

在上一章给大家介绍 open 函数的时候就提到过“原子操作”这个概念了，同样在 Linux 驱动编程中，也有这个概念，相信学习过 Linux 驱动编程开发的读者应该有印象。

从上一小节给大家提到的示例中可知，上述的问题出在逻辑操作“先定位到文件末尾，然后再写”，它使用了两个分开的函数调用，首先使用 lseek 函数将文件当前位置偏移量移动到文件末尾、然后在使用 write 函数将数据写入到文件。既然知道了问题所在，那么解决办法就是将这两个操作步骤合并成一个原子操作，所谓原子操作，是有多步操作组成的一个操作，原子操作要么一步也不执行，一旦执行，必须要执行完所有步骤，不可能只执行所有步骤中的一个子集。

(1)O_APPEND 实现原子操作

在上一小节给大家提到的示例中，进程 A 和进程 B 都对同一个文件进行追加写操作，导致进程 A 写入的数据覆盖了进程 B 写入的数据，解决办法就是将“先定位到文件末尾，然后写”这两个步骤组成一个原子操作即可，那如何使其变成一个原子操作呢？答案就是 O_APPEND 标志。

前面已经给大家多次提到过了 O_APPEND 标志，但是并没有给大家介绍 O_APPEND 的一个非常重要的作用，那就是实现原子操作。当 open 函数的 flags 参数中包含了 O_APPEND 标志，每次执行 write 写入操作时都会将文件当前写位置偏移量移动到文件末尾，然后再写入数据，这里“移动当前写位置偏移量到文件末尾、写入数据”这两个操作步骤就组成了一个原子操作，加入 O_APPEND 标志后，不管怎么写入数据都会是从文件末尾写，这样就不会导致出现“进程 A 写入的数据覆盖了进程 B 写入的数据”这种情况了。

(2)pread()和 pwrite()

pread()和 pwrite()都是系统调用，与 read()、write()函数的作用一样，用于读取和写入数据。区别在于，pread()和 pwrite()可用于实现原子操作，调用 pread 函数或 pwrite 函数可传入一个位置偏移量 offset 参数，用于指定文件当前读或写的位置偏移量，所以调用 pread 相当于调用 lseek 后再调用 read；同理，调用 pwrite 相当于调用 lseek 后再调用 write。所以可知，使用 pread 或 pwrite 函数不需要使用 lseek 来调整当前位置偏移量，并会将“移动当前位置偏移量、读或写”这两步操作组成一个原子操作。

pread、pwrite 函数原型如下所示（可通过"man 2 pread"或"man 2 pwrite"命令来查看）：

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

首先调用这两个函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

fd、buf、count 参数与 read 或 write 函数意义相同。

offset：表示当前需要进行读或写的位置偏移量。

返回值：返回值与 read、write 函数返回值意义一样。

虽然 pread（或 pwrite）函数相当于 lseek 与 pread（或 pwrite）函数的集合，但还是有下列区别：

- 调用 pread 函数时，无法中断其定位和读操作（也就是原子操作）；
- 不更新文件表中的当前位置偏移量。

关于第二点我们可以编写一个简单地代码进行测试，测试代码如下所示：

示例代码 3.9.1 pread 函数测试

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

int main(void)
{
    unsigned char buffer[100];
    int fd;
    int ret;

    /* 打开文件 test_file */
    fd = open("./test_file", O_RDWR);
    if (-1 == fd) {
        perror("open error");
```

```
        exit(-1);
    }

    /* 使用 pread 函数读取数据(从偏移文件头 1024 字节处开始读取) */
    ret = pread(fd, buffer, sizeof(buffer), 1024);
    if (-1 == ret) {
        perror("pread error");
        goto err;
    }

    /* 获取当前位置偏移量 */
    ret = lseek(fd, 0, SEEK_CUR);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    printf("Current Offset: %d\n", ret);
    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

在当前目录下存在一个文件 test_file，上述代码中会打开 test_file 文件，然后直接使用 pread 函数读取 100 个字节数据，从偏移文件头部 1024 字节处，读取完成之后再使用 lseek 函数获取到文件当前位置偏移量，并将其打印出来。假如 pread 函数会改变文件表中记录的当前位置偏移量，则打印出来的数据应该是 1024 + 100 = 1124；如果不会改变文件表中记录的当前位置偏移量，则打印出来的数据应该是 0，接下来编译代码测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Current Offset: 0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.9.2 pread 函数测试结果

从上图中可知，打印出来的数据为 0，正如前面所介绍那样，pread 函数确实不会改变文件表中记录的当前位置偏移量；同理，pwrite 函数也是如此，大家可以把 pread 换成 pwrite 函数再次进行测试，不出意外，打印出来的数据依然是 0。

如果把 pread 函数换成 read（或 write）函数，那么打印出来的数据就是 100 了，因为读取了 100 个字节数据，相应的当前位置偏移量会向后移动 100 个字节。

(3)创建一个文件

前面给大家介绍 open 函数的 O_EXCL 标志的时候，也提到了原子操作，其中介绍到：O_EXCL 可以用于测试一个文件是否存在，如果不存在则创建此文件，如果存在则返回错误，这使得测试和创建两者成为一个原子操作。接下来给大家，创建文件中存在着的一个竞争状态。

假设有这么一个情况：进程 A 和进程 B 都要去打开同一个文件、并且此文件还不存在。进程 A 当前正在运行状态、进程 B 处于等待状态，进程 A 首先调用 open("./file", O_RDWR)函数尝试去打开文件，结果返回错误，也就是调用 open 失败；接着进程 A 时间片耗尽、进程 B 运行，同样进程 B 调用 open("./file", O_RDWR)尝试打开文件，结果也失败，接着进程 B 再次调用 open("./file", O_RDWR | O_CREAT, ...)创建此文件，这一次 open 执行成功，文件创建成功；接着进程 B 时间片耗尽、进程 A 继续运行，进程 A 也调用 open("./file", O_RDWR | O_CREAT, ...)创建文件，函数执行成功，如下图所示：

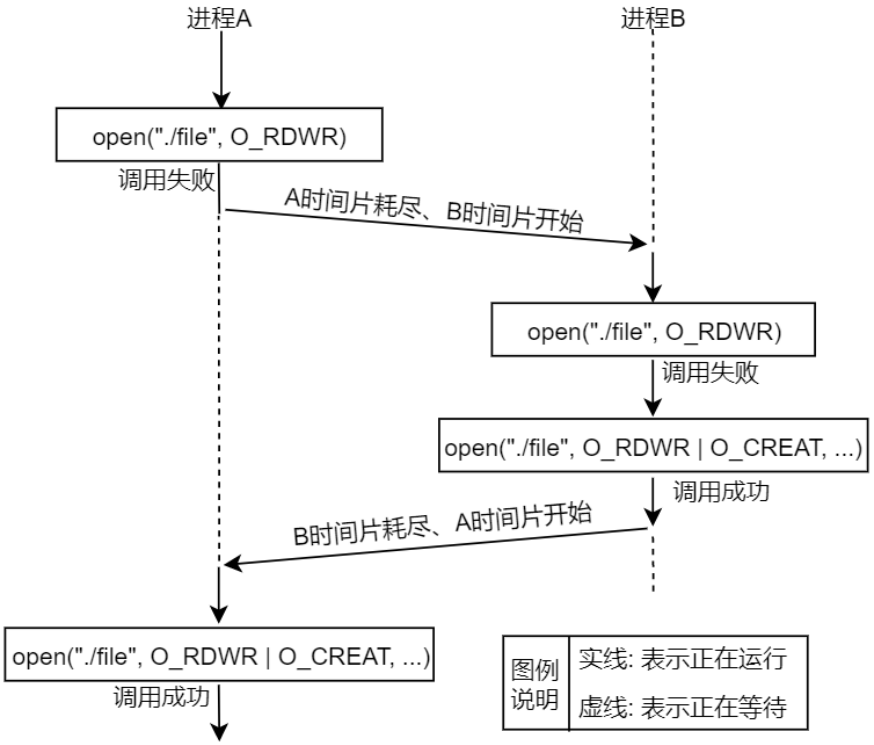


图 3.9.3 创建文件中存在的竞态

从上面的示例可知，进程 A 和进程 B 都会创建出同一个文件，同一个文件被创建两次这是不允许的，那如何规避这样的问题呢？那就是通过使用 O_EXCL 标志，当 open 函数中同时指定了 O_EXCL 和 O_CREAT 标志，如果要打开的文件已经存在，则 open 返回错误；如果指定的文件不存在，则创建这个文件，这里就提供了一种机制，保证进程是打开文件的创建者，将“判断文件是否存在、创建文件”这两个步骤合成为一个原子操作，有了原子操作，就保证不会出现图 3.9.3 中所示的情况。

3.10 fcntl 和 ioctl

本小节给大家介绍两个新的系统调用：fcntl()和 ioctl()。

3.10.1 fcntl 函数

fcntl()函数可以对一个已经打开的文件描述符执行一系列控制操作，譬如复制一个文件描述符（与 dup、dup2 作用相同）、获取/设置文件描述符标志、获取/设置文件状态标志等，类似于一个多功能文件描述符管理工具箱。fcntl()函数原型如下所示（可通过"man 2 fcntl"命令查看）：

```
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, ... /* arg */ )
```

函数参数和返回值含义如下：

fd：文件描述符。

cmd：操作命令。此参数表示我们将要对 fd 进行什么操作，cmd 参数支持很多操作命令，大家可以打开 man 手册查看到这些操作命令的详细介绍，这些命令都是以 F_XXX 开头的，譬如 F_DUPFD、F_GETFD、F_SETFD 等，不同的 cmd 具有不同的作用，cmd 操作命令大致可以分为以下 5 种功能：

- 复制文件描述符（cmd=F_DUPFD 或 cmd=F_DUPFD_CLOEXEC）；
- 获取/设置文件描述符标志（cmd=F_GETFD 或 cmd=F_SETFD）；
- 获取/设置文件状态标志（cmd=F_GETFL 或 cmd=F_SETFL）；
- 获取/设置异步 IO 所有权（cmd=F_GETOWN 或 cmd=F_SETOWN）；
- 获取/设置记录锁（cmd=F_GETLK 或 cmd=F_SETLK）；

这里列举出来，并不需要全部学会每一个 cmd 的作用，因为有些内容并没有给大家提及到，譬如什么异步 IO、锁之类的概念，在后面的学习过程中，当学习到相关知识内容的时候再给大家介绍。

…：fcntl 函数是一个可变参函数，第三个参数需要根据不同的 cmd 来传入对应的实参，配合 cmd 来使用。

返回值：执行失败情况下，返回-1，并且会设置 errno；执行成功的情况下，其返回值与 cmd（操作命令）有关，譬如 cmd=F_DUPFD（复制文件描述符）将返回一个新的文件描述符、cmd=F_GETFD（获取文件描述符标志）将返回文件描述符标志、cmd=F_GETFL（获取文件状态标志）将返回文件状态标志等。

fcntl 使用示例

(1)复制文件描述符

前面给大家介绍了 dup 和 dup2，用于复制文件描述符，除此之外，我们还可以通过 fcntl 函数复制文件描述符，可用的 cmd 包括 F_DUPFD 和 F_DUPFD_CLOEXEC，这里就只介绍 F_DUPFD，F_DUPFD_CLOEXEC 暂时先不讲。

当 cmd=F_DUPFD 时，它的作用会根据 fd 复制出一个新的文件描述符，此时需要传入第三个参数，第三个参数用于指出新复制出的文件描述符是一个大于或等于该参数的可用文件描述符（没有使用的文件描述符）；如果第三个参数等于一个已经存在的文件描述符，则取一个大于该参数的可用文件描述符。

测试代码如下所示：

示例代码 3.10.1 fcntl 复制文件描述符

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

int main(void)
{
    int fd1, fd2;
    int ret;

    /* 打开文件 test_file */
```

```
    fd1 = open("./test_file", O_RDONLY);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 使用 fcntl 函数复制一个文件描述符 */
    fd2 = fcntl(fd1, F_DUPFD, 0);
    if (-1 == fd2) {
        perror("fcntl error");
        ret = -1;
        goto err;
    }

    printf("fd1: %d\nfd2: %d\n", fd1, fd2);

    ret = 0;
    close(fd2);
err:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

在当前目录下存在 test_file 文件，上述代码会打开此文件，得到文件描述符 fd1，之后再使用 fcntl 函数复制 fd1 得到新的文件描述符 fd2，并将 fd1 和 fd2 打印出来，接下来编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 7
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.10.1 fcntl 复制文件描述符测试结果

可知复制得到的文件描述符是 7，因为在执行 fcntl 函数时，传入的第三个参数是 0，也就时指定复制得到的新文件描述符必须要大于或等于 0，但是因为 0~6 都已经被占用了，所以分配得到的 fd 就是 7；如果传入的第三个参数是 100，那么 fd2 就会等于 100，大家可以自己动手测试。

(2)获取/设置文件状态标志

cmd=F_GETFL 可用于获取文件状态标志，cmd=F_SETFL 可用于设置文件状态标志。cmd=F_GETFL 时不需要传入第三个参数，返回值成功表示获取到的文件状态标志；cmd=F_SETFL 时，需要传入第三个参数，此参数表示需要设置的文件状态标志。

这些标志指的就是我们在调用 open 函数时传入的 flags 标志，可以指定一个或多个（通过位或 | 运算符组合），但是文件权限标志（O_RDONLY、O_WRONLY、O_RDWR）以及文件创建标志（O_CREAT、O_EXCL、O_NOCTTY、O_TRUNC）不能被设置、会被忽略；在 Linux 系统中，只有 O_APPEND、O_ASYNC、

O_DIRECT、O_NOATIME 以及 O_NONBLOCK 这些标志可以被修改，这里面有些标志并没有给大家介绍过，后面我们在用到的时候再给大家介绍。所以对于一个已经打开的文件描述符，可以通过这种方式添加或移除标志。

测试代码如下：

示例代码 3.10.2 fcntl 读取/设置文件状态标志

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

int main(void)
{
    int fd;
    int ret;
    int flag;

    /* 打开文件 test_file */
    fd = open("./test_file", O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 获取文件状态标志 */
    flag = fcntl(fd, F_GETFL);
    if (-1 == flag) {
        perror("fcntl F_GETFL error");
        ret = -1;
        goto err;
    }

    printf("flags: 0x%x\n", flag);

    /* 设置文件状态标志,添加 O_APPEND 标志 */
    ret = fcntl(fd, F_SETFL, flag | O_APPEND);
    if (-1 == ret) {
        perror("fcntl F_SETFL error");
        goto err;
    }

    ret = 0;
```

```
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

上述代码会打开 test_file 文件，得到文件描述符 fd，之后调用 fcntl(fd, F_GETFL)来获取到当前文件状态标志 flag，并将其打印来；接着调用 fcntl(fd, F_SETFL, flag|O_APPEND)设置文件状态标志，在原标志的基础上添加 O_APPEND 标志。接下来编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
flags: 0x8002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.10.2 fcntl 测试结果 2

以上给大家介绍了 fcntl 函数的两种用法，除了这两种用法之外，还有其它多种不同的用法，这里暂时先不介绍了，后面学习到相应知识点的时候再给大家讲解。

3.10.2 ioctl 函数

ioctl()可以认为是一个文件 IO 操作的杂物箱，可以处理的事情非常杂、不统一，一般用于操作特殊文件或硬件外设，譬如可以通过 ioctl 获取 LCD 相关信息等，本小节只是给大家引出这个系统调用，暂时不会用到。此函数原型如下所示（可通过"man 2 ioctl"命令查看）：

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

使用此函数需要包含头文件<sys/ioctl.h>。

函数参数和返回值含义如下：

fd：文件描述符。

request：此参数与具体要操作的对象有关，没有统一值，表示向文件描述符请求相应的操作；后面用到的时候再给大家介绍。

...：此函数是一个可变参函数，第三个参数需要根据 request 参数来决定，配合 request 来使用。

返回值：成功返回 0，失败返回-1。

关于 ioctl 函数就给大家介绍这么多。

3.11 截断文件

使用系统调用 truncate()或 ftruncate()可将普通文件截断为指定字节长度，其函数原型如下所示：

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

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

这两个函数的区别在于：ftruncate()使用文件描述符 fd 来指定目标文件，而 truncate()则直接使用文件路径 path 来指定目标文件，其功能一样。

这两个函数都可以对文件进行截断操作，将文件截断为参数 length 指定的字节长度，什么是截断？如果文件目前的大小大于参数 length 所指定的大小，则多余的数据将被丢失，类似于多余的部分被“砍”掉了；如果文件目前的大小小于参数 length 所指定的大小，则将其进行扩展，对扩展部分进行读取将得到空字节"\0"。

使用 ftruncate()函数进行文件截断操作之前，必须调用 open()函数打开该文件得到文件描述符，并且必须要具有可写权限，也就是调用 open()打开文件时需要指定 O_WRONLY 或 O_RDWR。

调用这两个函数并不会导致文件读写位置偏移量发生改变，所以截断之后一般需要重新设置文件当前的读写位置偏移量，以免由于之前所指向的位置已经不存在而发生错误（譬如文件长度变短了，文件当前所指向的读写位置已不存在）。

调用成功返回 0，失败将返回-1，并设置 errno 以指示错误原因。

使用示例

示例代码 3.11.1 演示了文件的截断操作，分别使用 ftruncate()和 truncate()将当前目录下的文件 file1 截断为长度 0、将文件 file2 截断为长度 1024 个字节。

示例代码 3.11.1 文件截断操作

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

int main(void)
{
    int fd;

    /* 打开 file1 文件 */
    if (0 > (fd = open("./file1", O_RDWR))) {
        perror("open error");
        exit(-1);
    }

    /* 使用 ftruncate 将 file1 文件截断为长度 0 字节 */
    if (0 > ftruncate(fd, 0)) {
        perror("ftruncate error");
        exit(-1);
    }

    /* 使用 truncate 将 file2 文件截断为长度 1024 字节 */
    if (0 > truncate("./file2", 1024)) {
        perror("truncate error");
        exit(-1);
```

```
    }

    /* 关闭 file1 退出程序 */
    close(fd);
    exit(0);
}
```

上述代码中，首先使用 open()函数打开文件 file1，得到文件描述符 fd，接着使用 ftruncate()系统调用将文件截断为 0 长度，传入 file1 文件对应的文件描述符；接着调用 truncate()系统调用将文件 file2 截断为 1024 字节长度，传入 file2 文件的相对路径。

接下来进行测试，在当前目录下准备两个文件 file1 和 file2，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1  file2  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 12
-rw-rw-r-- 1 dt dt 592 5月  28 18:25 file1
-rw-rw-r-- 1 dt dt 592 5月  28 18:25 file2
-rw-rw-r-- 1 dt dt 592 5月  28 18:25 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.11.1 准备 file1 文件和 file2 文件

可以看到 file1 和 file2 文件此时均为 592 字节大小，接下来运行测试代码：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1  file2  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1  file2  testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l file1 file2
-rw-rw-r-- 1 dt dt    0 5月  28 18:40 file1
-rw-rw-r-- 1 dt dt 1024 5月  28 18:40 file2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.11.2 测试结果

程序运行之后，file1 文件大小变成了 0，而 file2 文件大小变成了 1024 字节，与测试代码想要实现的功能是一致的。

第四章 标准 I/O 库

本章介绍标准 I/O 库，不仅是 Linux，很多其它的操作系统都实现了标准 I/O 库。标准 I/O 虽然是对文件 I/O 进行了封装，但事实上并不仅仅只是如此，标准 I/O 会处理很多细节，譬如分配 stdio 缓冲区、以优化的块长度执行 I/O 等，这些处理使用户不必担心如何选择使用正确的块长度。

本章将会讨论如下主题内容。

- 标准 I/O 库简介；
- 流和 FILE 对象；
- 标准输入、标准输出以及标准错误；
- 使用标准 I/O 库函数打开、读写、关闭文件；
- 格式化 I/O，格式化输出 printf、格式化输入 scanf；
- 文件 I/O 缓冲，内核缓冲区和 stdio 缓冲区；
- 文件 I/O 与标准 I/O 混合编程。

4.1 标准 I/O 库简介

在第一章介绍应用编程概念时向大家介绍了系统调用与标准 C 语言函数库（以下简称标准 C 库），所谓标准 I/O 库则是标准 C 库中用于文件 I/O 操作（譬如读文件、写文件等）相关的一系列库函数的集合，通常标准 I/O 库函数相关的函数定义都在头文件<stdio.h>中，所以我们需要在程序源码中包含<stdio.h>头文件。

标准 I/O 库函数是构建于文件 I/O（open()、read()、write()、lseek()、close()等）这些系统调用之上的，譬如标准 I/O 库函数 fopen()就利用系统调用 open()来执行打开文件的操作、fread()利用系统调用 read()来执行读文件操作、fwrite()则利用系统调用 write()来执行写文件操作等等。

那既然如此，为何还需要设计标准 I/O 库？直接使用文件 I/O 系统调用不是更好吗？事实上，并非如此，在第一章中我们也提到过，设计库函数是为了提供比底层系统调用更为方便、好用的调用接口，虽然标准 I/O 构建于文件 I/O 之上，但标准 I/O 却有它自己的优势，标准 I/O 和文件 I/O 的区别如下：

- 虽然标准 I/O 和文件 I/O 都是 C 语言函数，但是标准 I/O 是标准 C 库函数，而文件 I/O 则是 Linux 系统调用；
- 标准 I/O 是由文件 I/O 封装而来，标准 I/O 内部实际上是调用文件 I/O 来完成实际操作的；
- 可移植性：标准 I/O 相比于文件 I/O 具有更好的可移植性，通常对于不同的操作系统，其内核向应用层提供的系统调用往往都是不同，譬如系统调用的定义、功能、参数列表、返回值等往往都是不一样的；而对于标准 I/O 来说，由于很多操作系统都实现了标准 I/O 库，标准 I/O 库在不同的操作系统之间其接口定义几乎是一样的，所以标准 I/O 在不同操作系统之间相比于文件 I/O 具有更好的可移植性。
- 性能、效率：标准 I/O 库在用户空间维护了自己的 stdio 缓冲区，所以标准 I/O 是带有缓存的，而文件 I/O 在用户空间是不带有缓存的，所以在性能、效率上，标准 I/O 要优于文件 I/O。

关于标准 I/O 库相关介绍就到这里了，从下小节开始将正式向大家介绍如何在我们的应用程序中使用标准 I/O 库函数。

4.2 FILE 指针

在 0 中，所介绍的所有文件 I/O 函数（open()、read()、write()、lseek()等）都是围绕文件描述符进行的，当调用 open()函数打开一个文件时，即返回一个文件描述符 fd，然后该文件描述符就用于后续的 I/O 操作。而对于标准 I/O 库函数来说，它们的操作是围绕 FILE 指针进行的，当使用标准 I/O 库函数打开或创建一个文件时，会返回一个指向 FILE 类型对象的指针（FILE *），使用该 FILE 指针与被打开或创建的文件相关联，然后该 FILE 指针就用于后续的标准 I/O 操作（使用标准 I/O 库函数进行 I/O 操作），所以由此可知，FILE 指针的作用相当于文件描述符，只不过 FILE 指针用于标准 I/O 库函数中、而文件描述符则用于文件 I/O 系统调用中。

FILE 是一个结构体数据类型，它包含了标准 I/O 库函数为管理文件所需要的所有信息，包括用于实际 I/O 的文件描述符、指向文件缓冲区的指针、缓冲区的长度、当前缓冲区中的字节数以及出错标志等。FILE 数据结构定义在标准 I/O 库函数头文件 stdio.h 中。

4.3 标准输入、标准输出和标准错误

关于标准输入、标准输出以及标准错误这三个概念在 2.2 小节有所提及，所谓标准输入设备指的就是计算机系统的标准的输入设备，通常指的是计算机所连接的键盘；而标准输出设备指的是计算机系统中用于输出标准信息的设备，通常指的是计算机所连接的显示器；标准错误设备则指的是计算机系统中用于显示错误信息的设备，通常也指的是显示器设备。

用户通过标准输入设备与系统进行交互，进程将从标准输入（stdin）文件中得到输入数据，将正常输出数据（譬如程序中 printf 打印输出的字符串）输出到标准输出（stdout）文件，而将错误信息（譬如函数调用报错打印的信息）输出到标准错误（stderr）文件。

标准输出文件和标准错误文件都对应终端的屏幕，而标准输入文件则对应于键盘。

每个进程启动之后都会默认打开标准输入、标准输出以及标准错误，得到三个文件描述符，即 0、1、2，其中 0 代表标准输入、1 代表标准输出、2 代表标准错误；在应用编程中可以使用宏 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO 分别代表 0、1、2，这些宏定义在 unistd.h 头文件中：

```
/* Standard file descriptors.  */
#define	STDIN_FILENO	0	/* Standard input.  */
#define	STDOUT_FILENO	1	/* Standard output.  */
#define	STDERR_FILENO	2	/* Standard error output.  */
```

0、1、2 这三个是文件描述符，只能用于文件 I/O（read()、write()等），那么在标准 I/O 中，自然是无法使用文件描述符来对文件进行 I/O 操作的，它们需要围绕 FILE 类型指针来进行，在 stdio.h 头文件中有相应的定义，如下：

```
/* 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.  */
/* C89/C99 say they're macros.  Make them happy.  */
#define stdin stdin
#define stdout stdout
#define stderr stderr
```

Tips：struct _IO_FILE 结构体就是 FILE 结构体，使用了 typedef 进行了重命名。

所以，在标准 I/O 中，可以使用 stdin、stdout、stderr 来表示标准输入、标准输出和标准错误。

4.4 打开文件 fopen()

在 0 所介绍的文件 I/O 中，使用 open()系统调用打开或创建文件，而在标准 I/O 中，我们将使用库函数 fopen()打开或创建文件，fopen()函数原型如下所示：

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
```

使用该函数需要包含头文件 stdio.h。

函数参数和返回值含义如下：

path：参数 path 指向文件路径，可以是绝对路径、也可以是相对路径。

mode：参数 mode 指定了对该文件的读写权限，是一个字符串，稍后介绍。

返回值：调用成功返回一个指向 FILE 类型对象的指针（FILE *），该指针与打开或创建的文件相关联，后续的标准 I/O 操作将围绕 FILE 指针进行。如果失败则返回 NULL，并设置 errno 以指示错误原因。

参数 mode 字符串类型，可取值为如下值之一：

mode	说明	对应于 open()函数的 flags 参数取值
r	以只读方式打开文件。	O_RDONLY
r+	以可读、可写方式打开文件。	O_RDWR

w	以只写方式打开文件，如果参数 path 指定的文件存在，将文件长度截断为 0；如果指定文件不存在则创建该文件。	O_WRONLY | O_CREAT | O_TRUNC
w+	以可读、可写方式打开文件，如果参数 path 指定的文件存在，将文件长度截断为 0；如果指定文件不存在则创建该文件。	O_RDWR | O_CREAT | O_TRUNC
a	以只写方式打开文件，打开以进行追加内容（在文件末尾写入），如果文件不存在则创建该文件。	O_WRONLY | O_CREAT | O_APPEND
a+	以可读、可写方式打开文件，以追加方式写入（在文件末尾写入），如果文件不存在则创建该文件。	O_RDWR | O_CREAT | O_APPEND

表 4.4.1 标注 I/O fopen()函数的 mode 参数

新建文件的权限

由 fopen()函数原型可知，fopen()只有两个参数 path 和 mode，不同于 open()系统调用，它并没有任何一个参数来指定新建文件的权限。当参数 mode 取值为"w"、"w+"、"a"、"a+"之一时，如果参数 path 指定的文件不存在，则会创建该文件，那么新的文件的权限是如何确定的呢？

虽然调用 fopen()函数新建文件时无法手动指定文件的权限，但却有一个默认值：

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH (0666)
```

使用示例

使用只读方式打开文件：

```
fopen(path, "r");
```

使用可读、可写方式打开文件：

```
fopen(path, "r+");
```

使用只写方式打开文件，并将文件长度截断为 0，如果文件不存在则创建该文件：

```
fopen(path, "w");
```

fclose()关闭文件

调用 fclose()库函数可以关闭一个由 fopen()打开的文件，其函数原型如下所示：

```
#include <stdio.h>

int fclose(FILE *stream);
```

参数 stream 为 FILE 类型指针，调用成功返回 0；失败将返回 EOF（也就是-1），并且会设置 errno 来指示错误原因。

4.5 读文件和写文件

当使用 fopen()库函数打开文件之后，接着我们便可以使用 fread()和 fwrite()库函数对文件进行读、写操作了，函数原型如下所示：

```
#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);
```

库函数 fread()用于读取文件数据，其参数和返回值含义如下：

ptr：fread()将读取到的数据存放在参数 ptr 指向的缓冲区中；

size：fread()从文件读取 nmemb 个数据项，每一个数据项的大小为 size 个字节，所以总共读取的数据大小为 nmemb * size 个字节。

nmemb：参数 nmemb 指定了读取数据项的个数。

stream：FILE 指针。

返回值：调用成功时返回读取到的数据项的数目（数据项数目并不等于实际读取的字节数，除非参数 size 等于 1）；如果发生错误或到达文件末尾，则 fread()返回的值将小于参数 nmemb，那么到底发生了错误还是到达了文件末尾，fread()不能区分文件结尾和错误，究竟是哪一种情况，此时可以使用 ferror()或 feof()函数来判断，具体参考 4.7 小节内容的介绍。

库函数 fwrite()用于将数据写入到文件中，其参数和返回值含义如下：

ptr：将参数 ptr 指向的缓冲区中的数据写入到文件中。

size：参数 size 指定了每个数据项的字节大小，与 fread()函数的 size 参数意义相同。

nmemb：参数 nmemb 指定了写入的数据项个数，与 fread()函数的 nmemb 参数意义相同。

stream：FILE 指针。

返回值：调用成功时返回写入的数据项的数目（数据项数目并不等于实际写入的字节数，除非参数 size 等于 1）；如果发生错误，则 fwrite()返回的值将小于参数 nmemb（或者等于 0）。

由此可知，库函数 fread()、fwrite()中指定读取或写入数据大小的方式与系统调用 read()、write()不同，前者通过 nmemb（数据项个数）*size（每个数据项的大小）的方式来指定数据大小，而后者则直接通过一个 size 参数指定数据大小。

譬如要将一个 struct mystr 结构体数据写入到文件中，可按如下方式写入：

```
fwrite(buf, sizeof(struct mystr), 1, file);
```

当然也可以按如下方式写：

```
fwrite(buf, 1, sizeof(struct mystr), file);
```

使用示例

结合使用本小节与上小节所学内容，我们来编写一个简单地示例代码，使用标准 I/O 方式对文件进行读写操作。示例代码 4.5.1 演示了使用 fwrite()库函数将数据写入到文件中。

示例代码 4.5.1 标准 I/O 之 fwrite()写文件

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

int main(void)
{
    char buf[] = "Hello World!\n";
    FILE *fp = NULL;

    /* 打开文件 */
    if (NULL == (fp = fopen("./test_file", "w"))) {
        perror("fopen error");
        exit(-1);
    }

    printf("文件打开成功!\n");
```

```
    /* 写入数据 */
    if (sizeof(buf) >
        fwrite(buf, 1, sizeof(buf), fp)) {
        printf("fwrite error\n");
        fclose(fp);
        exit(-1);
    }

    printf("数据写入成功!\n");

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

首先使用 fopen()函数将当前目录下的 test_file 文件打开，调用 fopen()时 mode 参数设置为"w"，表示以只写的方式打开文件，并将文件的长度截断为 0，如果指定文件不存在则创建该文件。打开文件之后调用 fwrite()函数将"Hello World!"字符串数据写入到文件中。

写入完成之后，调用 fclose()函数关闭文件，退出程序。

编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
数据写入成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.5.1 测试结果

示例代码 4.5.2 演示了使用库函数 fread()从文件中读取数据。

示例代码 4.5.2 标准 I/O 之 fread()读文件

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

int main(void)
{
    char buf[50] = {0};
    FILE *fp = NULL;
    int size;

    /* 打开文件 */
    if (NULL == (fp = fopen("./test_file", "r"))) {
```

```
        perror("fopen error");
        exit(-1);
    }

    printf("文件打开成功!\n");

    /* 读取数据 */
    if (12 > (size = fread(buf, 1, 12, fp))) {
        if (ferror(fp)) {     //使用 ferror 判断是否是发生错误
            printf("fread error\n");
            fclose(fp);
            exit(-1);
        }

        /* 如果未发生错误则意味着已经到达了文件末尾 */
    }

    printf("成功读取%d 个字节数据: %s\n", size, buf);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

首先同样使用 fopen()打开当前目录下的 test_file 文件得到 FILE 指针，调用 fopen()时其参数 mode 设置为"r"，表示以只读方式打开文件。

接着使用 fread()函数从文件中读取 12 * 1=12 个字节的数据，将读取到的数据存放在 buf 中，当读取到的字节数小于指定字节数时，表示发生了错误或者已经到达了文件末尾，程序中调用了库函数 ferror()来判断是不是发生了错误，该函数将会在 4.7 小节中介绍。如果未发生错误，那么就意味着已经达到了文件末尾，其实也就说明了在调用 fread()读文件时对应的读写位置到文件末尾之间的字节数小于指定的字节数。

最后调用 printf()打印结果，编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
成功读取12个字节数据: Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.5.2 测试结果

4.6 fseek 定位

库函数 fseek()的作用类似于 2.7 小节所学习的系统调用 lseek()，用于设置文件读写位置偏移量，lseek()用于文件 I/O，而库函数 fseek()则用于标准 I/O，其函数原型如下所示：

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
```

函数参数和返回值含义如下：

stream： FILE 指针。

offset： 与 lseek()函数的 offset 参数意义相同。

whence： 与 lseek()函数的 whence 参数意义相同。

返回值： 成功返回 0；发生错误将返回-1，并且会设置 errno 以指示错误原因；与 lseek()函数的返回值意义不同，这里要注意！

调用库函数 fread()、fwrite()读写文件时，文件的读写位置偏移量会自动递增，使用 fseek()可手动设置文件当前的读写位置偏移量。

譬如将文件的读写位置移动到文件开头处：

```
fseek(file, 0, SEEK_SET);
```

将文件的读写位置移动到文件末尾：

```
fseek(file, 0, SEEK_END);
```

将文件的读写位置移动到 100 个字节偏移量处：

```
fseek(file, 100, SEEK_SET);
```

使用示例

示例代码 4.6.1 使用 fseek()调整文件读写位置

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

int main(void)
{
    FILE *fp = NULL;
    char rd_buf[100] = {0};
    char wr_buf[] = "正点原子 http://www.openedv.com/forum.php\n";
    int ret;

    /* 打开文件 */
    if (NULL == (fp = fopen("./test_file", "w+"))) {
        perror("fopen error");
        exit(-1);
    }
    printf("文件打开成功!\n");

    /* 写文件 */
    if (sizeof(wr_buf) >
        fwrite(wr_buf, 1, sizeof(wr_buf), fp)) {
```

```
        printf("fwrite error\n");
        fclose(fp);
        exit(-1);
    }
    printf("数据写入成功!\n");

    /* 将读写位置移动到文件头部 */
    if (0 > fseek(fp, 0, SEEK_SET)) {
        perror("fseek error");
        fclose(fp);
        exit(-1);
    }

    /* 读文件 */
    if (sizeof(wr_buf) >
        (ret = fread(rd_buf, 1, sizeof(wr_buf), fp))) {
        printf("fread error\n");
        fclose(fp);
        exit(-1);
    }

    printf("成功读取%d 个字节数据: %s\n", ret, rd_buf);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

程序中首先调用 fopen()打开当前目录下的 test_file 文件，参数 mode 设置为"w+"；接着调用 fwrite()将 wr_buf 缓冲区中的字符串数据"正点原子 http://www.openedv.com/forum.php"写入到文件中；由于调用了 fwrite()，所以此时的读写位置已经发生了改变，不再是文件头部，所以程序中调用了 fseek()将读写位置移动到了文件头，接着调用 fread()从文件头部开始读取刚写入的数据，读取成功之后打印出信息。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功!
数据写入成功!
成功读取46个字节数据：正点原子http://www.openedv.com/forum.php

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
正点原子http://www.openedv.com/forum.php
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.6.1 测试结果

ftell()函数

库函数 ftell()可用于获取文件当前的读写位置偏移量，其函数原型如下所示：

```
#include <stdio.h>

long ftell(FILE *stream);
```

参数 stream 指向对应的文件，函数调用成功将返回当前读写位置偏移量；调用失败将返回-1，并会设置 errno 以指示错误原因。

我们可以通过 fseek()和 ftell()来计算出文件的大小，示例代码如下所示：

示例代码 4.6.2 使用 fseek()和 ftell()函数获取文件大小

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

int main(void)
{
    FILE *fp = NULL;
    int ret;

    /* 打开文件 */
    if (NULL == (fp = fopen("./testApp.c", "r"))) {
        perror("fopen error");
        exit(-1);
    }

    printf("文件打开成功!\n");

    /* 将读写位置移动到文件末尾 */
    if (0 > fseek(fp, 0, SEEK_END)) {
        perror("fseek error");
        fclose(fp);
        exit(-1);
    }
```

```
    /* 获取当前位置偏移量 */
    if (0 > (ret = ftell(fp))) {
        perror("ftell error");
        fclose(fp);
        exit(-1);
    }

    printf("文件大小: %d 个字节\n", ret);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

首先打开当前目录下的 testApp.c 文件，将文件的读写位置移动到文件末尾，然后再获取当前的位置偏移量，也就得到了整个文件的大小。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l testApp.c
-rw-rw-r-- 1 dt dt 576 5月  18 11:16 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
文件打开成功！
文件大小：576个字节
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.6.2 测试结果

从上图可知，程序计算出的文件大小与 ls 命令查看到的文件大小是一致的。

4.7 检查或复位状态

调用 fread()读取数据时，如果返回值小于参数 nmemb 所指定的值，表示发生了错误或者已经到了文件末尾（文件结束 end-of-file），但 fread()无法具体确定是哪一种情况；在这种情况下，可以通过判断错误标志或 end-of-file 标志来确定具体的情况。

4.7.1 feof()函数

库函数 feof()用于测试参数 stream 所指文件的 end-of-file 标志，如果 end-of-file 标志被设置了，则调用 feof()函数将返回一个非零值，如果 end-of-file 标志没有被设置，则返回 0。

其函数原型如下所示：

```
#include <stdio.h>

int feof(FILE *stream);
```

当文件的读写位置移动到了文件末尾时，end-of-file 标志将会被设置。

```
if (feof(file)) {
    /* 到达文件末尾 */
```

```
}
else {
    /* 未到达文件末尾 */
}
```

4.7.2 ferror()函数

库函数 ferror()用于测试参数 stream 所指文件的错误标志，如果错误标志被设置了，则调用 ferror()函数将返回一个非零值，如果错误标志没有被设置，则返回 0。

其函数原型如下所示：

```
#include <stdio.h>

int ferror(FILE *stream);
```

当对文件的 I/O 操作发生错误时，错误标志将会被设置。

```
if (ferror(file)) {
    /* 发生错误 */
}
else {
    /* 未发生错误 */
}
```

4.7.3 clearerr()函数

库函数 clearerr()用于清除 end-of-file 标志和错误标志，当调用 feof()或 ferror()校验这些标志后，通常需要清除这些标志，避免下次校验时使用到的是上一次设置的值，此时可以手动调用 clearerr()函数清除标志。

clearerr()函数原型如下所示：

```
#include <stdio.h>

void clearerr(FILE *stream);
```

此函数没有返回值，调用将总是会成功！

对于 end-of-file 标志，除了使用 clearerr()显式清除之外，当调用 fseek()成功时也会清除文件的 end-of-file 标志。

使用示例

示例代码 4.7.1 clearerr()函数使用示例

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

int main(void)
{
    FILE *fp = NULL;
    char buf[20] = {0};

    /* 打开文件 */
    if (NULL == (fp = fopen("./testApp.c", "r"))) {
```

```
        perror("fopen error");
        exit(-1);
    }

    printf("文件打开成功!\n");

    /* 将读写位置移动到文件末尾 */
    if (0 > fseek(fp, 0, SEEK_END)) {
        perror("fseek error");
        fclose(fp);
        exit(-1);
    }

    /* 读文件 */
    if (10 > fread(buf, 1, 10, fp)) {
        if (feof(fp))
            printf("end-of-file 标志被设置,已到文件末尾!\n");

        clearerr(fp);     //清除标志
    }

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

4.8 格式化 I/O

在前面编写的测试代码中，会经常使用到库函数 printf()用于输出程序中的打印信息，printf()函数可将格式化数据写入到标准输出，所以通常称为格式化输出。除了 printf()之外，格式化输出还包括：fprintf()、dprintf()、sprintf()、snprintf()这 4 个库函数。

除了格式化输出之外，自然也有格式化输入，从标准输入中获取格式化数据，格式化输入包括：scanf()、fscanf()、sscanf()这三个库函数，那么本小节将向大家介绍 C 语言库函数的格式化 I/O。

4.8.1 格式化输出

C 库函数提供了 5 个格式化输出函数，包括：printf()、fprintf()、dprintf()、sprintf()、snprintf()，其函数定义如下所示：

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *buf, const char *format, ...);
int snprintf(char *buf, size_t size, const char *format, ...);
```

可以看到，这 5 个函数都是可变参函数，它们都有一个共同的参数 format，这是一个字符串，称为格式控制字符串，用于指定后续的参数如何进行格式转换，所以才把这些函数称为格式化输出，因为它们可以以调用者指定的格式进行转换输出；学习这些函数的重点就是掌握这个格式控制字符串 format 的书写格式以及它们所代表的意义，稍后介绍 format 参数的格式。

每个函数除了固定参数之外，还可携带 0 个或多个可变参数。

printf()函数用于将格式化数据写入到标准输出；dprintf()和 fprintf()函数用于将格式化数据写入到指定的文件中，两者不同之处在于，fprintf()使用 FILE 指针指定对应的文件、而 dprintf()则使用文件描述符 fd 指定对应的文件；sprintf()、snprintf()函数可将格式化的数据存储在用户指定的缓冲区 buf 中。

printf()函数

前面章节内容编写的示例代码中多次使用了该函数，用于将程序中的字符串信息输出显示到终端（也就是标准输出），相信各位读者学习 C 语言时肯定用过该函数，它是一个可变参函数，除了一个固定参数 format 外，后面还可携带 0 个或多个参数。

函数调用成功返回打印输出的字符数；失败将返回一个负值！

打印“Hello World”：

```
printf("Hello World!\n");
```

打印数字 5：

```
printf("%d\n", 5);
```

fprintf()函数

fprintf()可将格式化数据写入到由 FILE 指针指定的文件中，譬如将字符串“Hello World”写入到标准错误：

```
fprintf(stderr, "Hello World!\n");
```

向标准错误写入数字 5：

```
fprintf(stderr, "%d\n", 5);
```

函数调用成功返回写入到文件中的字符数；失败将返回一个负值！

dprintf()函数

dprintf()可将格式化数据写入到由文件描述符 fd 指定的文件中，譬如将字符串“Hello World”写入到标准错误：

```
dprintf(STDERR_FILENO, "Hello World!\n");
```

向标准错误写入数字 5：

```
dprintf(STDERR_FILENO, "%d\n", 5);
```

函数调用成功返回写入到文件中的字符数；失败将返回一个负值！

sprintf()函数

sprintf()函数将格式化数据存储在由参数 buf 所指定的缓冲区中，譬如将字符串“Hello World”存放在缓冲区中：

```
char buf[100];
sprintf(buf, "Hello World!\n");
```

当然这种用法并没有意义，事实上，我们一般会使用这个函数进行格式化转换，并将转换后的字符串存放在缓冲区中，譬如将数字 100 转换为字符串"100"，将转换后得到的字符串存放在 buf 中：

```
char buf[20] = {0};
sprintf(buf, "%d", 100);
```

sprintf()函数会在字符串尾端自动加上一个字符串终止字符'\0'。

需要注意的是，sprintf()函数可能会造成由参数 buf 指定的缓冲区溢出，调用者有责任确保该缓冲区足够大，因为缓冲区溢出会造成程序不稳定甚至安全隐患！

函数调用成功返回写入到 buf 中的字节数；失败将返回一个负值！

snprintf()函数

sprintf()函数可能会发生缓冲区溢出的问题，存在安全隐患，为了解决这个问题，引入了 snprintf()函数；在该函数中，使用参数 size 显式的指定缓冲区的大小，如果写入到缓冲区的字节数大于参数 size 指定的大小，超出的部分将会被丢弃！如果缓冲区空间足够大，snprintf()函数就会返回写入到缓冲区的字符数，与 sprintf()函数相同，也会在字符串末尾自动添加终止字符'\0'。

若发生错误，snprintf()将返回一个负值！

格式控制字符串 format

接下来重点学习以上 5 个函数中的 format 参数应该怎么写，把这个参数称为格式控制字符串，顾名思义，首先它是一个字符串的形式，其次它能够控制后续变参的格式转换。

格式控制字符串由两部分组成：普通字符（非%字符）和转换说明。普通字符会进行原样输出，每个转换说明都会对应后续的一个参数，通常有几个转换说明就需要提供几个参数（除固定参数之外的参数），使之一一对应，用于控制对应的参数如何进行转换。如下所示：

```
printf("转换说明 1  转换说明 2  转换说明 3", arg1, arg2, arg3);
```

这里只是以 printf()函数举个例子，实际上并不这样用。三个转换说明与参数进行一一对应，按照顺序方式一一对应。

每个转换说明都是以%字符开头，其格式如下所示（使用[ ]括起来的部分是可选的）：

```
%[flags][width][.precision][length]type
```

flags：标志，可包含 0 个或多个标志；

width：输出最小宽度，表示转换后输出字符串的最小宽度；

precision：精度，前面有一个点号" . "；

length：长度修饰符；

type：转换类型，指定待转换数据的类型。

可以看到，只有%和 type 字段是必须的，其余都是可选的。下面分别对这些字段进行介绍。

㈠、type 类型

首先说明 type（类型），因为类型是格式控制字符串的重中之重，是必不可少的组成部分，其它的字段都是可选的，type 用于指定输出数据的类型，type 字段使用一个字符（字母字符）来表示，可取值如下：

字符	对应的数据类型	含义	示例说明
d/i	int	输出有符号十进制表示的整数，i 是老式写法	printf("%d\n", 123); 输出:123
o	unsigned int	输出无符号八进制表示的整数（默认不输出前缀 0，可在 type 字段指定标志#使其输出前缀 0）	printf("%o\n", 123); 输出:173
u	unsigned int	输出无符号十进制表示的整数	printf("%u\n", 123); 输出:123
x/X	unsigned int	输出无符号十六进制表示的整数，x 和 X 的区别在于字母的大小写问题（x 对应的是 abcdef，X 对应的是	printf("%x\n", 123); 输出:7b printf("%X\n", 123); 输出:7B

		ABCDEF）；不输出前缀 0x 或 0X，可在 type 字段指定标志#使其输出前缀。	

f/F	double	输出浮点数，单精度浮点数类型和双精度浮点数类型都可以使用，f 和 F 之间的区别就不去管了，一般表示浮点数使用 f 即可。在没指定精度的情况下，默认保留小数点后 6 位数字。	printf("%f\n", 520.1314);
			输出:520.131400
			printf("%F\n", 520.1314);
			输出:520.131400

e/E	double	输出以科学计数法表示的浮点数，使用指数(Exponent)表示浮点数，此处 e 和 E 的区别在于以科学计数法表示时，字母“e”的大小写问题。	printf("%e\n", 520.1314);
			输出:5.201314e+02
			printf("%E\n", 520.1314);
			输出:5.201314E+02

g	double	根据数值的长度，选择以最短的方式输出，%f/%e	printf("%g %g\n", 0.000000123, 0.123); 输出:1.23e-07 0.123
G	double	根据数值的长度，选择以最短的方式输出，%F/%E	printf("%G %G\n", 0.000000123, 0.123); 输出:1.23E-07 0.123
s	char *	字符串，输出字符串中的字符直至终止字符'\0'	printf("%s\n", "Hello World"); 输出:Hello World
p	void *	输出十六进制表示的指针	printf("%p\n", "Hello World"); 输出:0x400624
c	char	字符型，可以把输入的数字按照 ASCII 码相应转换为对应的字符输出	printf("%c\n", 64); 输出:A

表 4.8.1 转换说明中的 type 字段介绍

㈡、flags

flags 规定输出样式，%后面可以跟 0 个或多个以下标志：

字符	名称	作用
#	井号	type 等于 o 时，输出字符串增加前缀 0。 type 等于 x 或 X 时，输出字符串增加前缀 0x 或 0X。 type 等于 a、A、e、E、f、F、g 和 G 其中之一时，在默认情况下，只有输出小数部分时才会输出小数点，如果使用.0 控制不输出小数部分，那么小数点是不会输出的，然而在使用了标志#的情况下，输出结果始终包含小数点；type 等于 g 和 G 时，保留尾部的 0。
0	数字 0	当 type 不等于 c 或 s 时（也就是输出数字时，包括浮点数和整数），在输出字符串前面补 0，直到占满指定的最小输出宽度（位数）。譬如输出正数 100，指定的最小输出宽度是 5，那么最终就会输出 00100。如果没有指定标志 0，则默认使用空格占满指定的最小输出宽度。

-	减号	输出字符串默认情况下是右对齐的，不足最小输出宽度时在左边填空格或 0；使用了-标志，则会变成左对齐，然后在右边填空格，如果同时指定了标志 0 和标志-，则标志-会覆盖标志 0。
' '	空格	输出正数时在前面加一个空格，输出负数时，前面加一个负号-。
+	加号	默认情况下，只有输出负数时，才会输出负号-；正数前面是没有正号+的；而使用了标志+后，输出的数字前面都带有符号（正数为+、负数为-）；如果同时指定了标志+和标志' '（空格），则标志+会覆盖标志空格。

表 4.8.2 转换说明中的 flags 字段介绍

㈢、width

最小的输出宽度，用十进制数来表示输出的最小位数，若实际的输出位数大于指定的输出的最小位数，则以实际的位数进行输出，若实际的位数小于指定输出的最小位数，则可按照指定的 flags 标志补 0 或补空格。

width 的可能取值如下：

width	描述	示例
数字	十进制正数	printf("%06d", 1000); 输出: 001000
*	星号，不显示指出最小输出宽度，而是以星号代替，会在参数列表中指定	printf("%0*d", 6, 1000); 输出: 001000

表 4.8.3 转换说明中的 width 字段介绍

㈣、precision 精度

精度字段以点号" . "开头，后跟一个十进制正数，可取值如下：

.precision	描述
数字	十进制正数 ①对于整形(type 等于 d、i、o、u、x 和 X)，precision 表示输出的最小的数字个数，不足补前导零，超过不截断。这里要注意：是数字的个数、与 width 字段是有区别的，width 指的是整个输出字符串的最小位数（最小宽度），并不是数字的最小宽度，譬如： printf("%8.5d\n", 100); 输出: 00100（前面有 3 个空格）； 在这个例子中，width 字段为 8，表示需要整个字符串的输出长度为 8 个字符，.5 则表示数字部分位数最少为 5 个，不足 5 个则在前面补 0，所以 100 需要在前面补两个 0 才能满足这个要求；满足这个要求之后，接着需要使整个字符串长度为 8 个字符，那么只需要在前面补 3 个空格即可（这里是左对齐的情况）！ 会忽略 flags 字段的标志 0，意味着在这种情况下，指定标志 0 和不指定标志 0 都是一样的效果。 ②对于浮点型(type 等于 a、A、e、E、f、F)，precision 表示小数点后数字的个数，也就是浮点数精度；默认为六位，不足补后置 0，超过则截断。譬如： printf("%.8f\n", 520.1314); 输出:520.13140000 ③type 等于 g、G 时，表示最大有效位数； ④对于字符串(type 等于 s)，precision 表示输出字符串中最大可输出的字符数，不足正常输出，超过则截断。譬如： printf("%.5s\n", "hello world"); 输出:hello；超过 5 个字符的部分被丢弃！

*	以星号代替十进制数字，类似于 width 字段中的*，表示在参数列表中指定；譬如： printf("%.*s\n", 5, "hello world"); 输出:hello

表 4.8.4 转换说明中的 precision 字段介绍

㈤、length 长度修饰符

长度修饰符指明待转换数据的长度，因为 type 字段指定的的类型只有 int、unsigned int 以及 double 等几种数据类型，但是 C 语言内置的数据类型不止这几种，譬如有 16bit 的 short、unsigned short，8bit 的 char、unsigned char，也有 64bit 的 long long 等，为了能够区别不同长度的数据类型，于是乎，长度修饰符（length）应运而生，成为转换说明的一部分。

length 长度修饰符也是使用字符（字母字符）来表示，结合 type 字段以确定不同长度的数据类型，如下所示：

	type					
length	d、i	u、o、x、X	f、F、e、E、g、G	c	s	p
(none)	int	unsigned int	double	char	char *	void *
hh	signed char	unsigned char				
h	short int	unsigned short int				
l	long int	unsigned long int		wint_t	wchar_t	
ll	long long int	unsigned long long int				
L			long double			
j	intmax_t	uintmax_t				
z	size_t	ssize_t				
t	ptrdiff_t	ptrdiff_t				

表 4.8.5 length 长度修饰符说明

譬如：

```
printf("%hd\n", 12345);      //将数据以 short int 类型进行转换
printf("%ld\n", 12345);      //将数据以 long int 类型进行转换
printf("%lld\n", 12345);     //将数据以 long long int 类型进行转换
```

关于格式控制字符串 format 就给大家介绍完了，这种东西不用去记，需要时查询即可！需要说明的是，转换说明的描述信息需要和与之相对应的参数对应的数据类型要进行匹配，如果不匹配通常会编译报错或者警告！

示例代码

前面为了说明格式控制字符串 format 的输出效果，我们使用了 printf()函数进行演示，其它格式化输出函数也是一样，接下来我们编写一个简单的测试程序，对上面学习的内容进行练习。

示例代码 4.8.1 格式化输出函数使用练习

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

int main(void)
{
    char buf[50] = {0};
```

```
    printf("%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");
    fprintf(stdout, "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");
    dprintf(STDOUT_FILENO, "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");

    sprintf(buf, "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");
    printf("%s", buf);

    memset(buf, 0x00, sizeof(buf));
    snprintf(buf, sizeof(buf), "%d (%s) %d (%s)\n", 520, "我爱你", 1314, "一生一世");
    printf("%s", buf);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
520 (我爱你) 1314 (一生一世)
520 (我爱你) 1314 (一生一世)
520 (我爱你) 1314 (一生一世)
520 (我爱你) 1314 (一生一世)
520 (我爱你) 1314 (一生一世)
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.8.1 运行结果

关于格式化输出这几个函数其实用法上比较简单，主要是需要掌握格式控制字符串 format 参数的写法，对后续参数列表中不同类型的数据搭配不同的格式控制字符，以实现转换输出、并且控制输出样式。

本小节所学内容，大家可以多多练习！

4.8.2 格式化输入

C 库函数提供了 3 个格式化输入函数，包括：scanf()、fscanf()、sscanf()，其函数定义如下所示：

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

可以看到，这 3 个格式化输入函数也是可变参函数，它们都有一个共同的参数 format，同样也称为格式控制字符串，用于指定输入数据如何进行格式转换，与格式化输出函数中的 format 参数格式相似，但也有所不同。

每个函数除了固定参数之外，还可携带 0 个或多个可变参数。

scanf()函数可将用户输入（标准输入）的数据进行格式化转换；fscanf()函数从 FILE 指针指定文件中读取数据，并将数据进行格式化转换；sscanf()函数从参数 str 所指向的字符串中读取数据，并将数据进行格式化转换。

scanf()函数

相对于 printf 函数，scanf 函数就简单得多。scanf()函数的功能与 printf()函数正好相反，执行格式化输入功能；即 scanf()函数将用户输入（标准输入）的数据进行格式化转换并进行存储，它从格式化控制字符串 format 参数的最左端开始，每遇到一个转换说明便将其与下一个输入数据进行“匹配”，如果二者匹配则继续，否则结束对后面输入的处理。而每遇到一个转换说明，便按该转换说明所描述的格式对其后的输入数据进行转换，然后将转换得到的数据存储于与其对应的输入地址中。以此类推，直到对整个输入数据的处理结束为止。

从函数原型可以看出，scanf()函数也是一个“可变参数函数”，除第一个参数 format 之外，scanf()函数还可以有若干个输入地址（指针），这些指针指向对应的缓冲区，用于存储格式化转换后的数据；且对于每一个输入地址，在格式控制字符串 format 参数中都必须有一个转换说明与之一一对应。即从 format 字符串的左端第 1 个转换说明对应第 1 个输入地址，第 2 个格式说明符对应第 2 个输入地址，第 3 个格式说明符对应第 3 个输入地址，以此类推。譬如：

```
int a, b, c;
scanf("%d %d %d", &a, &b, &c);
```

当程序中调用 scanf()的时候，终端会被阻塞，等待用户输入数据，此时我们可以通过键盘输入一些字符，譬如数字、字母或者其它字符，输入完成按回车即可！接着来 scanf()函数就会对用户输入的数据进行格式转换处理。

函数调用成功后，将返回成功匹配和分配的输入项的数量；如果较早匹配失败，则该数目可能小于所提供的数目，甚至为零。发生错误则返回负值。

fscanf()函数

fscanf()函数从指定文件中读取数据，作为格式转换的输入数据，文件通过 FILE 指针指定，所以它有两个固定参数，FILE 指针和格式控制字符串 format。譬如从标准输入文件中读取数据进行格式化转换：

```
int a, b, c;
fscanf(stdin, "%d %d %d", &a, &b, &c);
```

此时它的作用与 scanf()就是相同的，因为标准输入文件的数据就是用户输入的数据，譬如通过键盘输入的数据。

函数调用成功后，将返回成功匹配和分配的输入项的数量；如果较早匹配失败，则该数目可能小于所提供的数目，甚至为零。发生错误则返回负值。

sscanf()函数

sscanf()将从参数 str 所指向的字符串缓冲区中读取数据，作为格式转换的输入数据，所以它也有两个固定参数，字符串 str 和格式控制字符串 format，譬如：

```
char *str = "5454 hello";
char buf[10];
int a;


sscanf(str, "%d %s", &a, buf);
```

函数调用成功后，将返回成功匹配和分配的输入项的数量；如果较早匹配失败，则该数目可能小于所提供的数目，甚至为零。发生错误则返回负值。

格式控制字符串 format

本小节的重点依然是这个 format 参数的格式，与格式化输出函数中的 format 参数格式、写法上比较相似，但也有一些区别。format 字符串包含一个或多个转换说明，每一个转换说明都是以百分号"%"或者"%n$"开头（n 是一个十进制数字），关于"%n$"这种开头的转换说明就不介绍了，实际上用的不多。

以%百分号开头的转换说明一般格式如下：

```
%[*][width][length]type
%[m][width][length]type
```

%后面可选择性添加星号*或字母 m，如果添加了星号*，格式化输入函数会按照转换说明的指示读取输入，但是丢弃输入，意味着不需要对转换后的结果进行存储，所以也就不需要提供相应的指针参数。

如果添加了 m，它只能与%s、%c 以及%[一起使用，调用者无需分配相应的缓冲区来保存格式转换后的数据，原因在于添加了 m，这些格式化输入函数内部会自动分配足够大小的缓冲区，并将缓冲区的地址值通过与该格式转换相对应的指针参数返回出来，该指针参数应该是指向 char *变量的指针。随后，当不再需要此缓冲区时，调用者应调用 free()函数来释放此缓冲区。

譬如：

```
char *buf;

scanf("%ms", &buf);
......
free(buf);
```

介绍了星号*和字母 m 之后，再来看看转换说明的格式，中括号[ ]表示的部分是可选的，所以可知，与格式化输出函数中的 format 参数一样，只有 type 字段是必须的。

- **width**：最大字符宽度；
- **length**：长度修饰符，与格式化输出函数的 format 参数中的 length 字段意义相同。
- **type**：指定输入数据的类型。

我们先来看看 type 字段。

㈠type（类型）

此 type 字段与格式化输出函数中的 format 参数的 type 字段是同样的意义，用于指定输入数据的类型，如下所示：

字符	对应的数据类型	含义	示例
d	int	匹配一个有符号十进制整数	int a; scanf("%d", &a); 用户输入: 100
i	int	匹配一个有符号整数，既可以是十进制表示、也可以是八进制或十六进制方式表示，譬如整数以 0x 或 0X 开头，则认为是 16 进制，以 0 开头则认为是八进制，否则认为是十进制。	int a; scanf("%i", &a); 用户输入: 0x100
o	unsigned int	匹配一个无符号八进制整数	unsigned int a; scanf("%o", &a); 用户输入: 0100
u	unsigned int	匹配一个无符号十进制整数	unsigned int a; scanf("%u", &a); 用户输入: 100
x	unsigned int	匹配一个无符号十六进制整数，数字以 0x 开头	unsigned int a; scanf("%x", &a); 用户输入: 0x100

X	unsigned int	匹配一个无符号十六进制整数，数字以 0X 开头	unsigned int a; scanf("%x", &a); 用户输入: 0X100
f	float	匹配一个带符号的浮点数	float a; scanf("%f", &a); 用户输入: 10.123
e	float	等效于 f	
E	float	等效于 f	
g	float	等效于 f	
a	float	等效于 f	
s	char *	匹配字符串，不匹配空白字符（包括空格、制表符、换行符），空白字符作为字符串的分隔符，所以就是匹配一系列非空白字符。所以存放字符串的缓冲区必须足够大，会自动添加终止字符"\0"	char buf[100]; scanf("%s", buf); 用户输入: HelloWorld
c	char	匹配一个字符	char c; scanf("%c", &c); 用户输入: A
p	void *	匹配一个指针值	void *a; scanf("%p", &a); 用户输入: 123456
[	char *	匹配一组字符序列集合，以字符串形式存储。譬如%[a-z]匹配 a 到 z 之间的所有字符（包括起始字符 a 和结束字符 z），譬如%[0-9]则表示匹配数字 0 到 9 之间所有数字字符；减号-是一个连字符，放在两个字符（一个起始字符和一个结束字符）之间，本身并不参与匹配，如果需要匹配减号-，则可将其放在中括号[ ]旁边，譬如%[-a-z]，表示匹配减号、以及数字 0 到 9 这些字符；如果要排除匹配这些字符，可以使用排除符^，将其放在最前面，譬如%[^a-z-]，表示匹配除 a 到 z 这些字符以及-之外的字符。	char buf[100]; scanf("%[a-z0-9]", buf); 匹配字母 a 到 z 以及数字 0 到 9 这些字符。 用户输入: 123abc

表 4.8.6 type 类型描述

㈡、width 最大字符宽度

是一个十进制表示的整数，用于指定最大字符宽度，当达到此最大值或发现不匹配的字符时（以先发生者为准），字符的读取将停止。大多数 type 类型会丢弃初始的空白字符，并且这些丢弃的字符不会计入最大字符宽度。对于字符串转换来说，scanf()会在字符串末尾自动添加终止符"\0"，最大字符宽度中不包括此终止符。

譬如调用 scanf()函数如下：

```
scanf("%4s", buf);          //匹配字符串，字符串长度不超过 4 个字符
```

用户输入 abcdefg，按回车，那么只能将 adcd 作为一个字符串存储在 buf 数组中。

㈢length 长度修饰符

与格式化输出函数的格式控制字符串 format 中的 length 字段意义相同，用于对 type 字段进行修饰，扩展识别更多不同长度的数据类型。如下所示：

	type				
length	d、i	u、o、x、X	e、f、g	c	s
(none)	int	unsigned int	float	char	char *
h	short int	unsigned short int			
hh	signed char	unsigned char			
j	intmax_t	uintmax_t			
l	long int	unsigned long int	double	wchar_t	wchar_t *
L	long long int	unsigned long long int	long double		

表 4.8.7 length 长度修饰符

譬如：

```
scanf("%hd", var);      //匹配 short int 类型数据
scanf("%hhd", var);     //匹配 signed char 类型数据
scanf("%ld", var);      //匹配 long int 类型数据
scanf("%f", var);       //匹配 float 类型数据
scanf("%lf", var);      //匹配 double 类型数据
scanf("%Lf", var);      //匹配 long double 类型数据
```

关于格式化输入函数的 format 参数就介绍到到这里了，接下来编写一个简单地示例进行测试。

使用示例

示例代码 4.8.2 scanf()函数使用示例

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

int main(void)
{
    int a;
    float b;
    char *str;

    printf("请输入一个整数:\n");
    scanf("%d", &a);
    printf("你输入的整数为: %d\n", a);

    printf("请输入一个浮点数:\n");
    scanf("%f", &b);
    printf("你输入的浮点数为: %f\n", b);

    printf("请输入一个字符串:\n");
    scanf("%ms", &str);
    printf("你输入的字符串为: %s\n", str);
```

```
    free(str);          //释放字符串占用的内存空间


    exit(0);
}
```

当程序中调用 scanf()之后，终端就会被阻塞、等待用户输入数据，当我们输入完成之后，按回车即可！第三个 scanf()函数调用中，使用%m，所以我们不需要提供存放字符串的缓冲区，scanf()函数内部会分配缓冲区，并将缓冲区地址存放在 str 这个我们给定的 char 指针变量中。使用完之后记得调用 free()释放内存即可。

编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入一个整数：
123456
你输入的整数为：123456
请输入一个浮点数：
10.11
你输入的浮点数为：10.110000
请输入一个字符串：
HelloWorld
你输入的字符串为：HelloWorld
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.8.2 测试结果

4.8.3 小结

本小节（4.8）对标准 I/O 中的格式化 I/O 做了比较详细的介绍，相信大家对此都比较熟悉了，当然，重要的是需要灵活使用它们。关于格式化 I/O 还存在一些比较细节的问题需要我们注意，主要是围绕缓冲的问题，关于缓冲的问题将会在下一小节向大家介绍！

4.9 I/O 缓冲

出于速度和效率的考虑，系统 I/O 调用（即文件 I/O，open、read、write 等）和标准 C 语言库 I/O 函数（即标准 I/O 函数）在操作磁盘文件时会对数据进行缓冲，本小节将讨论文件 I/O 和标准 I/O 这两种 I/O 方式的数据缓冲问题，并讨论其对应用程序性能的影响。

除此之外，本小节还讨论了屏蔽或影响缓冲的一些技术手段，以及直接 I/O 技术—绕过内核缓冲直接访问磁盘硬件。

4.9.1 文件 I/O 的内核缓冲

read()和 write()系统调用在进行文件读写操作的时候并不会直接访问磁盘设备，而是仅仅在用户空间缓冲区和内核缓冲区（kernel buffer cache）之间复制数据。譬如调用 write()函数将 5 个字节数据从用户空间内存拷贝到内核空间的缓冲区中：

```
write(fd, "Hello", 5);          //写入 5 个字节数据
```

调用 write()后仅仅只是将这 5 个字节数据拷贝到了内核空间的缓冲区中，拷贝完成之后函数就返回了，在后面的某个时刻，内核会将其缓冲区中的数据写入（刷新）到磁盘设备中，所以由此可知，系统调用 write()与磁盘操作并不是同步的，write()函数并不会等待数据真正写入到磁盘之后再返回。如果在此期间，其它进

程调用 read()函数读取该文件的这几个字节数据，那么内核将自动从缓冲区中读取这几个字节数据返回给应用程序。

与此同理，对于读文件而言亦是如此，内核会从磁盘设备中读取文件的数据并存储到内核的缓冲区中，当调用 read()函数读取数据时，read()调用将从内核缓冲区中读取数据，直至把缓冲区中的数据读完，这时，内核会将文件的下一段内容读入到内核缓冲区中进行缓存。

我们把这个内核缓冲区就称为文件 I/O 的内核缓冲。这样的设计，目的是为了提高文件 I/O 的速度和效率，使得系统调用 read()、write()的操作更为快速，不需要等待磁盘操作（将数据写入到磁盘或从磁盘读取出数据），磁盘操作通常是比较缓慢的。同时这一设计也更为高效，减少了内核操作磁盘的次数，譬如线程 1 调用 write()向文件写入数据"abcd"，线程 2 也调用 write()向文件写入数据"1234"，这样的话，数据"abcd"和"1234"都被缓存在了内核的缓冲区中，在稍后内核会将它们一起写入到磁盘中，只发起一次磁盘操作请求；加入没有内核缓冲区，那么每一次调用 write()，内核就会执行一次磁盘操作。

前面提到，当调用 write()之后，内核稍后会将数据写入到磁盘设备中，具体是什么时间点写入到磁盘，这个其实是不确定的，由内核根据相应的存储算法自动判断。

通过前面的介绍可知，文件 I/O 的内核缓冲区自然是越大越好，Linux 内核本身对内核缓冲区的大小没有固定上限。内核会分配尽可能多的内核来作为文件 I/O 的内核缓冲区，但受限于物理内存的总量，如果系统可用的物理内存越多，那自然对应的内核缓冲区也就越大，操作越大的文件也要依赖于更大空间的内核缓冲。

4.9.2 刷新文件 I/O 的内核缓冲区

强制将文件 I/O 内核缓冲区中缓存的数据写入（刷新）到磁盘设备中，对于某些应用程序来说，可能是很有必要的，例如，应用程序在进行某操作之前，必须要确保前面步骤调用 write()写入到文件的数据已经真正写入到了磁盘中，诸如一些数据库的日志进程。

联系到一个实际的使用场景，当我们在 Ubuntu 系统下拷贝文件到 U 盘时，文件拷贝完成之后，通常在拔掉 U 盘之前，需要执行 sync 命令进行同步操作，这个同步操作其实就是将文件 I/O 内核缓冲区中的数据更新到 U 盘硬件设备，所以如果在没有执行 sync 命令时拔掉 U 盘，很可能就会导致拷贝到 U 盘中的文件遭到破坏！

控制文件 I/O 内核缓冲的系统调用

Linux 中提供了一些系统调用可用于控制文件 I/O 内核缓冲，包括系统调用 sync()、syncfs()、fsync()以及 fdatasync()。

㈠、fsync()函数

系统调用 fsync()将参数 fd 所指文件的内容数据和元数据写入磁盘，只有在对磁盘设备的写入操作完成之后，fsync()函数才会返回，其函数原型如下所示：

```
#include <unistd.h>

int fsync(int fd);
```

参数 fd 表示文件描述符，函数调用成功将返回 0，失败返回-1 并设置 errno 以指示错误原因。

前面提到了元数据这个概念，元数据并不是文件内容本身的数据，而是一些用于记录文件属性相关的数据信息，譬如文件大小、时间戳、权限等等信息，这里统称为文件的元数据，这些信息也是存储在磁盘设备中的，在 3.1 小节中介绍过。

使用示例

示例代码 4.9.1 实现了一个文件拷贝操作，将源文件（当前目录下的 rfile 文件）的内容拷贝到目标文件中（当前目录下的 wfile 文件）。

示例代码 4.9.1 fsync()函数使用示例

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

#define BUF_SIZE        4096
#define READ_FILE       "./rfile"
#define WRITE_FILE      "./wfile"

static char buf[BUF_SIZE];

int main(void)
{
    int rfd, wfd;
    size_t size;

    /* 打开源文件 */
    rfd = open(READ_FILE, O_RDONLY);
    if (0 > rfd) {
        perror("open error");
        exit(-1);
    }

    /* 打开目标文件 */
    wfd = open(WRITE_FILE, O_WRONLY | O_CREAT | O_TRUNC, 0664);
    if (0 > wfd) {
        perror("open error");
        exit(-1);
    }

    /* 拷贝数据 */
    while(0 < (size = read(rfd, buf, BUF_SIZE)))
        write(wfd, buf, size);

    /* 对目标文件执行 fsync 同步 */
    fsync(wfd);

    /* 关闭文件退出程序 */
```

```
    close(rfd);
    close(wfd);
    exit(0);
}
```

代码没什么好说的，主要就是拷贝完成之后调用 fsync()函数，对目标文件的数据进行了同步操作，整个操作完成之后 close 关闭源文件和目标文件、退出程序。

㈡、fdatasync()函数

系统调用 fdatasync()与 fsync()类似，不同之处在于 fdatasync()仅将参数 fd 所指文件的内容数据写入磁盘，并不包括文件的元数据；同样，只有在对磁盘设备的写入操作完成之后，fdatasync()函数才会返回，其函数原型如下所示：

```
#include <unistd.h>

int fdatasync(int fd);
```

㈢、sync()函数

系统调用 sync()会将所有文件 I/O 内核缓冲区中的文件内容数据和元数据全部更新到磁盘设备中，该函数没有参数、也无返回值，意味着它不是对某一个指定的文件进行数据更新，而是刷新所有文件 I/O 内核缓冲区。其函数原型如下所示：

```
#include <unistd.h>

void sync(void);
```

在 Linux 实现中，调用 sync()函数仅在所有数据已经写入到磁盘设备之后才会返回；然后在其它系统中，sync()实现只是简单调度一下 I/O 传递，在动作未完成之后即可返回。

控制文件 I/O 内核缓冲的标志

调用 open()函数时指定一些标志也可以影响到文件 I/O 内核缓冲，譬如 O_DSYNC 标志和 O_SYNC 标志，这些标志在 2.3 小节并未向大家介绍过，联系本小节所学内容，接下来向大家简单地介绍下。

㈠、O_DSYNC 标志

在调用 open()函数时，指定 O_DSYNC 标志，其效果类似于在每个 write()调用之后调用 fdatasync()函数进行数据同步。譬如：

```
fd = open(filepath, O_WRONLY | O_DSYNC);
```

㈡、O_SYNC 标志

在调用 open()函数时，指定 O_SYNC 标志，使得每个 write()调用都会自动将文件内容数据和元数据刷新到磁盘设备中，其效果类似于在每个 write()调用之后调用 fsync()函数进行数据同步，譬如：

```
fd = open(filepath, O_WRONLY | O_SYNC);
```

对性能的影响

在程序中频繁调用 fsync()、fdatasync()、sync()（或者调用 open 时指定 O_DSYNC 或 O_SYNC 标志）对性能的影响极大，大部分的应用程序是没有这种需求的，所以在大部分应用程序当中基本不会使用到。

4.9.3 直接 I/O：绕过内核缓冲

从 Linux 内核 2.4 版本开始，Linux 允许应用程序在执行文件 I/O 操作时绕过内核缓冲区，从用户空间直接将数据传递到文件或磁盘设备，把这种操作也称为直接 I/O（direct I/O）或裸 I/O（raw I/O）。

在有些情况下，这种操作通常是很有必要的，例如，某应用程序的作用是测试磁盘设备的读写速率，那么在这种应用需要下，我们就需要保证 read/write 操作是直接访问磁盘设备，而不经过内核缓冲，如果不能得到这样的保证，必然会导致测试结果出现比较大的误差。

然后，对于大多数应用程序而言，使用直接 I/O 可能会大大降低性能，这是因为为了提高 I/O 性能，内核针对文件 I/O 内核缓冲区做了不少的优化，譬如包括按顺序预读取、在成簇磁盘块上执行 I/O、允许访问同一文件的多个进程共享高速缓存的缓冲区。如果应用程序使用直接 I/O 方式，将无法享受到这些优化措施所带来的性能上的提升，直接 I/O 只在一些特定的需求场合，譬如磁盘速率测试工具、数据库系统等。

我们可针对某一文件或块设备执行直接 I/O，要做到这一点，需要在调用 open()函数打开文件时，指定 O_DIRECT 标志，该标志至 Linux 内核 2.4.10 版本开始生效，譬如：

```
fd = open(filepath, O_WRONLY | O_DIRECT);
```

直接 I/O 的对齐限制

因为直接 I/O 涉及到对磁盘设备的直接访问，所以在执行直接 I/O 时，必须要遵守以下三个对齐限制要求：

- 应用程序中用于存放数据的缓冲区，其内存起始地址必须以块大小的整数倍进行对齐；
- 写文件时，文件的位置偏移量必须是块大小的整数倍；
- 写入到文件的数据大小必须是块大小的整数倍。

如果不满足以上任何一个要求，调用 write()均为以错误返回 Invalid argument。以上所说的块大小指的是磁盘设备的物理块大小（block size），常见的块大小包括 512 字节、1024 字节、2048 以及 4096 字节，那我们如何确定磁盘分区的块大小呢？可以使用 tune2fs 命令进行查看，如下所示：

```
tune2fs -l /dev/sda1 | grep "Block size"
```

-l 后面指定了需要查看的磁盘分区，可以使用 df -h 命令查看 Ubuntu 系统的根文件系统所挂载的磁盘分区：

```
dt@dt-virtual-machine:~$ df -h
文件系统        容量  已用  可用 已用% 挂载点
udev            1.9G     0  1.9G    0% /dev
tmpfs           393M   12M  382M    3% /run
/dev/sda1        75G  7.8G   64G   11% /
tmpfs           2.0G   33M  1.9G    2% /dev/shm
tmpfs           5.0M  4.0K  5.0M    1% /run/lock
tmpfs           2.0G     0  2.0G    0% /sys/fs/cgroup
tmpfs           393M   60K  393M    1% /run/user/1000
dt@dt-virtual-machine:~$
```

图 4.9.1 查看根文件系统挂载的磁盘分区

通过上图可知，Ubuntu 系统的根文件系统挂载在/dev/sda1 磁盘分区下，接着下使用 tune2fs 命令查看该分区的块大小：

```
dt@dt-virtual-machine:~$ sudo tune2fs -l /dev/sda1 | grep "Block size"
[sudo] dt 的密码：
Block size:               4096
dt@dt-virtual-machine:~$
```

图 4.9.2 磁盘块大小

从上图可知/dev/sda1 磁盘分区的块大小为 4096 个字节。

直接 I/O 测试与普通 I/O 对比测试

接下来编写一个使用直接 I/O 方式写文件的测试程序和一个使用普通 I/O 方式写文件的测试程序，进行对比。

示例代码 4.9.2 演示了以直接 I/O 方式写文件的操作，首先我们需要在程序开头处定义一个宏定义_GNU_SOURCE，原因在于后面 open()函数需要指定 O_DIRECT 标志，这个宏需要我们在程序中定义了 O_DIRECT 宏之后才能使用，否则编译程序就会报错提示：O_DIRECT 未定义。

Tips：_GNU_SOURCE 宏可用于开启/禁用 Linux 系统调用和 glibc 库函数的一些功能、特性，要打开这些特性，需要在应用程序中定义该宏，定义该宏之后意味着用户应用程序打开了所有的特性；默认情况下，_GNU_SOURCE 宏并没有被定义，所以当使用到它控制的一些特性时，应用程序编译将会报错！定义该宏的方式有两种：

- 直接在源文件中定义：#define _GNU_SOURCE
- gcc 编译时使用-D 选项定义_GNU_SOURCE 宏：

```
gcc -D_GNU_SOURCE -o testApp testApp.c
```

gcc 的-D 选项可用于定义一个宏，并且该宏定义在整个源码工程中都是生效的，是一个全局宏定义。使用以上哪种方式都可以。

示例代码 4.9.2 直接 I/O 示例程序

```
/** 使用宏定义 O_DIRECT 需要在程序中定义宏_GNU_SOURCE
 ** 不然提示 O_DIRECT 找不到 **/
#define _GNU_SOURCE

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

/** 定义一个用于存放数据的 buf，起始地址以 4096 字节进行对其 **/
static char buf[8192] __attribute((aligned (4096)));

int main(void)
{
    int fd;
    int count;

    /* 打开文件 */
    fd = open("./test_file",
              O_WRONLY | O_CREAT | O_TRUNC | O_DIRECT,
              0664);
    if (0 > fd) {
        perror("open error");
        exit(-1);
    }

    /* 写文件 */
    count = 10000;
```

```
    while(count--) {
        if (4096 != write(fd, buf, 4096)) {
            perror("write error");
            exit(-1);
        }
    }

    /* 关闭文件退出程序 */
    close(fd);
    exit(0);
}
```

前面提到过，使用直接 I/O 方式需要满足 3 个对齐要求，程序中定义了一个 static 静态数组 buf，将其作为数据存放的缓冲区，在变量定义后加了__attribute((aligned (4096)))修饰，使其起始地址以 4096 字节进行对其。

Tips：__attribute 是 gcc 支持的一种机制（也可以写成__attribute__），可用于设置函数属性、变量属性以及类型属性等，对此不了解的读者请自行查找资料学习，本书不会对此进行介绍！

程序中调用 open()函数是指定了 O_DIRECT 标志，使用直接 I/O，最后通过 while 循环，将数据写入文件中，循环 10000 次，每次写入 4096 个字节数据，也就是总共写入 4096*10000 个字节（约等于 40MB）。首次调用 write()时其文件读写位置偏移量为 0，之后均以 4096 字节进行递增，所以满足直接 I/O 方式的位置偏移量必须是块大小的整数倍这个要求；每次写入大小均是 4096 字节，所以满足了数据大小必须是块大小的整数倍这个要求。

接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp

real    0m2.763s
user    0m0.001s
sys     0m1.132s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp

real    0m2.705s
user    0m0.001s
sys     0m1.120s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.9.3 直接 I/O 测试结果

通过 time 命令测试可知，每次执行程序需要花费 2.7 秒左右的时间，使用直接 I/O 方式向文件写入约 40MB 数据大小。

Tips：对于直接 I/O 方式的 3 个对齐限制，大家可以自行进行验证，譬如修改上述示例代码使之不满足 3 个对齐条件种的任何一个，然后编译程序进行测试，会发生 write()函数会报错，均是“Invalid argument”错误。

对示例代码 4.9.2 进行修改，使其变成普通 I/O 方式，其它功能相同，最终修改后的示例代码如下所示：

示例代码 4.9.3 普通 I/O 方式

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

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

static char buf[8192];

int main(void)
{
    int fd;
    int count;

    /* 打开文件 */
    fd = open("./test_file", O_WRONLY | O_CREAT | O_TRUNC, 0664);
    if (0 > fd) {
        perror("open error");
        exit(-1);
    }

    /* 写文件 */
    count = 10000;
    while(count--) {//循环 10000 次，每次写入 4096 个字节数据
        if (4096 != write(fd, buf, 4096)) {
            perror("write error");
            exit(-1);
        }
    }

    /* 关闭文件退出程序 */
    close(fd);
    exit(0);
}
```

再次进行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp

real    0m0.136s
user    0m0.000s
sys     0m0.135s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ time ./testApp

real    0m0.146s
user    0m0.005s
sys     0m0.141s
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.9.4 普通 I/O 测试结果

使用 time 命令得到的程序运行时间大约是 0.13~0.14 秒左右，相比直接 I/O 方式的 2.7 秒，时间上提升了 20 倍左右（测试大小不同、每次写入的大小不同，均会导致时间上的差别），原因在于直接 I/O 方式每次 write()调用均是直接对磁盘发起了写操作，而普通方式只是将用户空间下的数据拷贝到了文件 I/O 内核缓冲区中，并没直接操作硬件，所以消耗的时间短，硬件操作占用的时间远比内存复制占用的时间大得多

直接 I/O 方式效率、性能比较低，绝大部分应用程序不会使用直接 I/O 方式对文件进行 I/O 操作，通常只在一些特殊的应用场合下才可能会使用，那我们可以使用直接 I/O 方式来测试磁盘设备的读写速率，这种测试方式相比普通 I/O 方式就会更加准确。

4.9.4 stdio 缓冲

介绍完文件 I/O 的内核缓冲后，接下来我们聊一聊标准 I/O 的 stdio 缓冲。

标准 I/O（fopen、fread、fwrite、fclose、fseek 等）是 C 语言标准库函数，而文件 I/O（open、read、write、close、lseek 等）是系统调用，虽然标准 I/O 是在文件 I/O 基础上进行封装而实现（譬如 fopen 内部实际上调用了 open、fread 内部调用了 read 等），但在效率、性能上标准 I/O 要优于文件 I/O，其原因在于标准 I/O 实现维护了自己的缓冲区，我们把这个缓冲区称为 stdio 缓冲区，接下来我们聊一聊标准 I/O 的 stdio 缓冲。

前面提到了文件 I/O 内核缓冲，这是由内核维护的缓冲区，而标准 I/O 所维护的 stdio 缓冲是用户空间的缓冲区，当应用程序中通过标准 I/O 操作磁盘文件时，为了减少调用系统调用的次数，标准 I/O 函数会将用户写入或读取文件的数据缓存在 stdio 缓冲区，然后再一次性将 stdio 缓冲区中缓存的数据通过调用系统调用 I/O（文件 I/O）写入到文件 I/O 内核缓冲区或者拷贝到应用程序的 buf 中。

通过这样的优化操作，当操作磁盘文件时，在用户空间缓存大块数据以减少调用系统调用的次数，使得效率、性能得到优化。使用标准 I/O 可以使编程者免于自行处理对数据的缓冲，无论是调用 write()写入数据、还是调用 read()读取数据。

对 stdio 缓冲进行设置

C 语言提供了一些库函数可用于对标准 I/O 的 stdio 缓冲区进行相关的一些设置，包括 setbuf()、setbuffer()以及 setvbuf()。

㈠、setvbuf()函数

调用 setvbuf()库函数可以对文件的 stdio 缓冲区进行设置，譬如缓冲区的缓冲模式、缓冲区的大小、起始地址等。其函数原型如下所示：

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

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

stream：FILE 指针，用于指定对应的文件，每一个文件都可以设置它对应的 stdio 缓冲区。

buf：如果参数 buf 不为 NULL，那么 buf 指向 size 大小的内存区域将作为该文件的 stdio 缓冲区，因为 stdio 库会使用 buf 指向的缓冲区，所以应该以动态（分配在堆内存，譬如 malloc，在 7.6 小节介绍）或静态的方式在堆中为该缓冲区分配一块空间，而不是分配在栈上的函数内的自动变量（局部变量）。如果 buf 等于 NULL，那么 stdio 库会自动分配一块空间作为该文件的 stdio 缓冲区（除非参数 mode 配置为非缓冲模式）。

mode：参数 mode 用于指定缓冲区的缓冲类型，可取值如下：

- **_IONBF**：不对 I/O 进行缓冲（无缓冲）。意味着每个标准 I/O 函数将立即调用 write()或者 read()，并且忽略 buf 和 size 参数，可以分别指定两个参数为 NULL 和 0。标准错误 stderr 默认属于这一种类型，从而保证错误信息能够立即输出。
- **_IOLBF**：采用行缓冲 I/O。在这种情况下，当在输入或输出中遇到换行符"\n"时，标准 I/O 才会执行文件 I/O 操作。对于输出流，在输出一个换行符前将数据缓存（除非缓冲区已经被填满），当输出换行符时，再将这一行数据通过文件 I/O write()函数刷入到内核缓冲区中；对于输入流，每次读取一行数据。对于终端设备默认采用的就是行缓冲模式，譬如标准输入和标准输出。
- **_IOFBF**：采用全缓冲 I/O。在这种情况下，在填满 stdio 缓冲区后才进行文件 I/O 操作（read、write）。对于输出流，当 fwrite 写入文件的数据填满缓冲区时，才调用 write()将 stdio 缓冲区中的数据刷入内核缓冲区；对于输入流，每次读取 stdio 缓冲区大小个字节数据。默认普通磁盘上的常规文件默认常用这种缓冲模式。

size：指定缓冲区的大小。

返回值：成功返回 0，失败将返回一个非 0 值，并且会设置 errno 来指示错误原因。

需要注意的是，当 stdio 缓冲区中的数据被刷入到内核缓冲区或被读取之后，这些数据就不会存在于缓冲区中了，数据被刷入了内核缓冲区或被读走了。

㈡、setbuf()函数

setbuf()函数构建与 setvbuf()之上，执行类似的任务，其函数原型如下所示：

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
```

setbuf()调用除了不返回函数结果（void）外，就相当于：

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

要么将 buf 设置为 NULL 以表示无缓冲，要么指向由调用者分配的 BUFSIZ 个字节大小的缓冲区（BUFSIZ 定义于头文件<stdio.h>中，该值通常为 8192）。

㈢、setbuffer()函数

setbuffer()函数类似于 setbuf()，但允许调用者指定 buf 缓冲区的大小，其函数原型如下所示：

```
#include <stdio.h>

void setbuffer(FILE *stream, char *buf, size_t size);
```

setbuffer()调用除了不返回函数结果（void）外，就相当于：

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, size);
```

关于标准 I/O 库 stdio 缓冲相关的内容就给大家介绍这么多，接下来我们进行一些测试，来说明无缓冲、行缓冲以及全缓冲区之间的区别。

标准输出 printf()的行缓冲模式测试

我们先看看下面这个简单地示例代码，调用了 printf()函数，区别在于第二个 printf()没有输出换行符。

示例代码 4.9.4 printf()输出测试

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

int main(void)
{
    printf("Hello World!\n");
    printf("Hello World!");

    for ( ; ; )
        sleep(1);
}
```

printf()函数是标准 I/O 库函数，向终端设备（标准输出）输出打印信息，编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
```

图 4.9.5 运行结果

运行之后可以发现只有第一个 printf()打印的信息显示出来了，第二个并没有显示出来，这是为什么呢？这就是 stdio 缓冲的问题，前面提到了标准输出默认采用的是行缓冲模式，printf()输出的字符串写入到了标准输出的 stdio 缓冲区中，只有输出换行符时（不考虑缓冲区填满的情况）才会将这一行数据刷入到内核缓冲区，也就是写入标准输出文件（终端设备），因为第一个 printf()包含了换行符，所以已经刷入了内核缓冲区，而第二个 printf 并没有包含换行符，所以第二个 printf 输出的"Hello World!"还缓存在 stdio 缓冲区中，需要等待一个换行符才可输出到终端。

联系 4.8.2 小节介绍的格式化输入 scanf()函数，程序中调用 scanf()函数进行阻塞，用户通过键盘输入数据，只有在按下回车键（换行符键）时程序才会接着往下执行，因为标准输入默认也是采用了行缓冲模式。

譬如对示例代码 4.9.4 进行修改，使标准输出变成无缓冲模式，修改后代码如下所示：

示例代码 4.9.5 将标准输出配置为无缓冲模式

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

int main(void)
{
    /* 将标准输出设置为无缓冲模式 */
```

```
    if (setvbuf(stdout, NULL, _IONBF, 0)) {
        perror("setvbuf error");
        exit(0);
    }

    printf("Hello World!\n");
    printf("Hello World!");

    for ( ; ; )
        sleep(1);
}
```

在使用 printf()之前，调用 setvbuf()函数将标准输出的 stdio 缓冲设置为无缓冲模式，接着编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Hello World!
```

图 4.9.6 无缓冲标准输出测试结果

可以发现该程序却能够成功输出两个“Hello World!”，并且白色的光标在第二个“Hello World!”后面，意味着输出没有换行，与程序中第二个 printf 没有加换行符的效果是一直。

所以通过以上两个示例代码对比可知，标准输出默认是行缓冲模式，只有输出了换行符时，才会将换行符这一行字符进行输出显示（也就是刷入到内核缓冲区），在没有输出换行符之前，会将数据缓存在 stdio 缓冲区中。

刷新 stdio 缓冲区

无论我们采取何种缓冲模式，在任何时候都可以使用库函数 fflush()来强制刷新（将输出到 stdio 缓冲区中的数据写入到内核缓冲区，通过 write()函数）stdio 缓冲区，该函数会刷新指定文件的 stdio 输出缓冲区，此函数原型如下所示：

```
#include <stdio.h>

int fflush(FILE *stream);
```

参数 stream 指定需要进行强制刷新的文件，如果该参数设置为 NULL，则表示刷新所有的 stdio 缓冲区。

函数调用成功返回 0，否则将返回-1，并设置 errno 以指示错误原因。

接下来我们对示例代码 4.9.4 进行修改，在第二个 printf 后面调用 fflush()函数，修改后示例代码如下所示：

示例代码 4.9.6 使用 fflush()刷新 stdio 缓冲区

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

int main(void)
```

```
{
    printf("Hello World!\n");
    printf("Hello World!");
    fflush(stdout); //刷新标准输出 stdio 缓冲区

    for ( ; ; )
        sleep(1);
}
```

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Hello World!
```

图 4.9.7 刷新缓冲区

可以看到，打印了两次“Hello World!”，这就是 fflush()的作用了强制刷新 stdio 缓冲区。

除了使用库函数 fflush()之外，还有其它方法会自动刷新 stdio 缓冲区吗？是的，使用库函数 fflush()是一种强制刷新的手段，在一些其它的情况下，也会自动刷新 stdio 缓冲区，譬如当文件关闭时、程序退出时，接下来我们进行演示。

㈠、关闭文件时刷新 stdio 缓冲区

同样还是直接对示例代码 4.9.4 进行修改，在调用第二个 printf 函数后关闭标准输出，如下所示：

示例代码 4.9.7 关闭标准输出

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

int main(void)
{
    printf("Hello World!\n");
    printf("Hello World!");
    fclose(stdout); //关闭标准输出

    for ( ; ; )
        sleep(1);
}
```

至于运行结果文档中就不贴出来了，运行结果与图 4.9.7 是一样的。所以由此可知，文件关闭时系统会自动刷新该文件的 stdio 缓冲区。

㈡、程序退出时刷新 stdio 缓冲区

可以看到上面使用的测试程序中，在最后都使用了一个 for 死循环，让程序处于休眠状态无法退出，为什么要这样做呢？原因在于程序退出时也会自动刷新 stdio 缓冲区，这样的话就会影响到测试结果。同样对示例代码 4.9.4 进行修改，去掉 for 死循环，让程序结束，修改完之后如下所示：

示例代码 4.9.8 程序结束

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

int main(void)
{
    printf("Hello World!\n");
    printf("Hello World!");
}
```

运行结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Hello World!dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 4.9.8 测试结果

从结果可知，当程序退出时，确实会自动刷新 stdio 缓冲区。但是，与程序退出方式有关，如果使用 exit()、return 或像上述示例代码一样不显式调用相关函数或执行 return 语句来结束程序，这些情况下程序终止时会自动刷新 stdio 缓冲区；如果使用_exit 或_Exit()终止程序则不会刷新，这里各位读者可以自行测试、验证。

关于刷新 stdio 缓冲区相关内容，最后进行一个总结：

- 调用 fflush()库函数可强制刷新指定文件的 stdio 缓冲区；
- 调用 fclose()关闭文件时会自动刷新文件的 stdio 缓冲区；
- 程序退出时会自动刷新 stdio 缓冲区（注意区分不同的情况）。

关于本小节内容就给大家介绍这么多，笔者觉得已经非常详细了，如果还有不太理解的地方，希望大家能够自己动手进行测试、验证，然后总结出相应的结论，前面笔者一直强调，编程是一门实践性很强的工作，一定要学会自己分析、验证。

4.9.5 I/O 缓冲小节

本小节对前面学习的内容进行一个简单地总结，概括说明文件 I/O 内核缓冲区和 stdio 缓冲区之间的联系与区别，以及各种 stdio 库函数，如下图所示：

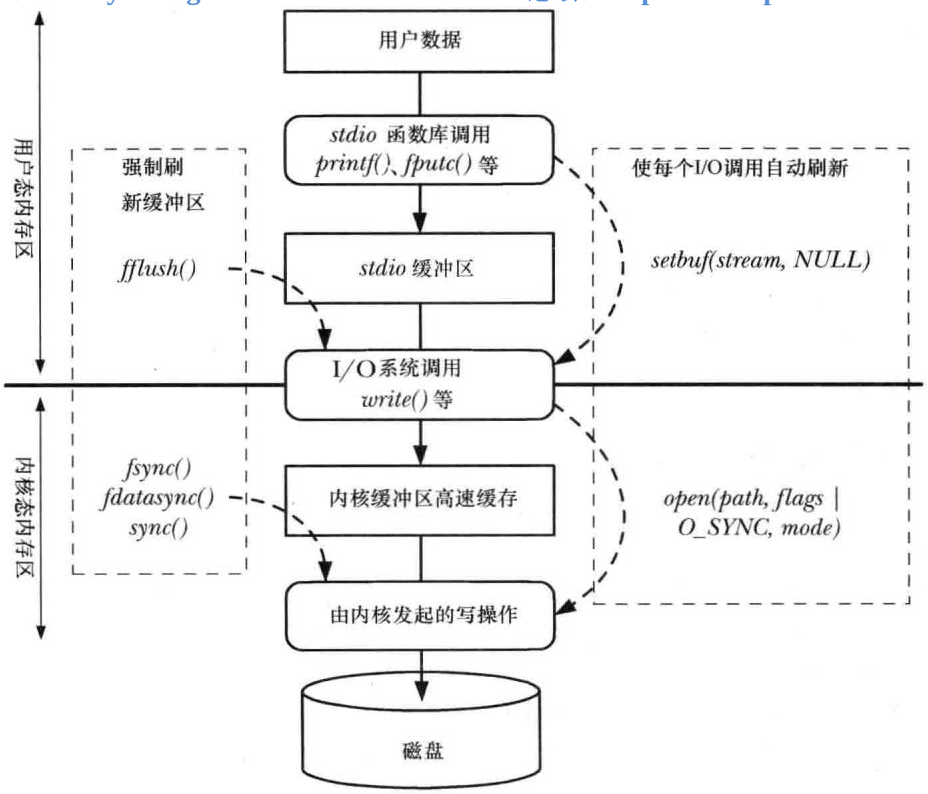


图 4.9.9 I/O 缓冲小结

从图中自上而下，首先应用程序调用标准 I/O 库函数将用户数据写入到 stdio 缓冲区中，stdio 缓冲区是由 stdio 库所维护的用户空间缓冲区。针对不同的缓冲模式，当满足条件时，stdio 库会调用文件 I/O（系统调用 I/O）将 stdio 缓冲区中缓存的数据写入到内核缓冲区中，内核缓冲区位于内核空间。最终由内核向磁盘设备发起读写操作，将内核缓冲区中的数据写入到磁盘（或者从磁盘设备读取数据到内核缓冲区）。

应用程序调用库函数可以对 stdio 缓冲区进行相应的设置，设置缓冲区缓冲模式、缓冲区大小以及由调用者指定一块空间作为 stdio 缓冲区，并且可以强制调用 fflush()函数刷新缓冲区；而对于内核缓冲区来说，应用程序可以调用相关系统调用对内核缓冲区进行控制，譬如调用 fsync()、fdatasync()或 sync()来刷新内核缓冲区（或通过 open 指定 O_SYNC 或 O_DSYNC 标志），或者使用直接 I/O 绕过内核缓冲区（open 函数指定 O_DIRECT 标志）。

4.10 文件描述符与 FILE 指针互转

在应用程序中，在同一个文件上执行 I/O 操作时，还可以将文件 I/O（系统调用 I/O）与标准 I/O 混合使用，这个时候我们就需要将文件描述符和 FILE 指针对象之间进行转换，此时可以借助于库函数 fdopen()、fileno()来完成。

库函数 fileno()可以将标准 I/O 中使用的 FILE 指针转换为文件 I/O 中所使用的文件描述符，而 fdopen()则进行着相反的操作，其函数原型如下所示：

```
#include <stdio.h>

int fileno(FILE *stream);
FILE *fdopen(int fd, const char *mode);
```

首先使用这两个函数需要包含头文件<stdio.h>。

对于 fileno()函数来说，根据传入的 FILE 指针得到整数文件描述符，通过返回值得到文件描述符，如果转换错误将返回-1，并且会设置 errno 来指示错误原因。得到文件描述符之后，便可以使用诸如 read()、write()、lseek()、fcntl()等文件 I/O 方式操作文件。

fdopen()函数与 fileno()功能相反，给定一个文件描述符，得到该文件对应的 FILE 指针，之后便可以使用诸如 fread()、fwrite()等标准 I/O 方式操作文件了。参数 mode 与 fopen()函数中的 mode 参数含义相同，具体参考表 4.4.1 中所述，若该参数与文件描述符 fd 的访问模式不一致，则会导致调用 fdopen()失败。

当混合使用文件 I/O 和标准 I/O 时，需要特别注意缓冲的问题，文件 I/O 会直接将数据写入到内核缓冲区进行高速缓存，而标准 I/O 则会将数据写入到 stdio 缓冲区，之后再调用 write()将 stdio 缓冲区中的数据写入到内核缓冲区。譬如下面这段代码：

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

int main(void)
{
    printf("print");
    write(STDOUT_FILENO, "write\n", 6);
    exit(0);
}
```

执行结果你会发现，先输出了"write"字符串信息，接着再输出了"print"字符串信息，产生这个问题的原因很简单，大家自己去思考下！

第五章 文件属性与目录

在前面的章节内容中，都是围绕普通文件 I/O 操作进行的一系列讨论，譬如打开文件、读写文件、关闭文件等，本章将抛开文件 I/O 相关话题，来讨论 Linux 文件系统的其它特性以及文件相关属性；我们将从系统调用 stat 开始，可利用其返回一个包含多种文件属性（包括文件时间戳、文件所有权以及文件权限等）的结构体，逐个说明 stat 结构中的每一个成员以了解文件的所有属性，然后将向大家介绍用以改变文件属性的各种系统调用；除此之外，还会向大家介绍 Linux 系统中的符号链接以及目录相关的操作。

本章将会讨论如下主题内容。

- Linux 系统的文件类型；
- stat 系统调用；
- 文件各种属性介绍：文件属主、访问权限、时间戳；
- 符号链接与硬链接；
- 目录；
- 删除文件与文件重命名。

5.1 Linux 系统中的文件类型

Linux 下一切皆文件，文件作为 Linux 系统设计思想的核心理念，在 Linux 系统下显得尤为重要。在前面章节内容中，我们都是以普通文件（文本文件、二进制文件等）为例来给大家讲解文件 I/O 相关的知识内容；虽然在 Linux 系统中大部分文件都是普通文件，但并不仅仅只有普通文件，那么本小节将向大家介绍 Linux 系统中的文件类型。

在 Windows 系统下，操作系统识别文件类型一般是通过文件名后缀来判断，譬如 C 语言头文件.h、C 语言源文件.c、.txt 文本文件、压缩包文件.zip 等，在 Windows 操作系统下打开文件，首先会识别文件名后缀得到该文件的类型，然后再使用相应的调用相应的程序去打开它；譬如.c 文件，则会使用 C 代码编辑器去打开它；.zip 文件，则会使用解压软件去打开它。

但是在 Linux 系统下，并不会通过文件后缀名来识别一个文件的类型，话虽如此，但并不是意味着大家可以随便给文件加后缀；文件名也好、后缀也好都是给“人”看的，虽然 Linux 系统并不会通过后缀来识别文件，但是文件后缀也要规范、需要根据文件本身的功能属性来添加，譬如 C 源文件就以.c 为后缀、C 头文件就以.h 为后缀、shell 脚本文件就以.sh 为后缀、这是为了我们自己方便查看、浏览。

Linux 系统下一共分为 7 种文件类型，下面依次给大家介绍。

5.1.1 普通文件

普通文件（regular file）在 Linux 系统下是最常见的，譬如文本文件、二进制文件，我们编写的源代码文件这些都是普通文件，也就是一般意义上的文件。普通文件中的数据存在系统磁盘中，可以访问文件中的内容，文件中的内容以字节为单位进行存储于访问。

普通文件可以分为两大类：文本文件和二进制文件。

- **文本文件：**文件中的内容是由文本构成的，所谓文本指的是 ASCII 码字符。文件中的内容其本质上都是数字（因为计算机本身只有 0 和 1，存储在磁盘上的文件内容也都是由 0 和 1 所构成），而文本文件中的数字应该被理解为这个数字所对应的 ASCII 字符码；譬如常见的.c、.h、.sh、.txt 等这些都是文本文件，文本文件的好处就是方便人阅读、浏览以及编写。
- **二进制文件：**二进制文件中存储的本质上也是数字，只不过对于二进制文件来说，这些数字并不是文本字符编码，而是真正的数字。譬如 Linux 系统下的可执行文件、C 代码编译之后得到的.o 文件、.bin 文件等都是二进制文件。

在 Linux 系统下，可以通过 stat 命令或者 ls 命令来查看文件类型，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat testApp.c
  文件： 'testApp.c'
  大小： 732             块： 8          IO 块： 4096   普通文件
设备： 801h/2049d        Inode: 3701845     硬链接： 1
权限： (0664/-rw-rw-r--)  Uid: ( 1000/      dt)   Gid: ( 1000/      dt)
最近访问： 2021-01-12 18:32:03.192340301 +0800
最近更改： 2021-01-12 18:31:43.581326690 +0800
最近改动： 2021-01-12 18:31:43.581326690 +0800
创建时间： -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.1.1 stat 查看文件类型

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l testApp.c
-rw-rw-r-- 1 dt dt 732 1月  12 18:31 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.1.2 ls 查看文件类型

stat 命令非常友好，会直观把文件类型显示出来；对于 ls 命令来说，并没有直观的显示出文件的类型，而是通过符号表示出来，在图 5.1.2 中画红色框位置显示出的一串字符中，其中第一个字符（'-'）就用于表示文件的类型，减号'-'就表示该文件是一个普通文件；除此之外，来看看其它文件类型使用什么字符表示：

- '-'：普通文件
- 'd'：目录文件
- 'c'：字符设备文件
- 'b'：块设备文件
- 'l'：符号链接文件
- 's'：套接字文件
- 'p'：管道文件

关于普通文件就给大家介绍这么多。

5.1.2 目录文件

目录（directory）就是文件夹，文件夹在 Linux 系统中也是一种文件，是一种特殊文件，同样我们也可以使用 vi 编辑器来打开文件夹，如下所示：

```
" ============================================================================
" Netrw Directory Listing                                    (netrw v155)
"   /home/dt/vscode_ws/2_chapter
"   Sorted by      name
"   Sort sequence: [\/]$,\<core\%(\.\d\+\)\=\>,\.h$,\.c$,\.cpp$,\~\=\*$,*,\.o$,\.obj$,\.info$,\.swp$,\.bak$,\~$
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:special
" ============================================================================
../
./
testApp.c
testApp*
test_file
~
```

图 5.1.3 使用 vi 打开文件夹

可以看到，文件夹中记录了该文件夹本省的路径以及该文件夹下所存放的文件。文件夹作为一种特殊文件，本身并不适合使用前面给大家介绍的文件 I/O 的方式来读写，在 Linux 系统下，会有一些专门的系统调用用于读写文件夹，这部分内容后面再给大家介绍。

5.1.3 字符设备文件和块设备文件

学过 Linux 驱动编程开发的读者，对字符设备文件（character）、块设备文件（block）这些文件类型应该并不陌生，Linux 系统下，一切皆文件，也包括各种硬件设备。设备文件（字符设备文件、块设备文件）对应的是硬件设备，在 Linux 系统中，硬件设备会对应到一个设备文件，应用程序通过对设备文件的读写来操控、使用硬件设备，譬如 LCD 显示屏、串口、音频、按键等。

Linux 系统中，可将硬件设备分为字符设备和块设备，所以就有了字符设备文件和块设备文件两种文件类型。虽然有设备文件，但是设备文件并不对应磁盘上的一个文件，也就是说设备文件并不存在于磁盘中，而是由文件系统虚拟出来的，一般是由内存来维护，当系统关机时，设备文件都会消失；字符设备文件一般存放在 Linux 系统/dev/目录下，所以/dev 也称为虚拟文件系统 devfs。以 Ubuntu 系统为例，如下所示：

```
dt@dt-virtual-machine:~$ ls -l /dev/
总用量 0
crw-------  1 root root     10, 175 12月  19 12:34 agpgart
crw-------  1 root root     10, 235 12月  19 12:34 autofs
drwxr-xr-x  2 root root          300 12月  19 12:34 block
drwxr-xr-x  2 root root           80 12月  19 12:34 bsg
crw-rw----  1 root disk     10, 234 12月  22 07:16 btrfs-control
drwxr-xr-x  3 root root           60 12月  19 12:34 bus
lrwxrwxrwx  1 root root            3 12月  19 12:34 cdrom -> sr0
lrwxrwxrwx  1 root root            3 12月  19 12:34 cdrw -> sr0
drwxr-xr-x  2 root root         3760 1月    6 06:22 char
crw-------  1 root root      5,    1 12月  19 12:34 console
lrwxrwxrwx  1 root root           11 12月  19 12:34 core -> /proc/kcore
drwxr-xr-x  8 root root          160 1月    6 06:22 cpu
crw-------  1 root root     10,   59 12月  19 12:34 cpu_dma_latency
crw-------  1 root root     10,  203 12月  19 12:34 cuse
drwxr-xr-x  5 root root          100 12月  19 12:34 disk
crw-rw----+ 1 root audio    14,    9 12月  19 12:34 dmmidi
drwxr-xr-x  2 root root           80 12月  19 12:34 dri
lrwxrwxrwx  1 root root            3 12月  19 12:34 dvd -> sr0
crw-------  1 root root     10,   61 12月  19 12:34 ecryptfs
crw-rw----  1 root video    29,    0 12月  19 12:34 fb0
lrwxrwxrwx  1 root root           13 12月  19 12:34 fd -> /proc/self/fd
crw-rw-rw-  1 root root      1,    7 12月  19 12:34 full
crw-rw-rw-  1 root root     10, 229 12月  22 07:16 fuse
crw-------  1 root root    245,    0 12月  19 12:34 hidraw0
crw-------  1 root root     10, 228 12月  19 12:34 hpet
drwxr-xr-x  2 root root            0 12月  19 12:34 hugepages
crw-------  1 root root     10, 183 12月  19 12:34 hwrng
lrwxrwxrwx  1 root root           25 12月  19 12:34 initctl -> /run/systemd/initctl/fifo
drwxr-xr-x  4 root root          260 12月  19 12:34 input
crw-r--r--  1 root root      1,   11 12月  19 12:34 kmsg
drwxr-xr-x  2 root root           60 12月  19 12:34 lightnvm
lrwxrwxrwx  1 root root           28 12月  19 12:34 log -> /run/systemd/journal/dev-log
brw-rw----  1 root disk      7,    0 12月  22 07:16 loop0
brw-rw----  1 root disk      7,    1 12月  19 12:34 loop1
```

图 5.1.4 /dev 目录下的设备文件

上图中 agpgart、autofs、btrfs-control、console 等这些都是字符设备文件，而 loop0、loop1 这些便是块设备文件。

5.1.4 符号链接文件

符号链接文件（link）类似于 Windows 系统中的快捷方式文件，是一种特殊文件，它的内容指向的是另一个文件路径，当对符号链接文件进行操作时，系统根据情况会对这个操作转移到它指向的文件上去，而不是对它本身进行操作，譬如，读取一个符号链接文件内容时，实际上读到的是它指向的文件的内容。

如果大家理解了 Windows 下的快捷方式，那么就会很容易理解 Linux 下的符号链接文件。图 5.1.4 中的 cdrom、cdrw、fd、initctl 等这些文件都是符号链接文件，箭头所指向的文件路径便是符号链接文件所指向的文件。

关于链接文件，在后面的内容中还会给大家进行介绍，这里暂时给大家介绍这么多！

5.1.5 管道文件

管道文件（pipe）主要用于进程间通信，当学习到相关知识内容的时候再给大家详解。

5.1.6 套接字文件

套接字文件（socket）也是一种进程间通信的方式，与管道文件不同的是，它们可以在不同主机上的进程间通信，实际上就是网络通信，当学习到网络编程相关知识内容再给大家介绍。

5.1.7 总结

本小节给大家简单地介绍了 Linux 系统中的 7 种文件类型，包括：普通文件、目录、字符设备文件、块设备文件、符号链接文件、管道文件以及套接字文件，下面对它们进行一个简单地概括：

普通文件是最常见的文件类型；
目录也是一种文件类型；
设备文件对应于硬件设备；
符号链接文件类似于 Windows 的快捷方式；
管道文件用于进程间通信；
套接字文件用于网络通信。

5.2 stat 函数

Linux 下可以使用 stat 命令查看文件的属性，其实这个命令内部就是通过调用 stat()函数来获取文件属性的，stat 函数是 Linux 中的系统调用，用于获取文件相关的信息，函数原型如下所示（可通过"man 2 stat"命令查看）：

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

int stat(const char *pathname, struct stat *buf);
```

首先使用该函数需要包含<sys/types.h>、<sys/stat.h>以及<unistd.h>这三个头文件。

函数参数及返回值含义如下：

pathname：用于指定一个需要查看属性的文件路径。

buf：struct stat 类型指针，用于指向一个 struct stat 结构体变量。调用 stat 函数的时候需要传入一个 struct stat 变量的指针，获取到的文件属性信息就记录在 struct stat 结构体中，稍后给大家介绍 struct stat 结构体中有记录了哪些信息。

返回值：成功返回 0；失败返回-1，并设置 error。

5.2.1 struct stat 结构体

struct stat 是内核定义的一个结构体，在<sys/stat.h>头文件中申明，所以可以在应用层使用，这个结构体中的所有元素加起来构成了文件的属性信息，结构体内容如下所示：

示例代码 5.2.1 struct stat 结构体

```
struct stat
{
    dev_t st_dev;           /* 文件所在设备的 ID */
    ino_t st_ino;           /* 文件对应 inode 节点编号 */
    mode_t st_mode;         /* 文件对应的模式 */
    nlink_t st_nlink;       /* 文件的链接数 */
    uid_t st_uid;           /* 文件所有者的用户 ID */
    gid_t st_gid;           /* 文件所有者的组 ID */
    dev_t st_rdev;          /* 设备号（指针对设备文件） */
    off_t st_size;          /* 文件大小（以字节为单位） */
    blksize_t st_blksize;   /* 文件内容存储的块大小 */
    blkcnt_t st_blocks;     /* 文件内容所占块数 */
    struct timespec st_atim;    /* 文件最后被访问的时间 */
    struct timespec st_mtim;    /* 文件内容最后被修改的时间 */
```

```
    struct timespec st_ctim;       /* 文件状态最后被改变的时间 */
};
```

st_dev：该字段用于描述此文件所在的设备。不常用，可以不用理会。

st_ino：文件的 inode 编号。

st_mode：该字段用于描述文件的模式，譬如文件类型、文件权限都记录在该变量中，关于该变量的介绍请看 5.2.2 小节。

st_nlink：该字段用于记录文件的硬链接数，也就是为该文件创建了多少个硬链接文件。链接文件可以分为软链接（符号链接）文件和硬链接文件，关于这些内容后面再给大家介绍。

st_uid、st_gid：此两个字段分别用于描述文件所有者的用户 ID 以及文件所有者的组 ID，后面再给大家介绍。

st_rdev：该字段记录了设备号，设备号只针对于设备文件，包括字符设备文件和块设备文件，不用理会。

st_size：该字段记录了文件的大小（逻辑大小），以字节为单位。

st_atim、st_mtim、st_ctim：此三个字段分别用于记录文件最后被访问的时间、文件内容最后被修改的时间以及文件状态最后被改变的时间，都是 struct timespec 类型变量，具体介绍请看 5.2.3 小节。

5.2.2 st_mode 变量

st_mode 是 struct stat 结构体中的一个成员变量，是一个 32 位无符号整形数据，该变量记录了文件的类型、文件的权限这些信息，其表示方法如下所示：

图 5.2.1 st_mode 数据信息示意图

看到图 5.2.1 的时候，大家有没有似曾相识的感觉，确实，前面章节内容给大家介绍 open 函数的第三个参数 mode 时也用到了类似的图，如图 2.3.2 所示。唯一不同的在于 open 函数的 mode 参数只涉及到 S、U、G、O 这 12 个 bit 位，并不包括用于描述文件类型的 4 个 bit 位。

O 对应的 3 个 bit 位用于描述其它用户的权限；

G 对应的 3 个 bit 位用于描述同组用户的权限；

U 对应的 3 个 bit 位用于描述文件所有者的权限；

S 对应的 3 个 bit 位用于描述文件的特殊权限。

这些 bit 位表达内容与 open 函数的 mode 参数相对应，这里不再重述。同样，在 mode 参数中表示权限的宏定义，在这里也是可以使用的，这些宏定义如下（以下数字使用的是八进制方式表示）：

S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	group has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	others (not in group) have read, write, and execute permission

```
S_IROTH         00004    others have read permission
S_IWOTH         00002    others have write permission
S_IXOTH         00001    others have execute permission
```

譬如，判断文件所有者对该文件是否具有可执行权限，可以通过以下方法测试（假设 st 是 struct stat 类型变量）：

```
if (st.st_mode & S_IXUSR) {
    //有权限
} else {
    //无权限
}
```

这里我们重点来看看“文件类型”这 4 个 bit 位，这 4 个 bit 位用于描述该文件的类型，譬如该文件是普通文件、还是链接文件、亦或者是一个目录等，那么就可以通过这 4 个 bit 位数据判断出来，如下所示：

```
S_IFSOCK    0140000    socket（套接字文件）
S_IFLNK     0120000    symbolic link（链接文件）
S_IFREG     0100000    regular file（普通文件）
S_IFBLK     0060000    block device（块设备文件）
S_IFDIR     0040000    directory（目录）
S_IFCHR     0020000    character device（字符设备文件）
S_IFIFO     0010000    FIFO（管道文件）
```

注意上面这些数字使用的是八进制方式来表示的，在 C 语言中，八进制方式表示一个数字需要在数字前面添加一个 0（零）。所以由上面可知，当“文件类型”这 4 个 bit 位对应的数字是 14（八进制）时，表示该文件是一个套接字文件、当“文件类型”这 4 个 bit 位对应的数字是 12（八进制）时，表示该文件是一个链接文件、当“文件类型”这 4 个 bit 位对应的数字是 10（八进制）时，表示该文件是一个普通文件等。

所以通过 st_mode 变量判断文件类型就很简单了，如下（假设 st 是 struct stat 类型变量）：

```
/* 判断是不是普通文件 */
if ((st.st_mode & S_IFMT) == S_IFREG) {
    /* 是 */
}

/* 判断是不是链接文件 */
if ((st.st_mode & S_IFMT) == S_IFLNK) {
    /* 是 */
}
```

S_IFMT 宏是文件类型字段位掩码：

```
S_IFMT      0170000
```

除了这样判断之外，我们还可以使用 Linux 系统封装好的宏来进行判断，如下所示（m 是 st_mode 变量）：

```
S_ISREG(m)      #判断是不是普通文件，如果是返回 true，否则返回 false
S_ISDIR(m)      #判断是不是目录，如果是返回 true，否则返回 false
S_ISCHR(m)      #判断是不是字符设备文件，如果是返回 true，否则返回 false
S_ISBLK(m)      #判断是不是块设备文件，如果是返回 true，否则返回 false
S_ISFIFO(m)     #判断是不是管道文件，如果是返回 true，否则返回 false
S_ISLNK(m)      #判断是不是链接文件，如果是返回 true，否则返回 false
```

```
S_ISSOCK(m)      #判断是不是套接字文件，如果是返回 true，否则返回 false
```

有了这些宏之后，就可以通过如下方式来判断文件类型了：

```
/* 判断是不是普通文件 */
if (S_ISREG(st.st_mode)) {
    /* 是 */
}

/* 判断是不是目录 */
if (S_ISDIR(st.st_mode)) {
    /* 是 */
}
```

关于 st_mode 变量就给大家介绍这么多。

5.2.3 struct timespec 结构体

该结构体定义在<time.h>头文件中，是 Linux 系统中时间相关的结构体。应用程序中包含了<time.h>头文件，就可以在应用程序中使用该结构体了，结构体内容如下所示：

示例代码 5.2.2 struct timespec 结构体

```
struct timespec
{
    time_t tv_sec;              /* 秒 */
    syscall_slong_t tv_nsec;    /* 纳秒 */
};
```

struct timespec 结构体中只有两个成员变量，一个秒（tv_sec）、一个纳秒（tv_nsec），time_t 其实指的就是 long int 类型，所以由此可知，该结构体所表示的时间可以精确到纳秒，当然，对于文件的时间属性来说，并不需要这么高的精度，往往只需精确到秒级别即可。

在 Linux 系统中，time_t 时间指的是一个时间段，从某一个时间点到某一个时间点所经过的秒数，譬如对于文件的三个时间属性来说，指的是从过去的某一个时间点（这个时间点是一个起始基准时间点）到文件最后被访问、文件内容最后被修改、文件状态最后被改变的这个时间点所经过的秒数。time_t 时间在 Linux 下被称为日历时间，7.2 小计中对此有详细介绍。

由示例代码 5.2.1 可知，struct stat 结构体中包含了三个文件相关的时间属性，但这里得到的仅仅只是以秒+微秒为单位的时间值，对于我们来说，并不利用查看，我们一般喜欢的是“2020-10-10 18:30:30”这种形式表示的时间，直观、明了，那有没有办法通过秒来得到这种形式表达的时间呢？答案当然是可以，譬如可以通过 localtime()/localtime_r()或者 strftime()来得到更利于我们查看的时间表达方式，关于这些函数的介绍以及使用方法在 7.2.4 小节有详细说明。

5.2.4 练习

到这里本小节内容就给大家介绍完了，主要给大家介绍了 stat 函数以及由此引出来的一系列知识内容。为了巩固本小节所学内容，这里出一些简单地编程练习题，大家可以根据本小节所学知识完成它。

⑴获取文件的 inode 节点编号以及文件大小，并将它们打印出来。

⑵获取文件的类型，判断此文件对于其它用户（Other）是否具有可读可写权限。

⑶获取文件的时间属性，包括文件最后被访问的时间、文件内容最后被修改的时间以及文件状态最后被改变的时间，并使用字符串形式将其打印出来，包括时间和日期、表示形式自定。

以上就是根据本小节内容整理出来的一些简单的编程练习题，下面笔者将给出对应的示例代码。

(1)编程实战练习 1

示例代码 5.2.3 编程实战练习 1

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

int main(void)
{
    struct stat file_stat;
    int ret;

    /* 获取文件属性 */
    ret = stat("./test_file", &file_stat);
    if (-1 == ret) {
        perror("stat error");
        exit(-1);
    }

    /* 打印文件大小和 inode 编号 */
    printf("file size: %ld bytes\n"
           "inode number: %ld\n", file_stat.st_size,
           file_stat.st_ino);
    exit(0);
}
```

测试之前先使用 ls 命令查看 test_file 文件的 inode 节点和大小，如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li test_file
3701841 -rw-rw-r-- 1 dt dt 8864 1月  11 18:09 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.2 ls 命令查看文件的 inode 编号和大小

从图中可以得知，此文件的大小为 8864 个字节，inode 编号为 3701841；接下来编译我们的测试程序、并运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
file size: 8864 bytes
inode number: 3701841
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.3 编程实战 1 测试结果

(2)编程实战练习 2

示例代码 5.2.4 编程实战练习 2

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

int main(void)
{
    struct stat file_stat;
    int ret;

    /* 获取文件属性 */
    ret = stat("./test_file", &file_stat);
    if (-1 == ret) {
        perror("stat error");
        exit(-1);
    }

    /* 判读文件类型 */
    switch (file_stat.st_mode & S_IFMT) {
    case S_IFSOCK: printf("socket"); break;
    case S_IFLNK: printf("symbolic link"); break;
    case S_IFREG: printf("regular file"); break;
    case S_IFBLK: printf("block device"); break;
    case S_IFDIR: printf("directory"); break;
    case S_IFCHR: printf("character device"); break;
    case S_IFIFO: printf("FIFO"); break;
    }

    printf("\n");

    /* 判断该文件对其它用户是否具有读权限 */
    if (file_stat.st_mode & S_IROTH)
        printf("Read: Yes\n");
    else
        printf("Read: No\n");

    /* 判断该文件对其它用户是否具有写权限 */
    if (file_stat.st_mode & S_IWOTH)
        printf("Write: Yes\n");
    else
        printf("Write: No\n");
```

```
    exit(0);
}
```

测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
regular file
Read: Yes
Write: No
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.4 编程实战 2 测试结果

(3)编程实战练习 3

示例代码 5.2.5 编程实战练习 3

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

int main(void)
{
    struct stat file_stat;
    struct tm file_tm;
    char time_str[100];
    int ret;

    /* 获取文件属性 */
    ret = stat("./test_file", &file_stat);
    if (-1 == ret) {
        perror("stat error");
        exit(-1);
    }

    /* 打印文件最后被访问的时间 */
    localtime_r(&file_stat.st_atim.tv_sec, &file_tm);
    strftime(time_str, sizeof(time_str),
                "%Y-%m-%d %H:%M:%S", &file_tm);
    printf("time of last access: %s\n", time_str);

    /* 打印文件内容最后被修改的时间 */
    localtime_r(&file_stat.st_mtim.tv_sec, &file_tm);
    strftime(time_str, sizeof(time_str),
                "%Y-%m-%d %H:%M:%S", &file_tm);
```

```
    printf("time of last modification: %s\n", time_str);

    /* 打印文件状态最后改变的时间 */
    localtime_r(&file_stat.st_ctim.tv_sec, &file_tm);
    strftime(time_str, sizeof(time_str),
                    "%Y-%m-%d %H:%M:%S", &file_tm);
    printf("time of last status change: %s\n", time_str);

    exit(0);
}
```

测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
time of last access: 2021-01-12 14:52:03
time of last modification: 2021-01-11 18:09:28
time of last status change: 2021-01-12 14:51:39
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.5 实战编程 3 测试结果

可以使用 stat 命令查看 test_file 文件的这些时间属性，对比程序打印出来是否正确：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
  文件：'test_file'
  大小：8864            块：24         IO 块：4096   普通文件
设备：801h/2049d        Inode：3701841     硬链接：1
权限：(0664/-rw-rw-r--)  Uid：( 1000/      dt)   Gid：( 1000/      dt)
最近访问：2021-01-12 14:52:03.485262883 +0800
最近更改：2021-01-11 18:09:28.641178667 +0800
最近改动：2021-01-12 14:51:39.136452379 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.2.6 stat 命令查看文件的时间属性

5.3 fstat 和 lstat 函数

前面给大家介绍了 stat 系统调用，起始除了 stat 函数之外，还可以使用 fstat 和 lstat 两个系统调用来获取文件属性信息。fstat、lstat 与 stat 的作用一样，但是参数、细节方面有些许不同。

5.3.1 fstat 函数

fstat 与 stat 区别在于，stat 是从文件名出发得到文件属性信息，不需要先打开文件；而 fstat 函数则是从文件描述符出发得到文件属性信息，所以使用 fstat 函数之前需要先打开文件得到文件描述符。具体该用 stat 还是 fstat，看具体的情况；譬如，并不想通过打开文件来得到文件属性信息，那么就使用 stat，如果文件已经打开了，那么就使用 fstat。

fstat 函数原型如下（可通过"man 2 fstat"命令查看）：

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

```
int fstat(int fd, struct stat *buf);
```

第一个参数 fd 表示文件描述符，第二个参数以及返回值与 stat 一样。fstat 函数使用示例如下：

示例代码 5.3.1 fstat 函数使用示例

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

int main(void)
{
    struct stat file_stat;
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 获取文件属性 */
    ret = fstat(fd, &file_stat);
    if (-1 == ret)
        perror("fstat error");

    close(fd);
    exit(ret);
}
```

5.3.2 lstat 函数

lstat()与 stat、fstat 的区别在于，对于符号链接文件，stat、fstat 查阅的是符号链接文件所指向的文件对应的文件属性信息，而 lstat 查阅的是符号链接文件本身的属性信息。

lstat 函数原型如下所示：

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

int lstat(const char *pathname, struct stat *buf);
```

函数参数列表、返回值与 stat 函数一样，使用方法也一样，这里不再重述！

5.4 文件属主

Linux 是一个多用户操作系统，系统中一般存在着好几个不同的用户，而 Linux 系统中的每一个文件都有一个与之相关联的用户和用户组，通过这个信息可以判断文件的所有者和所属组。

文件所有者表示该文件属于“谁”，也就是属于哪个用户。一般来说文件在创建时，其所有者就是创建该文件的那个用户。譬如，当前登录用户为 dt，使用 touch 命令创建了一个文件，那么这个文件的所有者就是 dt；同理，在程序中调用 open 函数创建新文件时也是如此，执行该程序的用户是谁，其文件所有者便是谁。

文件所属组则表示该文件属于哪一个用户组。在 Linux 中，系统并不是通过用户名或用户组名来识别不同的用户和用户组，而是通过 ID。ID 就是一个编号，Linux 系统会为每一个用户或用户组分配一个 ID，将用户名或用户组名与对应的 ID 关联起来，所以系统通过用户 ID（UID）或组 ID（GID）就可以识别出不同的用户和用户组。

Tips：用户 ID 简称 UID、用户组 ID 简称 GID。这些都是 Linux 操作系统的基础知识，如果对用户和用户组的概念尚不熟悉，建议先自行学习这些基础知识。

譬如使用 ls 命令或 stat 命令便可以查看到文件的所有者和所属组，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l testApp.c
-rw-rw-rw- 1 dt dt 117 1月  14 20:25 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat testApp.c
  文件： 'testApp.c'
  大小： 117            块： 8          IO 块： 4096   普通文件
设备： 801h/2049d       Inode: 3701453     硬链接： 1
权限： (0666/-rw-rw-rw-)  Uid: ( 1000/      dt)   Gid: ( 1000/      dt)
最近访问： 2021-01-15 11:07:26.542747368 +0800
最近更改： 2021-01-14 20:25:12.272141838 +0800
最近改动： 2021-01-15 11:39:11.816567723 +0800
创建时间： -
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.1 查看文件的所有者和所属组

由上图可知，testApp.c 文件的用户 ID 是 1000，用户组 ID 也是 1000。

文件的用户 ID 和组 ID 分别由 struct stat 结构体中的 st_uid 和 st_gid 所指定。既然 Linux 下的每一个文件都有与之相关联的用户 ID 和组 ID，那么对于一个进程来说亦是如此，与一个进程相关联的 ID 有 5 个或更多，如下表所示：

表 5.4.1 与进程相关联的用户 ID 和组 ID

ID 类型	作用
实际用户 ID	我们实际上是谁
实际组 ID	
有效用户 ID	用于文件访问权限检查
有效组 ID	
附属组 ID	

- 实际用户 ID 和实际组 ID 标识我们究竟是谁，也就是执行该进程的用户是谁、以及该用户对应的所属组；实际用户 ID 和实际组 ID 确定了进程所属的用户和组。
- 进程的有效用户 ID、有效组 ID 以及附属组 ID 用于文件访问权限检查，详情请查看 5.4.1 小节内容。

5.4.1 有效用户 ID 和有效组 ID

首先对于有效用户 ID 和有效组 ID 来说，这是进程所持有的概念，对于文件来说，并无此属性！有效用户 ID 和有效组 ID 是站在操作系统的角度，用于给操作系统判断当前执行该进程的用户在当前环境下对某个文件是否拥有相应的权限。

在 Linux 系统中，当进程对文件进行读写操作时，系统首先会判断该进程是否具有对该文件的读写权限，那如何判断呢？自然是通过该文件的权限位来判断，struct stat 结构体中的 st_mode 字段中就记录了该文件的权限位以及文件类型。关于文件权限检查相关内容将会在 5.5 小节中说明。

当进行权限检查时，并不是通过进程的实际用户和实际组来参与权限检查的，而是通过有效用户和有效组来参与文件权限检查。通常，绝大部分情况下，进程的有效用户等于实际用户（有效用户 ID 等于实际用户 ID），有效组等于实际组（有效组 ID 等于实际组 ID）。

那么大家可能就要问了，什么情况下有效用户 ID 不等于实际用户 ID、有效组 ID 不等于实际组 ID？那么关于这个问题，后面将给大家揭晓！

Tips：文中所指的"进程对文件是否拥有 xx 权限"其实质是当前执行该进程的用户是否拥有对文件的 xx 权限。若无特别指出，文中的描述均为此意！

5.4.2 chown 函数

chown 是一个系统调用，该系统调用可用于改变文件的所有者（用户 ID）和所属组（组 ID）。其实在 Linux 系统下也有一个 chown 命令，该命令的作用也是用于改变文件的所有者和所属组，譬如将 testApp.c 文件的所有者和所属组修改为 root：

```
sudo chown root:root testApp.c
```

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 0
-rw-rw-r-- 1 dt dt 0 1月  19 12:29 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo chown root:root testApp.c
[sudo] dt 的密码：
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 0
-rw-rw-r-- 1 root root 0 1月  19 12:29 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.2 使用 chown 命令修改文件所有者和所属组

可以看到，通过该命令确实可以改变文件的所有者和所属组，这个命令内部其实就是调用了 chown 函数来实现功能的，chown 函数原型如下所示（可通过"man 2 chown"命令查看）：

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
```

首先，使用该命令需要包含头文件<unistd.h>。

函数参数和返回值如下所示：

pathname：用于指定一个需要修改所有者和所属组的文件路径。

owner：将文件的所有者修改为该参数指定的用户（以用户 ID 的形式描述）；

group：将文件的所属组修改为该参数指定的用户组（以用户组 ID 的形式描述）；

返回值：成功返回 0；失败将返回-1，兵并且会设置 errno。

该函数的用法非常简单，只需指定对应的文件路径以及相应的 owner 和 group 参数即可！如果只需要修改文件的用户 ID 和用户组 ID 当中的一个，那么又该如何做呢？方法很简单，只需将其中不用修改的 ID（用

户 ID 或用户组 ID）与文件当前的 ID（用户 ID 或用户组 ID）保持一致即可，即调用 chown 函数时传入的用户 ID 或用户组 ID 就是该文件当前的用户 ID 或用户组 ID，而文件当前的用户 ID 或用户组 ID 可以通过 stat 函数查询获取。

虽然该函数用法很简单，但是有以下两个限制条件：

- 只有超级用户进程能更改文件的用户 ID；
- 普通用户进程可以将文件的组 ID 修改为其所从属的任意附属组 ID，前提条件是该进程的有效用户 ID 等于文件的用户 ID；而超级用户进程可以将文件的组 ID 修改为任意值。

所以，由此可知，文件的用户 ID 和组 ID 并不是随随便便就可以更改的，其实这种设计是为系统安全着想，如果系统中的任何普通用户进程都可以随便更改系统文件的用户 ID 和组 ID，那么也就意味着任何普通用户对系统文件都有任意权限了，这对于操作系统来说将是非常不安全的。

测试

接下来看一些 chown 函数的使用例程，如下所示：

示例代码 5.4.1 chown 函数使用示例

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

int main(void)
{
    if (-1 == chown("./test_file", 0, 0)) {
        perror("chown error");
        exit(-1);
    }

    exit(0);
}
```

代码很简单，直接调用 chown 函数将 test_file 文件的用户 ID 和用户组 ID 修改为 0、0。0 指的就是 root 用户和 root 用户组，接下来我们测试下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
  文件：'test_file'
  大小：170             块：8          IO 块：4096   普通文件
设备：801h/2049d        Inode: 3670365     硬链接：1
权限：(0664/-rw-rw-r--)  Uid: ( 1000/      dt)   Gid: ( 1000/      dt)
最近访问：2021-01-19 15:34:46.476125110 +0800
最近更改：2021-01-19 15:34:37.799778966 +0800
最近改动：2021-01-19 15:35:20.132591206 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
chown error: Operation not permitted
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.3 chown 测试结果

在运行测试代码之前，先使用了 stat 命令查看到 test_file 文件的用户 ID 和用户组 ID 都等于 1000，然后执行测试程序，结果报错"Operation not permitted"，显示不允许操作；接下来重新执行程序，此时加上 sudo，如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码：
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
  文件：'test_file'
  大小：170             块：8          IO 块：4096   普通文件
设备：801h/2049d        Inode：3670365     硬链接：1
权限：(0664/-rw-rw-r--)  Uid：(    0/    root)   Gid：(    0/    root)
最近访问：2021-01-19 15:34:46.476125110 +0800
最近更改：2021-01-19 15:34:37.799778966 +0800
最近改动：2021-01-19 15:48:41.392432160 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.4 chown 测试结果 2

此时便可以看到，执行之后没有打印错误提示信息，说明 chown 函数调用成功了，并且通过 stat 命令也可以看到文件的用户 ID 和组 ID 确实都被修改为 0 了（也就是 root 用户）。原因在于，加上 sudo 执行应用程序，而此时应用程序便可以临时获得 root 用户的权限，也就是会以 root 用户的身份运行程序，也就意味着此时该应用程序的用户 ID（也就是前面给大家提到的实际用户 ID）变成了 root 超级用户的 ID（也就是 0），自然 chown 函数便可以调用成功。

在 Linux 系统下，可以使用 getuid 和 getgid 两个系统调用分别用于获取当前进程的用户 ID 和用户组 ID，这里说的进程的用户 ID 和用户组 ID 指的就是进程的实际用户 ID 和实际组 ID，这两个系统调用函数原型如下所示：

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

uid_t getuid(void);
gid_t getgid(void);
```

我们可以在示例代码 5.4.1 中加入打印用户 ID 的语句，如下所示：

示例代码 5.4.2 chown 使用示例 2

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

int main(void)
{
    printf("uid: %d\n", getuid());

    if (-1 == chown("./test_file", 0, 0)) {
        perror("chown error");
        exit(-1);
    }

    exit(0);
}
```

再来重复上面的测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
uid: 1000
chown error: Operation not permitted
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
uid: 0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.4.5 chown 测试结果 3

很明显可以看到两次执行同一个应用程序它们的用户 ID 是不一样的，因为加上了 sudo 使得应用程序的用户 ID 由原本的普通用户 ID 1000 变成了超级用户 ID 0，使得该进程变成了超级用户进程，所以调用 chown 函数就不会报错。

关于 chown 就给大家介绍这么多，在实际应用编程中，此系统调用被用到的概率并不多，但是理论性知识还是得知道。

5.4.3 fchown 和 lchown 函数

这两个同样也是系统调用，作用与 chown 函数相同，只是参数、细节方面有些许不同。fchown()、lchown()这两个函数与 chown()的区别就像是 fstat()、lstat()与 stat 的区别，本小节就不再重述这种问题了，如果大家对此还不清楚，可以看 5.3 小节，具体使用 fchown、lchown 还是 chown，看情况而定。

5.5 文件访问权限

struct stat 结构体中的 st_mode 字段记录了文件的访问权限位。当提及到文件时，指的是前面给大家介绍的任何类型的文件，并不仅仅指的是普通文件；所有文件类型（目录、设备文件）都有访问权限（access permission），可能有很多人认为只有普通文件才有访问权限，这是一种误解！

5.5.1 普通权限和特殊权限

文件的权限可以分为两个大类，分别是普通权限和特殊权限（也可称为附加权限）。普通权限包括对文件的读、写以及执行，而特殊权限则包括一些对文件的附加权限，譬如 Set-User-ID、Set-Group-ID 以及 Sticky。接下来，分别对普通权限和特殊权限进行介绍。

普通权限

每个文件都有 9 个普通的访问权限位，可将它们分为 3 类，如下表：

st_mode 权限表示宏	含义
S_IRUSR	文件所有者读权限
S_IWUSR	文件所有者写权限
S_IXUSR	文件所有者执行权限
S_IRGRP	同组用户读权限
S_IWGRP	同组用户写权限
S_IXGRP	同组用户执行权限
S_IROTH	其它用户读权限
S_IWOTH	其它用户写权限
S_IXOTH	其它用户执行权限

表 5.5.1 9 个文件访问权限位

譬如使用 ls 命令或 stat 命令可以查看到文件的这 9 个访问权限，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 20
-rwxrwxr-x 1 dt dt 8816 1月   19 16:16 testApp
-rw-rw-r-- 1 dt dt  203 1月   19 16:16 testApp.c
-rw-rw-r-- 1 dt dt  170 1月   19 15:34 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.1 ls 命令查看文件的 9 个访问权限位

每一行打印信息中，前面的一串字符串就描述了该文件的 9 个访问权限以及文件类型，譬如"-rwxrwxr-x"：

图 5.5.2 文件权限位

最前面的一个字符表示该文件的类型，这个前面给大家介绍过，" - "表示该文件是一个普通文件。

r 表示具有读权限；

w 表示具有写权限；

x 表示具有执行权限；

-表示无此权限。

当进程每次对文件进行读、写、执行等操作时，内核就会对文件进行访问权限检查，以确定该进程对文件是否拥有相应的权限。而文件的权限检查就涉及到了文件的所有者（st_uid）、文件所属组（st_gid）以及其它用户，当然这里指的是从文件的角度来看；而对于进程来说，参与文件权限检查的是进程的有效用户、有效用户组以及进程的附属组用户。

如何判断权限，首先要搞清楚该进程对于需要进行操作的文件来说是属于哪一类“角色”：

- 如果进程的有效用户 ID 等于文件所有者 ID（st_uid），意味着该进程以文件所有者的角色存在；
- 如果进程的有效用户 ID 并不等于文件所有者 ID，意味着该进程并不是文件所有者身份；但是进程的有效用户组 ID 或进程的附属组 ID 之一等于文件的组 ID（st_gid），那么意味着该进程以文件所属组成员的角色存在，也就是文件所属组的同组用户成员。
- 如果进程的有效用户 ID 不等于文件所有者 ID、并且进程的有效用户组 ID 或进程的所有附属组 ID 均不等于文件的组 ID（st_gid），那么意味着该进程以其它用户的角色存在。
- 如果进程的有效用户 ID 等于 0（root 用户），则无需进行权限检查，直接对该文件拥有最高权限。

确定了进程对于文件来说是属于哪一类“角色”之后，相应的权限就直接“对号入座”即可。接下来聊一聊文件的附加的特殊权限。

特殊权限

st_mode 字段中除了记录文件的 9 个普通权限之外，还记录了文件的 3 个特殊权限，也就是图 5.2.1 中所表示的 S 字段权限位，S 字段三个 bit 位中，从高位到低位依次表示文件的 set-user-ID 位权限、set-group-ID 位权限以及 sticky 位权限，如下所示：

特殊权限	含义
S_ISUID	set-user-ID 位权限
S_ISGID	set-group-ID 位权限
S_ISVTX	Sticky 位权限

表 5.5.2 文件的特殊权限位

这三种权限分别使用 S_ISUID、S_ISGID 和 S_ISVTX 三个宏来表示：

```
S_ISUID     04000     set-user-ID bit
S_ISGID     02000     set-group-ID bit (see below)
S_ISVTX     01000     sticky bit (see below)
```

同样，以上数字使用的是八进制方式表示。对应的 bit 位数字为 1，则表示设置了该权限、为 0 则表示并未设置该权限；譬如通过 st_mode 变量判断文件是否设置了 set-user-ID 位权限，代码如下：

```
if (st.st_mode & S_ISUID) {
    //设置了 set-user-ID 位权限
} else {
    //没有设置 set-user-ID 位权限
}
```

这三个权限位具体有什么作用呢？接下里给大家简单地介绍一下：

- 当进程对文件进行操作的时候、将进行权限检查，如果文件的 set-user-ID 位权限被设置，内核会将进程的有效 ID 设置为该文件的用户 ID（文件所有者 ID），意味着该进程直接获取了文件所有者的权限、以文件所有者的身份操作该文件。
- 当进程对文件进行操作的时候、将进行权限检查，如果文件的 set-group-ID 位权限被设置，内核会将进程的有效用户组 ID 设置为该文件的用户组 ID（文件所属组 ID），意味着该进程直接获取了文件所属组成员的权限、以文件所属组成员的身份操作该文件。

看到这里，大家可能就要问了，如果两个权限位同时被设置呢？关于这个问题，我们后面可以进行相应的测试，答案自然会揭晓！

当然，set-user-ID 位和 set-group-ID 位权限的作用并不如此简单，关于其它的功能本文档便不再叙述了，因为这些特殊权限位实际中用到的机会确实不多。除此之外，Sticky 位权限也不再给大家介绍了，笔者对此也不是很了解，有兴趣的读者可以自行查阅相关的书籍。

Linux 系统下绝大部分的文件都没有设置 set-user-ID 位权限和 set-group-ID 位权限，所以通常情况下，进程的有效用户等于实际用户（有效用户 ID 等于实际用户 ID），有效组等于实际组（有效组 ID 等于实际组 ID）。

5.5.2 目录权限

前面我们一直谈论的都是文件的读、写、执行权限，那对于创建文件、删除文件等这些操作难道就不需要相应的权限了吗？事实并不如此，譬如：有时删除文件或创建文件也会提示"权限不够"，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ touch hello.c
touch: 无法创建'hello.c': 权限不够
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf testApp.c
rm: 无法删除'testApp.c': 权限不够
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.3 创建文件、删除文件

那说明删除文件、创建文件这些操作也是需要相应权限的，那这些权限又是从哪里获取的呢？答案就是目录。目录（文件夹）在 Linux 系统下也是一种文件，拥有与普通文件相同的权限方案（S/U/G/O），只是这些权限的含义另有所指。

- 目录的读权限：可列出（譬如：通过 ls 命令）目录之下的内容（即目录下有哪些文件）。
- 目录的写权限：可以在目录下创建文件、删除文件。
- 目录的执行权限：可访问目录下的文件，譬如对目录下的文件进行读、写、执行等操作。

拥有对目录的读权限，用户只能查看目录中的文件列表，譬如使用 ls 命令进行查看：

```
dt@dt-virtual-machine:~/vscode_ws$ ls -l
总用量 8
drwxr-xr-x 2 dt dt 4096 1月    7 20:53 1_chapter
dr--r-xr-x 2 dt dt 4096 1月   20 15:13 2_chapter
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ ls 2_chapter/
ls: 无法访问'2_chapter/file1': 权限不够
ls: 无法访问'2_chapter/file3': 权限不够
ls: 无法访问'2_chapter/file2': 权限不够
file1  file2  file3
dt@dt-virtual-machine:~/vscode_ws$
```

图 5.5.4 只有读权限查看目录下的文件

通过"ls -l"命令可以查看到 2_chapter 目录对于文件所有者只有读权限，当前操作的用户正是该目录所有者 dt，之后通过"ls 2_chapter"命令查看该目录下的文件，确实获取到了该目录下的 3 个文件：file1、file2、file3，说明只有读权限时，可以查看到目录下有哪些文件、显示出文件的名称；但是会看到上面打印出了一些"权限不够"信息，这是因为 Ubuntu 发行版对 ls 命令做了别名处理，执行 ls 命令的时候携带了一些选项，而这些选项会访问文件的一些信息，所以导致出现"权限不够"问题，这也说明，只拥有读权限、是没法访问目录下的文件的；为了确保使用的是 ls 命令本身，执行时需要给出路径的完整路径/bin/ls：

```
dt@dt-virtual-machine:~/vscode_ws$ ls -l
总用量 8
drwxr-xr-x 2 dt dt 4096 1月    7 20:53 1_chapter
dr--r-xr-x 2 dt dt 4096 1月   20 15:13 2_chapter
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ /bin/ls 2_chapter/
file1  file2  file3
dt@dt-virtual-machine:~/vscode_ws$
```

图 5.5.5 使用/bin/ls 再次执行

要想访问目录下的文件，譬如查看文件的 inode 节点、大小、权限等信息，还需要对目录拥有执行权限。

反之，若拥有对目录的执行权限、而无读权限，只要知道目录内文件的名称，仍可对其进行访问，但不能列出目录下的内容（即目录下包含的其它文件的名称）。

要想在目录下创建文件或删除原有文件，需要同时拥有对该目录的执行和写权限。

所以由此可知，如果需要对文件进行读、写或执行等操作，不光是需要拥有该文件本身的读、写或执行权限，还需要拥有文件所在目录的执行权限。

5.5.3 检查文件权限 access

通过前面的介绍，大家应该知道了，文件的权限检查不单单只讨论文件本身的权限，还需要涉及到文件所在目录的权限，只有同时都满足了，才能通过操作系统的权限检查，进而才可以对文件进行相关操作；所以，程序当中对文件进行相关操作之前，需要先检查执行进程的用户是否对该文件拥有相应的权限。那如何检查呢？可以使用 access 系统调用，函数原型如下：

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

首先，使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

pathname：需要进行权限检查的文件路径。

mode：该参数可以取以下值：

- F_OK：检查文件是否存在

- R_OK：检查是否拥有读权限
- W_OK：检查是否拥有写权限
- X_OK：检查是否拥有执行权限

除了可以单独使用之外，还可以通过按位或运算符"|"组合在一起。

返回值：检查项通过则返回 0，表示拥有相应的权限并且文件存在；否则返回-1，如果多个检查项组合在一起，只要其中任何一项不通过都会返回-1。

测试

通过 access 函数检查文件是否存在，若存在、则继续检查执行进程的用户对该文件是否有读、写、执行权限。

示例代码 5.5.1 access 函数使用示例

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

#define        MY_FILE        "./test_file"

int main(void)
{
    int ret;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("%: file does not exist.\n", MY_FILE);
        exit(-1);
    }

    /* 检查权限 */
    ret = access(MY_FILE, R_OK);
    if (!ret)
        printf("Read permission: Yes\n");
    else
        printf("Read permission: NO\n");

    ret = access(MY_FILE, W_OK);
    if (!ret)
        printf("Write permission: Yes\n");
    else
        printf("Write permission: NO\n");

    ret = access(MY_FILE, X_OK);
    if (!ret)
        printf("Execution permission: Yes\n");
```

```
    else
        printf("Execution permission: NO\n");

    exit(0);
}
```

接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
-rw-rw-r-- 1 dt dt 666 1月  20 16:52 testApp.c
-rw-rw-r-- 1 dt dt 666 1月  20 16:49 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Read permission: Yes
Write permission: Yes
Execution permission: NO
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.6 access 测试结果

5.5.4 修改文件权限 chmod

在 Linux 系统下，可以使用 chmod 命令修改文件权限，该命令内部实现方法其实是调用了 chmod 函数，chmod 函数是一个系统调用，函数原型如下所示（可通过"man 2 chmod"命令查看）：

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

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

首先，使用该函数需要包含头文件<sys/stat.h>。

函数参数及返回值如下所示：

pathname：需要进行权限修改的文件路径，若该参数所指为符号链接，实际改变权限的文件是符号链接所指向的文件，而不是符号链接文件本身。

mode：该参数用于描述文件权限，与 open 函数的第三个参数一样，这里不再重述，可以直接使用八进制数据来描述，也可以使用相应的权限宏（单个或通过位或运算符"|"组合）。

返回值：成功返回 0；失败返回-1，并设置 errno。

文件权限对于文件来说是非常重要的属性，是不能随随便便被任何用户所修改的，要想更改文件权限，要么是超级用户（root）进程、要么进程有效用户 ID 与文件的用户 ID（文件所有者）相匹配。

fchmod 函数

该函数功能与 chmod 一样，参数略有不同。fchmod()与 chmod()的区别在于使用了文件描述符来代替文件路径，就像是 fstat 与 stat 的区别。函数原型如下所示：

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

int fchmod(int fd, mode_t mode);
```

使用了文件描述符 fd 代替了文件路径 pathname，其它功能都是一样的。

测试

示例代码 5.5.2 chmod 函数使用示例

```
#include <sys/stat.h>
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void)
{
    int ret;

    ret = chmod("./test_file", 0777);
    if (-1 == ret) {
        perror("chmod error");
        exit(-1);
    }

    exit(0);
}
```

上述代码中，通过调用 chmod 函数将当前目录下的 test_file 文件，其权限修改为 0777（八进制表示方式，也可以使用 S_IRUSR、S_IWUSR 等这些宏来表示），也就是文件所有者、文件所属组用户以及其它用户都拥有读、写、执行权限，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
-rw-rw-r-- 1 dt dt 195 1月  20 18:09 testApp.c
-rw-r--r-- 1 dt dt 195 1月  20 18:09 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l test_file
-rwxrwxrwx 1 dt dt 195 1月  20 18:09 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.7 chmod 函数测试结果

执行程序之前，test_file 文件的权限为 rw-r--r--（0644），程序执行完成之后，再次查看文件权限为 rwxrwxrwx（0777），修改成功！

5.5.5 umask 函数

在 Linux 下有一个 umask 命令，在 Ubuntu 系统下执行看看：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ umask
0002
dt@dt-virtual-machine:~$
```

图 5.5.8 运行 umask 命令

可以看到该命令打印出了"0002"，这数字表示什么意思呢？这就要从 umask 命令的作用说起了，umask 命令用于查看/设置权限掩码，权限掩码主要用于对新建文件的权限进行屏蔽。权限掩码的表示方式与文件权限的表示方式相同，但是需要去除特殊权限位，umask 不能对特殊权限位进行屏蔽。

当新建文件时，文件实际的权限并不等于我们所设置的权限，譬如：调用 open 函数新建文件时，文件实际的权限并不等于 mode 参数所描述的权限，而是通过如下关系得到实际权限：

```
mode & ~umask
```

譬如调用 open 函数新建文件时，mode 参数指定为 0777，假设 umask 为 0002，那么实际权限为：

```
0777 & (~0002) = 0775
```

前面给大家介绍 open 函数的 mode 参数时，并未向大家提及到 umask，所以这里重新向大家说明。

umask 权限掩码是进程的一种属性，用于指明该进程新建文件或目录时，应屏蔽哪些权限位。进程的 umask 通常继承至其父进程（关于父、子进程相关的内容将会在后面章节给大家介绍），譬如在 Ubuntu shell 终端下执行的应用程序，它的 umask 继承至该 shell 进程。

当然，Linux 系统提供了 umask 函数用于设置进程的权限掩码，该函数是一个系统调用，函数原型如下所示（可通过"man 2 umask"命令查看）：

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

mode_t umask(mode_t mask);
```

首先，使用该命令需要包含头文件<sys/types.h>和<sys/stat.h>。

函数参数和返回值含义如下：

mask：需要设置的权限掩码值，可以发现 make 参数的类型与 open 函数、chmod 函数中的 mode 参数对应的类型一样，所以其表示方式也是一样的，前面也给大家介绍了，既可以使用数字表示（譬如八进制数）也可以直接使用宏（S_IRUSR、S_IWUSR 等）。

返回值：返回设置之前的 umask 值，也就是旧的 umask。

测试

接下来我们编写一个测试代码，使用 umask()函数修改进程的 umask 权限掩码，测试代码如下所示：

示例代码 5.5.3 umask 函数使用示例

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

int main(void)
{
    mode_t old_mask;

    old_mask = umask(0003);
    printf("old mask: %04o\n", old_mask);

    exit(0);
}
```

上述代码中，使用 umask 函数将该进程的 umask 设置为 0003（八进制），返回得到的 old_mask 则是设置之前旧的 umask 值，然后将其打印出来：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ umask
0002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
old mask: 0002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.5.9 umask 函数测试结果

从打印信息可以看出，旧的 umask 等于 0002，这个 umask 是从当前 vscode 的 shell 终端继承下来的，如果没有修改进程的 umask 值，默认就是从父进程继承下来的 umask。

这里再次强调，umask 是进程自身的一种属性、A 进程的 umask 与 B 进程的 umask 无关（父子进程关系除外）。在 shell 终端下可以使用 umask 命令设置 shell 终端的 umask 值，但是该 shell 终端关闭之后、再次打开一个终端，新打开的终端将与之前关闭的终端并无任何瓜葛！

5.6 文件的时间属性

前面给大家介绍了 3 个文件的时间属性：文件最后被访问的时间、文件内容最后被修改的时间以及文件状态最后被改变的时间，分别记录在 struct stat 结构体的 st_atim、st_mtim 以及 st_ctim 变量中，如下所示：

字段	说明
st_atim	文件最后被访问的时间
st_mtim	文件内容最后被修改的时间
st_ctim	文件状态最后被改变的时间

表 5.6.1 与文件相关的 3 个时间属性

- 文件最后被访问的时间：访问指的是读取文件内容，文件内容最后一次被读取的时间，譬如使用 read()函数读取文件内容便会改变该时间属性；
- 文件内容最后被修改的时间：文件内容发生改变，譬如使用 write()函数写入数据到文件中便会改变该时间属性；
- 文件状态最后被改变的时间：状态更改指的是该文件的 inode 节点最后一次被修改的时间，譬如更改文件的访问权限、更改文件的用户 ID、用户组 ID、更改链接数等，但它们并没有更改文件的实际内容，也没有访问（读取）文件内容。为什么文件状态的更改指的是 inode 节点的更改呢？3.1 小节给大家介绍 inode 节点的时候给大家介绍过，inode 中包含了很多文件信息，譬如：文件字节大小、文件所有者、文件对应的读/写/执行权限、文件时间戳（时间属性）、文件数据存储的 block（块）等，所以由此可知，状态的更改指的就是 inode 节点内容的更改。譬如 chmod()、chown()等这些函数都能改变该时间属性。

表 5.6.2 列出了一些系统调用或 C 库函数对文件时间属性的影响，有些操作并不仅仅只会影响文件本身的时间属性，还会影响到其父目录的相关时间属性。

函数	文件			父目录			注释
	a	m	c	a	m	c	
chmod()			*				与 fchmod()相同
chown()			*				与 fchown()和 lchown()相同
exec()	*						
link()			*		*	*	影响第二个参数的父目录
mkdir()	*	*	*		*	*	
mkfifo()	*	*	*		*	*	
mknod()	*	*	*		*	*	
open()	*	*	*		*	*	新建文件时
read()	*						与 pread()相同
rename()			*		*	*	
rmdir()					*	*	与 remove()相同
unlink()			*		*	*	
utime()	*	*	*				与 utimes()、futimesat()相同
write()		*	*				与 pwrite()相同

表 5.6.2 不同函数对文件时间属性的影响

5.6.1 utime()、utimes()修改时间属性

文件的时间属性虽然会在我们对文件进行相关操作（譬如：读、写）的时候发生改变，但这些改变都是隐式、被动的发生改变，除此之外，还可以使用 Linux 系统提供的系统调用显式的修改文件的时间属性。本小节给大家介绍如何使用 utime()和 utimes()函数来修改文件的时间属性。

Tips：只能显式修改文件的最后一次访问时间和文件内容最后被修改的时间，不能显式修改文件状态最后被改变的时间，大家可以想一想为什么？笔者把这个作为思考题留给大家！

utime()函数

utime()函数原型如下所示：

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

int utime(const char *filename, const struct utimbuf *times);
```

首先，使用该函数需要包含头文件<sys/types.h>和<utime.h>。

函数参数和返回值含义如下：

filename：需要修改时间属性的文件路径。

times：将时间属性修改为该参数所指定的时间值，times 是一个 struct utimbuf 结构体类型的指针，稍后给大家介绍，如果将 times 参数设置为 NULL，则会将文件的访问时间和修改时间设置为系统当前时间。

返回值：成功返回值 0；失败将返回-1，并会设置 errno。

来看看 struct utimbuf 结构体：

示例代码 5.6.1 struct utimbuf 结构体

```
struct utimbuf {
    time_t actime;      /* 访问时间 */
    time_t modtime;     /* 内容修改时间 */
};
```

该结构体中包含了两个 time_t 类型的成员，分别用于表示访问时间和内容修改时间，time_t 类型其实就是 long int 类型，所以这两个时间是以秒为单位的，所以由此可知，utime()函数设置文件的时间属性精度只能到秒。

同样对于文件来说，时间属性也是文件非常重要的属性之一，对文件时间属性的修改也不是任何用户都可以随便修改的，只有以下两种进程可对其进行修改：

- 超级用户进程（以 root 身份运行的进程）。
- 有效用户 ID 与该文件用户 ID（文件所有者）相匹配的进程。
- 在参数 times 等于 NULL 的情况下，对文件拥有写权限的进程。

除以上三种情况之外的用户进程将无法对文件时间戳进行修改。

utime 测试

接下来我们编写一个简单地测试程序，使用 utime()函数修改文件的访问时间和内容修改时间，示例代码如下：

示例代码 5.6.2 utime 函数使用示例

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

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

#define MY_FILE         "./test_file"

int main(void)
{
    struct utimbuf utm_buf;
    time_t cur_sec;
    int ret;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 获取当前时间 */
    time(&cur_sec);
    utm_buf.actime = cur_sec;
    utm_buf.modtime = cur_sec;

    /* 修改文件时间戳 */
    ret = utime(MY_FILE, &utm_buf);
    if (-1 == ret) {
        perror("utime error");
        exit(-1);
    }

    exit(0);
}
```

上述代码尝试将 test_file 文件的访问时间和内容修改时间修改为当前系统时间。程序中使用到了 time()函数，time()是 Linux 系统调用，用于获取当前时间（也可以直接将 times 参数设置为 NULL，这样就不需要使用 time 函数来获取当前时间了），单位为秒，关于该函数在后面的章节内容中会给大家介绍，这里简单地了解一下。接下来编译测试，在运行程序之间，先使用 stat 命令查看 test_file 文件的时间戳，如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
  文件：'test_file'
  大小：8712            块：24         IO 块：4096   普通文件
设备：801h/2049d        Inode：3670377     硬链接：1
权限：(0664/-rw-rw-r--)  Uid：( 1000/      dt)   Gid：( 1000/      dt)
最近访问：2021-01-21 10:27:00.814290077 +0800
最近更改：2021-01-21 10:26:50.441103476 +0800
最近改动：2021-01-21 15:12:18.624478845 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.6.1 查看 test_file 文件的时间戳

接下来编译程序、运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ stat test_file
  文件：'test_file'
  大小：8712            块：24         IO 块：4096   普通文件
设备：801h/2049d        Inode：3670377     硬链接：1
权限：(0664/-rw-rw-r--)  Uid：( 1000/      dt)   Gid：( 1000/      dt)
最近访问：2021-01-21 15:23:05.000000000 +0800
最近更改：2021-01-21 15:23:05.000000000 +0800
最近改动：2021-01-21 15:23:05.787446834 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.6.2 运行测试程序

会发现执行完测试程序之后，test_file 文件的访问时间和内容修改时间均被更改为当前时间了（大家可以使用 date 命令查看当前系统时间），并且会发现状态更改时间也会修改为当前时间了，当然这个不是在程序中修改、而是内核帮它自动修改的，为什么会这样呢？如果大家理解了之前介绍的知识内容，完全可以理解这个问题，这里笔者不再重述！

utimes()函数

utimes()也是系统调用，功能与 utime()函数一致，只是参数、细节上有些许不同，utimes()与 utime()最大的区别在于前者可以以微秒级精度来指定时间值，其函数原型如下所示：

```
#include <sys/time.h>

int utimes(const char *filename, const struct timeval times[2]);
```

首先，使用该函数需要包含头文件<sys/time.h>。

函数参数和返回值含义如下：

filename：需要修改时间属性的文件路径。

times：将时间属性修改为该参数所指定的时间值，times 是一个 struct timeval 结构体类型的数组，数组共有两个元素，第一个元素用于指定访问时间，第二个元素用于指定内容修改时间，稍后给大家介绍，如果 times 参数为 NULL，则会将文件的访问时间和修改时间设置为当前时间。

返回值：成功返回 0；失败返回-1，并且会设置 errno。

来看看 struct timeval 结构体：

示例代码 5.6.3 struct timeval 结构体

```
struct timeval {
    long tv_sec;          /* 秒 */
```

```
    long tv_usec;           /* 微秒 */
};
```

该结构体包含了两个成员变量 tv_sec 和 tv_usec，分别用于表示秒和微秒。

utimes()遵循与 utime()相同的时间戳修改权限规则。

utimes 测试

示例代码 5.6.4 utimes 使用示例

```
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define MY_FILE         "./test_file"

int main(void)
{
    struct timeval tmval_arr[2];
    time_t cur_sec;
    int ret;
    int i;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 获取当前时间 */
    time(&cur_sec);
    for (i = 0; i < 2; i++) {

        tmval_arr[i].tv_sec = cur_sec;
        tmval_arr[i].tv_usec = 0;
    }

    /* 修改文件时间戳 */
    ret = utimes(MY_FILE, tmval_arr);
    if (-1 == ret) {
        perror("utimes error");
        exit(-1);
    }
```

```
    exit(0);
}
```

代码不再给大家进行介绍了，功能与示例代码 5.6.2 相同，大家可以自己动手编译、运行测试。

5.6.2 futimens()、utimensat()修改时间属性

除了上面给大家介绍了两个系统调用外，这里再向大家介绍两个系统调用，功能与 utime()和 utimes()函数功能一样，用于显式修改文件时间戳，它们是 futimens()和 utimensat()。

这两个系统调用相对于 utime 和 utimes 函数有以下三个优点：

- 可按纳秒级精度设置时间戳。相对于提供微秒级精度的 utimes()，这是重大改进！
- 可单独设置某一时间戳。譬如，只设置访问时间、而修改时间保持不变，如果要使用 utime()或 utimes()来实现此功能，则需要首先使用 stat()获取另一个时间戳的值，然后再将获取值与打算变更的时间戳一同指定。
- 可独立将任一时间戳设置为当前时间。使用 utime()或 utimes()函数虽然也可以通过将 times 参数设置为 NULL 来达到将时间戳设置为当前时间的效果，但是不能单独指定某一个时间戳，必须全部设置为当前时间（不考虑使用额外函数获取当前时间的方式，譬如 time()）。

futimens()函数

futimens 函数原型如下所示（可通过"man 2 utimensat"命令查看）：

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

int futimens(int fd, const struct timespec times[2]);
```

函数原型和返回值含义如下：

fd：文件描述符。

times：将时间属性修改为该参数所指定的时间值，times 指向拥有 2 个 struct timespec 结构体类型变量的数组，数组共有两个元素，第一个元素用于指定访问时间，第二个元素用于指定内容修改时间，该结构体在 5.2.3 小节给大家介绍过了，这里不再重述！

返回值：成功返回 0；失败将返回-1，并设置 errno。

所以由此可知，使用 futimens()设置文件时间戳，需要先打开文件获取到文件描述符。

该函数的时间戳可以按下列 4 种方式之一进行指定：

- 如果 times 参数是一个空指针，也就是 NULL，则表示将访问时间和修改时间都设置为当前时间。
- 如果 times 参数指向两个 struct timespec 结构体类型变量的数组，任一数组元素的 tv_nsec 字段的值设置为 UTIME_NOW，则表示相应的时间戳设置为当前时间，此时忽略相应的 tv_sec 字段。
- 如果 times 参数指向两个 struct timespec 结构体类型变量的数组，任一数组元素的 tv_nsec 字段的值设置为 UTIME_OMIT，则表示相应的时间戳保持不变，此时忽略 tv_sec 字段。
- 如果 times 参数指向两个 struct timespec 结构体类型变量的数组，且 tv_nsec 字段的值既不是 UTIME_NOW 也不是 UTIME_OMIT，在这种情况下，相应的时间戳设置为相应的 tv_sec 和 tv_nsec 字段指定的值。

Tips：UTIME_NOW 和 UTIME_OMIT 是两个宏定义。

使用 futimens()函数只有以下进程，可对文件时间戳进行修改：

- 超级用户进程。
- 在参数 times 等于 NULL 的情况下，对文件拥有写权限的进程。
- 有效用户 ID 与该文件用户 ID（文件所有者）相匹配的进程。

futimens()测试

示例代码 5.6.5 futimens 函数使用示例

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

#define MY_FILE         "./test_file"

int main(void)
{
    struct timespec tmsp_arr[2];
    int ret;
    int fd;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(MY_FILE, O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 修改文件时间戳 */
#if 1
    ret = futimens(fd, NULL);     //同时设置为当前时间
#endif

#if 0
    tmsp_arr[0].tv_nsec = UTIME_OMIT;//访问时间保持不变
    tmsp_arr[1].tv_nsec = UTIME_NOW;//内容修改时间设置为当期时间
    ret = futimens(fd, tmsp_arr);
#endif
```

```
#if 0
    tmsp_arr[0].tv_nsec = UTIME_NOW;//访问时间设置为当前时间
    tmsp_arr[1].tv_nsec = UTIME_OMIT;//内容修改时间保持不变
    ret = futimens(fd, tmsp_arr);
#endif

    if (-1 == ret) {
        perror("futimens error");
        goto err;
    }

err:
    close(fd);
    exit(ret);
}
```

代码不再给大家进行介绍，大家可以自己动手编译、运行测试。

utimensat()函数

utimensat()与 futimens()函数在功能上是一样的，同样可以实现纳秒级精度设置时间戳、单独设置某一时间戳、独立将任一时间戳设置为当前时间，与 futimens()在参数以及细节上存在一些差异，使用 futimens()函数，需要先将文件打开，通过文件描述符进行操作，utimensat()可以直接使用文件路径方式进行操作。utimensat 函数原型如下所示：

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

int utimensat(int dirfd, const char *pathname, const struct timespec times[2], int flags);
```

首先，使用该函数需要包含头文件<fcntl.h>和<sys/stat.h>。

函数参数和返回值含义如下：

dirfd：该参数可以是一个目录的文件描述符，也可以是特殊值 AT_FDCWD；如果 pathname 参数指定的是文件的绝对路径，则此参数会被忽略。

pathname：指定文件路径。如果 pathname 参数指定的是一个相对路径、并且 dirfd 参数不等于特殊值 AT_FDCWD，则实际操作的文件路径是相对于文件描述符 dirfd 指向的目录进行解析。如果 pathname 参数指定的是一个相对路径、并且 dirfd 参数等于特殊值 AT_FDCWD，则实际操作的文件路径是相对于调用进程的当前工作目录进行解析，关于进程的工作目录在 5.7 小节中有介绍。

times：与 futimens()的 times 参数含义相同。

flags：此参数可以为 0，也可以设置为 AT_SYMLINK_NOFOLLOW，如果设置为 AT_SYMLINK_NOFOLLOW，当 pathname 参数指定的文件是符号链接，则修改的是该符号链接的时间戳，而不是它所指向的文件。

返回值：成功返回 0；失败返回-1、并会设置时间戳。

utimensat()遵循与 futimens()相同的时间戳修改权限规则。

utimensat()函数测试

示例代码 5.6.6 utimensat 函数使用示例

```
#include <fcntl.h>
```

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

#define MY_FILE         "/home/dt/vscode_ws/2_chapter/test_file"

int main(void)
{
    struct timespec tmsp_arr[2];
    int ret;

    /* 检查文件是否存在 */
    ret = access(MY_FILE, F_OK);
    if (-1 == ret) {
        printf("Error: %s file does not exist!\n", MY_FILE);
        exit(-1);
    }

    /* 修改文件时间戳 */
#if 1
    ret = utimensat(-1, MY_FILE, NULL, AT_SYMLINK_NOFOLLOW);       //同时设置为当前时间
#endif

#if 0
    tmsp_arr[0].tv_nsec = UTIME_OMIT;//访问时间保持不变
    tmsp_arr[1].tv_nsec = UTIME_NOW;//内容修改时间设置为当期时间
    ret = utimensat(-1, MY_FILE, tmsp_arr, AT_SYMLINK_NOFOLLOW);
#endif

#if 0
    tmsp_arr[0].tv_nsec = UTIME_NOW;//访问时间设置为当前时间
    tmsp_arr[1].tv_nsec = UTIME_OMIT;//内容修改时间保持不变
    ret = utimensat(-1, MY_FILE, tmsp_arr, AT_SYMLINK_NOFOLLOW);
#endif

    if (-1 == ret) {
        perror("futimens error");
        exit(-1);
    }

    exit(0);
}
```

代码不再给大家进行介绍，大家可以自己动手编译、运行测试。

5.7 符号链接（软链接）与硬链接

在 Linux 系统中有两种链接文件，分为软链接（也叫符号链接）文件和硬链接文件，软链接文件也就是前面给大家的 Linux 系统下的七种文件类型之一，其作用类似于 Windows 下的快捷方式。那么硬链接文件又是什么呢？本小节就来聊一聊它们之间的区别。

首先，从使用角度来讲，两者没有任何区别，都与正常的文件访问方式一样，支持读、写以及执行。那它们的区别在哪呢？在底层原理上，为了说明这个问题，先来创建一个硬链接文件，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ln test_file hard1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ln test_file hard2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 36
3670377 -r-------- 3 dt dt 8712 1月  22 17:07 hard1
3670377 -r-------- 3 dt dt 8712 1月  22 17:07 hard2
3670377 -r-------- 3 dt dt 8712 1月  22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.1 创建硬链接文件

Tips：使用 ln 命令可以为一个文件创建软链接文件或硬链接文件，用法如下：

硬链接：ln 源文件 链接文件

软链接：ln -s 源文件 链接文件

关于该命令其它用法，可以查看 man 手册。

从图 5.7.1 中可知，使用 ln 命令创建的两个硬链接文件与源文件 test_file 都拥有相同的 inode 号，既然 inode 相同，也就意味着它们指向了物理硬盘的同一个区块，仅仅只是文件名字不同而已，创建出来的硬链接文件与源文件对文件系统来说是完全平等的关系。那么大家可能要问了，如果删除了硬链接文件或源文件其中之一，那文件所对应的 inode 以及文件内容在磁盘中的数据块会被文件系统回收吗？事实上并不会这样，因为 inode 数据结结构中会记录文件的链接数，这个链接数指的就是硬链接数，struct stat 结构体中的 st_nlink 成员变量就记录了文件的链接数，这些内容前面已经给大家介绍过了。

当为文件每创建一个硬链接，inode 节点上的链接数就会加一，每删除一个硬链接，inode 节点上的链接数就会减一，直到为 0，inode 节点和对应的数据块才会被文件系统所回收，也就意味着文件已经从文件系统中被删除了。从图 5.7.1 中可知，使用"ls -li"命令查看到，此时链接数为 3（dt 用户名前面的那个数字），我们明明创建了 2 个链接文件，为什么链接数会是 3？其实源文件 test_file 本身就是一个硬链接文件，所以这里才是 3。

当我们删除其中任何一个文件后，链接数就会减少，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
hard1  hard2  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf hard2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 24
3670377 -r-------- 2 dt dt 8712 1月   22 17:07 hard1
3670377 -r-------- 2 dt dt 8712 1月   22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf hard1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 12
3670377 -r-------- 1 dt dt 8712 1月   22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.2 删除链接文件

接下来再来聊一聊软链接文件，软链接文件与源文件有着不同的 inode 号，如图 5.7.3 所示，所以也就是意味着它们之间有着不同的数据块，但是软链接文件的数据块中存储的是源文件的路径名，链接文件可以通过这个路径找到被链接的源文件，它们之间类似于一种“主从”关系，当源文件被删除之后，软链接文件依然存在，但此时它指向的是一个无效的文件路径，这种链接文件被称为悬空链接，如图 5.7.4 所示。

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ln -s test_file soft1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ln -s test_file soft2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 12
3671798 lrwxrwxrwx 1 dt dt    9 1月   23 10:49 soft1 -> test_file
3671799 lrwxrwxrwx 1 dt dt    9 1月   23 10:49 soft2 -> test_file
3670377 -r-------- 1 dt dt 8712 1月   22 17:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.3 创建软链接

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ rm -rf test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -li
总用量 0
3671798 lrwxrwxrwx 1 dt dt 9 1月   23 10:49 soft1 -> test_file
3671799 lrwxrwxrwx 1 dt dt 9 1月   23 10:49 soft2 -> test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.4 删除源文件

从图中还可看出，inode 节点中记录的链接数并未将软链接计算在内。

介绍完它们之间的区别之后，大家可能觉得硬链接相对于软链接来说有较大的优势，其实并不是这样，对于硬链接来说，存在一些限制情况，如下：

- 不能对目录创建硬链接（超级用户可以创建，但必须在底层文件系统支持的情况下）。
- 硬链接通常要求链接文件和源文件位于同一文件系统中。

而软链接文件的使用并没有上述限制条件，优点如下所示：

- 可以对目录创建软链接；
- 可以跨越不同文件系统；
- 可以对不存在的文件创建软链接。

5.7.1 创建链接文件

在 Linux 系统下，可以使用系统调用创建硬链接文件或软链接文件，本小节向大家介绍如何通过这些系统调用创建链接文件。

创建硬链接 link()

link()系统调用用于创建硬链接文件，函数原型如下（可通过"man 2 link"命令查看）：

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

首先，使用该函数需要包含头文件<unistd.h>。

函数原型和返回值含义如下：

oldpath：用于指定被链接的源文件路径，应避免 oldpath 参数指定为软链接文件，为软链接文件创建硬链接没有意义，虽然并不会报错。

newpath：用于指定硬链接文件路径，如果 newpath 指定的文件路径已存在，则会产生错误。

返回值：成功返回 0；失败将返回-1，并且会设置 errno。

link 函数测试

接下来我们编写一个简单地程序，演示 link 函数如何使用：

示例代码 5.7.1 link 函数使用示例

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

int main(void)
{
    int ret;

    ret = link("./test_file", "./hard");
    if (-1 == ret) {
        perror("link error");
        exit(-1);
    }

    exit(0);
}
```

程序中通过 link 函数为当前目录下的 test_file 文件创建了一个硬链接 hard，编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
-rw-rw-r-- 1 dt dt 195 1月  23 11:43 testApp.c
-rw-rw-r-- 1 dt dt 195 1月  23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 24
-rw-rw-r-- 2 dt dt  195 1月  23 11:43 hard
-rwxrwxr-x 1 dt dt 8704 1月  23 11:46 testApp
-rw-rw-r-- 1 dt dt  195 1月  23 11:43 testApp.c
-rw-rw-r-- 2 dt dt  195 1月  23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.5 link 函数测试结果

创建软链接 symlink()

symlink()系统调用用于创建软链接文件，函数原型如下（可通过"man 2 symlink"命令查看）：

```
#include <unistd.h>

int symlink(const char *target, const char *linkpath);
```

首先，使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

target：用于指定被链接的源文件路径，target 参数指定的也可以是一个软链接文件。

linkpath：用于指定硬链接文件路径，如果 newpath 指定的文件路径已存在，则会产生错误。

返回值：成功返回 0；失败将返回-1，并会设置 errno。

创建软链接时，并不要求 target 参数指定的文件路径已经存在，如果文件不存在，那么创建的软链接将成为“悬空链接”。

symlink 函数测试

接下来我们编写一个简单地程序，演示 symlink 函数如何使用：

示例代码 5.7.2 symlink 函数使用示例

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

int main(void)
{
    int ret;

    ret = symlink("./test_file", "./soft");
    if (-1 == ret) {
        perror("symlink error");
        exit(-1);
    }

    exit(0);
}
```

程序中通过 symlink 函数为当前目录下的 test_file 文件创建了一个软链接 soft，编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 20
lrwxrwxrwx 1 dt dt   11 1月  23 12:06 soft -> ./test_file
-rwxrwxr-x 1 dt dt 8712 1月  23 12:06 testApp
-rw-rw-r-- 1 dt dt  201 1月  23 12:04 testApp.c
-rw-rw-r-- 1 dt dt  195 1月  23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.6 symlink 函数测试结果

5.7.2 读取软链接文件

前面给大家介绍到，软链接文件数据块中存储的是被链接文件的路径信息，那如何读取出软链接文件中存储的路径信息呢？大家认为使用 read 函数可以吗？答案是不可以，因为使用 read 函数之前，需要先 open 打开该文件得到文件描述符，但是调用 open 打开一个链接文件本身是不会成功的，因为打开的并不是链接文件本身、而是其指向的文件，所以不能使用 read 来读取，那怎么办呢？可以使用系统调用 readlink。

readlink 函数原型如下所示：

```
#include <unistd.h>

ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

函数参数和返回值含义如下：

pathname：需要读取的软链接文件路径。只能是软链接文件路径，不能是其它类型文件，否则调用函数将报错。

buf：用于存放路径信息的缓冲区。

bufsiz：读取大小，一般读取的大小需要大于链接文件数据块中存储的文件路径信息字节大小。

返回值：失败将返回-1，并会设置 errno；成功将返回读取到的字节数。

readlink 函数测试

接下来我们编写一个简单地程序，演示 readlink 函数如何使用：

示例代码 5.7.3 readlink 函数使用示例

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

int main(void)
{
    char buf[50];
    int ret;

    memset(buf, 0x0, sizeof(buf));
    ret = readlink("./soft", buf, sizeof(buf));
    if (-1 == ret) {
        perror("readlink error");
        exit(-1);
    }

    printf("%s\n", buf);
    exit(0);
}
```

使用 readlink 函数读取当前目录下的软链接文件 soft，并将读取到的信息打印出来，测试如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 8
lrwxrwxrwx 1 dt dt  11 1月  23 12:06 soft -> ./test_file
-rw-rw-r-- 1 dt dt 295 1月  23 12:36 testApp.c
-rw-rw-r-- 1 dt dt 195 1月  23 11:43 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
./test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.7.7 readlink 函数测试结果

5.8 目录

目录（文件夹）在 Linux 系统也是一种文件，是一种特殊文件，同样可以使用前面给大家介绍 open、read 等这些系统调用以及 C 库函数对其进行操作，但是目录作为一种特殊文件，并不适合使用前面介绍的文件 I/O 方式进行读写等操作。在 Linux 系统下，会有一些专门的系统调用或 C 库函数用于对文件夹进行操作，譬如：打开、创建文件夹、删除文件夹、读取文件夹以及遍历文件夹中的文件等，那么本小节将向大家介绍目录相关的知识内容。

5.8.1 目录存储形式

3.1 小节中给大家介绍了普通文件的管理形式或存储形式，本小节聊一聊目录这种特殊文件在文件系统中的存储形式，其实目录在文件系统中的存储方式与常规文件类似，常规文件包括了 inode 节点以及文件内容数据存储块（block），参考图 3.1.1 所示；但对于目录来说，其存储形式则是由 inode 节点和目录块所构成，目录块当中记录了有哪些文件组织在这个目录下，记录它们的文件名以及对应的 inode 编号。

其存储形式如下图所示：

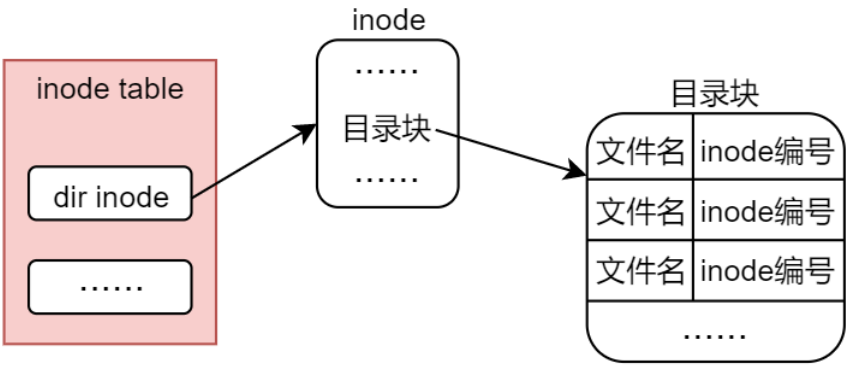


图 5.8.1 目录在文件系统中的存储形式

目录块当中有多个目录项（或叫目录条目），每一个目录项（或目录条目）都会对应到该目录下的某一个文件，目录项当中记录了该文件的文件名以及它的 inode 节点编号，所以通过目录的目录块便可以遍历找到该目录下的所有文件以及所对应的 inode 节点。

所以对此总结如下：

- 普通文件由 inode 节点和数据块构成
- 目录由 inode 节点和目录块构成

5.8.2 创建和删除目录

使用 open 函数可以创建一个普通文件，但不能用于创建目录文件，在 Linux 系统下，提供了专门用于创建目录 mkdir()以及删除目录 rmdir 相关的系统调用。

mkdir 函数

函数原型如下所示：

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

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

函数参数和返回值含义如下：

pathname：需要创建的目录路径。

mode：新建目录的权限设置，设置方式与 open 函数的 mode 参数一样，最终权限为（mode & ~umask）。

返回值：成功返回 0；失败将返回-1，并会设置 errno。

pathname 参数指定的新建目录的路径，该路径名可以是相对路径，也可以是绝对路径，若指定的路径名已经存在，则调用 mkdir()将会失败。

mode 参数指定了新目录的权限，目录拥有与普通文件相同的权限位，但是其表示的含义与普通文件却有不同，5.5.2 小计对此作了说明。

mkdir 函数测试

示例代码 5.8.1 mkdir 函数使用示例

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

int main(void)
{
    int ret;

    ret = mkdir("./new_dir", S_IRWXU |
                S_IRGRP | S_IXGRP |
                S_IROTH | S_IXOTH);
    if (-1 == ret) {
        perror("mkdir error");
        exit(-1);
    }

    exit(0);
}
```

上述代码中，我们通过 mkdir 函数在当前目录下创建了一个目录 new_dir，并将其权限设置为 0755（八进制），编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 20
drwxr-xr-x 2 dt dt 4096 1月  23 14:50 new_dir
-rwxrwxr-x 1 dt dt 8712 1月  23 14:50 testApp
-rw-rw-r-- 1 dt dt  267 1月  23 14:47 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.2 mkdir 创建目录

rmdir 函数

rmdir()用于删除一个目录

```
#include <unistd.h>

int rmdir(const char *pathname);
```

首先，使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

pathname：需要删除的目录对应的路径名，并且该目录必须是一个空目录，也就是该目录下只有.和..这两个目录项；pathname 指定的路径名不能是软链接文件，即使该链接文件指向了一个空目录。

返回值：成功返回 0；失败将返回-1，并会设置 errno。

rmdir 函数测试

示例代码 5.8.2 rmdir 函数使用示例

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

int main(void)
{
    int ret;

    ret = rmdir("./new_dir");
    if (-1 == ret) {
        perror("rmdir error");
        exit(-1);
    }

    exit(0);
}
```

5.8.3 打开、读取以及关闭目录

打开、读取、关闭一个普通文件可以使用 open()、read()、close()，而对于目录来说，可以使用 opendir()、readdir()和 closedir()来打开、读取以及关闭目录，接下来将向大家介绍这 3 个 C 库函数的用法。

打开文件 opendir

opendir()函数用于打开一个目录，并返回指向该目录的句柄，供后续操作使用。Opendir 是一个 C 库函数，opendir()函数原型如下所示：

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

DIR *opendir(const char *name);
```

函数参数和返回值含义如下：

name：指定需要打开的目录路径名，可以是绝对路径，也可以是相对路径。

返回值：成功将返回指向该目录的句柄，一个 DIR 指针（其实质是一个结构体指针），其作用类似于 open 函数返回的文件描述符 fd，后续对该目录的操作需要使用该 DIR 指针变量；若调用失败，则返回 NULL。

读取目录 readdir

readdir()用于读取目录，获取目录下所有文件的名称以及对应 inode 号。这里给大家介绍的 readdir()是一个 C 库函数（事实上 Linux 系统还提供了一个 readdir 系统调用），其函数原型如下所示：

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

首先，使用该函数需要包含头文件<dirent.h>。

函数参数和返回值含义如下：

dirp：目录句柄 DIR 指针。

返回值：返回一个指向 struct dirent 结构体的指针，该结构体表示 dirp 指向的目录流中的下一个目录条目。在到达目录流的末尾或发生错误时，它返回 NULL。

Tips：“流”是从自然界中抽象出来的一种概念，有点类似于自然界当中的水流，在文件操作中，文件内容数据类似池塘中存储的水，N 个字节数据被读取出来或将 N 个字节数据写入到文件中，这些数据就构成了字节流。

“流”这个概念是动态的，而不是静态的。编程当中提到这个概念，一般都是与 I/O 相关，所以也经常叫做 I/O 流；但对于目录这种特殊文件来说，这里将目录块中存储的数据称为目录流，存储了一个一个的目录项（目录条目）。

struct dirent 结构体内容如下所示：

示例代码 5.8.3 struct dirent 结构体

```
struct dirent {
    ino_t          d_ino;       /* inode 编号 */
    off_t          d_off;       /* not an offset; see NOTES */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported by all filesystem types */
    char           d_name[256]; /* 文件名 */
};
```

对于 struct dirent 结构体，我们只需要关注 d_ino 和 d_name 两个字段即可，分别记录了文件的 inode 编号和文件名，其余字段并不是所有系统都支持，所以也不再给大家介绍，这些字段一般也不会使用到。

每调用一次 readdir()，就会从 drip 所指向的目录流中读取下一条目录项（目录条目），并返回 struct dirent 结构体指针，指向经静态分配而得的 struct dirent 类型结构，每次调用 readdir()都会覆盖该结构。一旦遇到目录结尾或是出错，readdir()将返回 NULL，针对后一种情况，还会设置 errno 以示具体错误。那如何区别究竟是到了目录末尾还是出错了呢，可通过如下代码进行判断：

```
    error = 0;
```

```
    direntp = readdir(dirp);
    if (NULL == direntp) {
        if (0 != error) {
            /* 出现了错误 */
        } else {
            /* 已经到了目录末尾 */
        }
    }
```

使用 readdir()返回时并未对文件名进行排序，而是按照文件在目录中出现的天然次序（这取决于文件系统向目录添加文件时所遵循的次序，及其在删除文件后对目录列表中空隙的填补方式）。

当使用 opendir()打开目录时，目录流将指向了目录列表的头部（0），使用 readdir()读取一条目录条目之后，目录流将会向后移动、指向下一个目录条目。这其实跟 open()类似，当使用 open()打开文件的时候，文件位置偏移量默认指向了文件头部，当使用 read()或 write()进行读写时，文件偏移量会自动向后移动。

rewinddir 函数

rewinddir()是 C 库函数，可将目录流重置为目录起点，以便对 readdir()的下一次调用将从目录列表中的第一个文件开始。rewinddir 函数原型如下所示：

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

void rewinddir(DIR *dirp);
```

首先，使用该函数需要包含头文件<dirent.h>。

函数参数和返回值含义如下：

dirp：目录句柄。

返回值：无返回值。

关闭目录 closedir 函数

closedir()函数用于关闭处于打开状态的目录，同时释放它所使用的资源，其函数原型如下所示：

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

int closedir(DIR *dirp);
```

首先，使用该函数需要包含头文件<sys/types.h>和<dirent.h>。

函数参数和返回值含义如下：

dirp：目录句柄。

返回值：成功返回 0；失败将返回-1，并设置 errno。

练习

根据本小节所学知识内容，可以做一个简单地编程练习，打开一个目录、并将目录下的所有文件的名称以及其对应 inode 编号打印出来。示例代码如下所示：

示例代码 5.8.4 本节编程练习

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
```

```
#include <errno.h>

int main(void)
{
    struct dirent *dir;
    DIR *dirp;
    int ret = 0;

    /* 打开目录 */
    dirp = opendir("./my_dir");
    if (NULL == dirp) {
        perror("opendir error");
        exit(-1);
    }

    /* 循环读取目录流中的所有目录条目 */
    errno = 0;

    while (NULL != (dir = readdir(dirp)))
        printf("%s %ld\n", dir->d_name, dir->d_ino);
    if (0 != errno) {
        perror("readdir error");
        ret = -1;
        goto err;
    } else
        printf("End of directory!\n");

err:
    closedir(dirp);
    exit(ret);
}
```

使用 opendir()打开了当前目录下的 my_dir 目录，该目录下的文件如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
my_dir  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -lia my_dir/
总用量 8
3803212 drwxrwxr-x 2 dt dt 4096 1月  25 11:53 .
3702156 drwxr-xr-x 3 dt dt 4096 1月  25 11:58 ..
3803234 -rw-rw-r-- 1 dt dt    0 1月  25 11:53 file1
3803235 -rw-rw-r-- 1 dt dt    0 1月  25 11:53 file2
3803238 -rw-rw-r-- 1 dt dt    0 1月  25 11:53 file3
3803239 -rw-rw-r-- 1 dt dt    0 1月  25 11:53 file4
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.3 my_dir 目录下的文件列表

接下来编译、运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
my_dir  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
. 3803212
file4 3803239
file1 3803234
file3 3803238
file2 3803235
.. 3702156
End of directory!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.4 运行测试程序

由此可知，示例代码 5.8.4 能够将 my_dir 目录下的所有文件全部扫描出来，打印出它们的名字以及 inode 节点。

5.8.4 进程的当前工作目录

Linux 下的每一个进程都有自己的当前工作目录（current working directory），当前工作目录是该进程解析、搜索相对路径名的起点（不是以"/"斜杆开头的绝对路径）。譬如，代码中调用 open 函数打开文件时，传入的文件路径使用相对路径方式进行表示，那么该进程解析这个相对路径名时、会以进程的当前工作目录作为参考目录。

一般情况下，运行一个进程时、其父进程的当前工作目录将被该进程所继承，成为该进程的当前工作目录。可通过 getcwd 函数来获取进程的当前工作目录，如下所示：

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

这是一个系统调用，使用该函数之前，需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

buf：getcwd()将内含当前工作目录绝对路径的字符串存放在 buf 缓冲区中。

size：缓冲区的大小，分配的缓冲区大小必须要大于字符串长度，否则调用将会失败。

返回值：如果调用成功将返回指向 buf 的指针，失败将返回 NULL，并设置 errno。

Tips：*若传入的 buf 为 NULL，且 size 为 0，则 getcwd()内部会按需分配一个缓冲区，并将指向该缓冲区的指针作为函数的返回值，为了避免内存泄漏，调用者使用完之后必须调用 free()来释放这一缓冲区所占内存空间。*

测试

接下来，我们编写一个简单地测试程序用于读取进程的当前工作目录：

示例代码 5.8.5 getcwd 函数测试例程

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

int main(void)
{
    char buf[100];
    char *ptr;
```

```
    memset(buf, 0x0, sizeof(buf));

    ptr = getcwd(buf, sizeof(buf));
    if (NULL == ptr) {
        perror("getcwd error");
        exit(-1);
    }

    printf("Current working directory: %s\n", buf);
    exit(0);
}
```

编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ pwd
/home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Current working directory: /home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.5 测试结果

改变当前工作目录

系统调用 chdir()和 fchdir()可以用于更改进程的当前工作目录，函数原型如下所示：

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

首先，使用这两个函数之一需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

path：将进程的当前工作目录更改为 path 参数指定的目录，可以是绝对路径、也可以是相对路径，指定的目录必须要存在，否则会报错。

fd：将进程的当前工作目录更改为 fd 文件描述符所指定的目录（譬如使用 open 函数打开一个目录）。

返回值：成功均返回 0；失败均返回-1，并设置 errno。

此两函数的区别在于，指定目录的方式不同，chdir()是以路径的方式进行指定，而 fchdir()则是通过文件描述符，文件描述符可调用 open()打开相应的目录时获得。

测试

示例代码 5.8.6 chdir 函数使用示例

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

```
int main(void)
{
    char buf[100];
    char *ptr;
    int ret;

    /* 获取更改前的工作目录 */
    memset(buf, 0x0, sizeof(buf));
    ptr = getcwd(buf, sizeof(buf));
    if (NULL == ptr) {
        perror("getcwd error");
        exit(-1);
    }

    printf("Before the change:    %s\n", buf);

    /* 更改进程的当前工作目录 */
    ret = chdir("./new_dir");
    if (-1 == ret) {
        perror("chdir error");
        exit(-1);
    }

    /* 获取更改后的工作目录 */
    memset(buf, 0x0, sizeof(buf));
    ptr = getcwd(buf, sizeof(buf));
    if (NULL == ptr) {
        perror("getcwd error");
        exit(-1);
    }

    printf("After the change:     %s\n", buf);
    exit(0);
}
```

上述程序会在更改工作目录之前获取当前工作目录、并将其打印出来，之后调用 chdir 函数将进程的工作目录更改为当前目录下的 new_dir 目录，更改成功之后再将进程的当前工作目录获取并打印出来，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
new_dir  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Before the change:  /home/dt/vscode_ws/2_chapter
After the change:   /home/dt/vscode_ws/2_chapter/new_dir
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.8.6 编译运行

5.9 删除文件

前面给大家介绍了如何删除一个目录，使用 rmdir()函数即可，显然该函数并不能删除一个普通文件，那如何删除一个普通文件呢？方法就是通过系统调用 unlink()或使用 C 库函数 remove()。

使用 unlink 函数删除文件

unlink()用于删除一个文件（不包括目录），函数原型如下所示：

```
#include <unistd.h>

int unlink(const char *pathname);
```

使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

pathname：需要删除的文件路径，可使用相对路径、也可使用绝对路径，如果 pathname 参数指定的文件不存在，则调用 unlink()失败。

返回值：成功返回 0；失败将返回-1，并设置 errno。

前面给大家介绍 link 函数，用于创建一个硬链接文件，创建硬链接时，inode 节点上的链接数就会增加；unlink()的作用与 link()相反，unlink()系统调用用于移除/删除一个硬链接（从其父级目录下删除该目录条目）。

所以 unlink()系统调用实质上是移除 pathname 参数指定的文件路径对应的目录项（从其父级目录中移除该目录项），并将文件的 inode 链接计数将 1，如果该文件还有其它硬链接，则任可通过其它链接访问该文件的数据；只有当链接计数变为 0 时，该文件的内容才可被删除。另一个条件也会阻止删除文件的内容---只要有进程打开了该文件，其内容也不能被删除。关闭一个文件时，内核会检查打开该文件的进程个数，如果这个计数达到 0，内核再去检查其链接计数，如果链接计数也是 0，那么就删除该文件对应的内容（也就是文件对应的 inode 以及数据块被回收，如果一个文件存在多个硬链接，删除其中任何一个硬链接，其 inode 和数据块并没有被回收，还可通过其它硬链接访问文件的数据）。

unlink()系统调用并不会对软链接进行解引用操作，若 pathname 指定的文件为软链接文件，则删除软链接文件本身，而非软链接所指定的文件。

测试

示例代码 5.9.1 unlink 函数使用示例

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

int main(void)
{
    int ret;
```

```
    ret = unlink("./test_file");
    if (-1 == ret) {
        perror("unlink error");
        exit(-1);
    }

    exit(0);
}
```

上述代码调用 unlink()删除当前目录下的 test_file 文件，编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 5.9.1 unlink 删除文件

使用 remove 函数删除文件

remove()是一个 C 库函数，用于移除一个文件或空目录，其函数原型如下所示：

```
#include <stdio.h>

int remove(const char *pathname);
```

使用该函数需要包含 C 库函数头文件<stdio.h>。

函数参数和返回值含义如下：

pathname：需要删除的文件或目录路径，可以是相对路径、也可是决定路径。

返回值：成功返回 0；失败将返回-1，并设置 errno。

pathname 参数指定的是一个非目录文件，那么 remove()去调用 unlink()，如果 pathname 参数指定的是一个目录，那么 remove()去调用 rmdir()。

与 unlink()、rmdir()一样，remove()不对软链接进行解引用操作，若 pathname 参数指定的是一个软链接文件，则 remove()会删除链接文件本身、而非所指向的文件。

测试

示例代码 5.9.2 remove 函数使用示例

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

int main(void)
{
    int ret;

    ret = remove("./test_file");
    if (-1 == ret) {
```

```
        perror("remove error");
        exit(-1);
    }

    exit(0);
}
```

5.10 文件重命名

本小节给大家介绍 rename()系统调用，借助于 rename()既可以对文件进行重命名，又可以将文件移至同一文件系统中的另一个目录下，其函数原型如下所示：

```
#include <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

oldpath：原文件路径。

newpath：新文件路径。

返回值：成功返回 0；失败将返回-1，并设置 errno。

调用 rename()会将现有的一个路径名 oldpath 重命名为 newpath 参数所指定的路径名。rename()调用仅操作目录条目，而不移动文件数据（不改变文件 inode 编号、不移动文件数据块中存储的内容），重命名既不影响指向该文件的其它硬链接，也不影响已经打开该文件的进程（譬如，在重命名之前该文件已被其它进程打开了，而且还未被关闭）。

根据 oldpath、newpath 的不同，有以下不同的情况需要进行说明：

- 若 newpath 参数指定的文件或目录已经存在，则将其覆盖；
- 若 newpath 和 oldpath 指向同一个文件，则不发生变化（且调用成功）。
- rename()系统调用对其两个参数中的软链接均不进行解引用。如果 oldpath 是一个软链接，那么将重命名该软链接；如果 newpath 是一个软链接，则会将其移除、被覆盖。
- 如果 oldpath 指代文件，而非目录，那么就不能将 newpath 指定为一个目录的路径名。要想重命名一个文件到某一个目录下，newpath 必须包含新的文件名。
- 如果 oldpath 指代为一个目录，在这种情况下，newpath 要么不存在，要么必须指定为一个空目录。
- oldpath 和 newpath 所指代的文件必须位于同一文件系统。由前面的介绍，可以得出此结论！
- 不能对.（当前目录）和..（上一级目录）进行重命名。

测试

示例代码 5.10.1 rename 函数使用示例

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

int main(void)
{
    int ret;

    ret = rename("./test_file", "./new_file");
```

```
    if (-1 == ret) {
        perror("rename error");
        exit(-1);
    }

    exit(0);
}
```

将当前目录下的 test_file 文件重命名为 new_file，接下来编译测试：

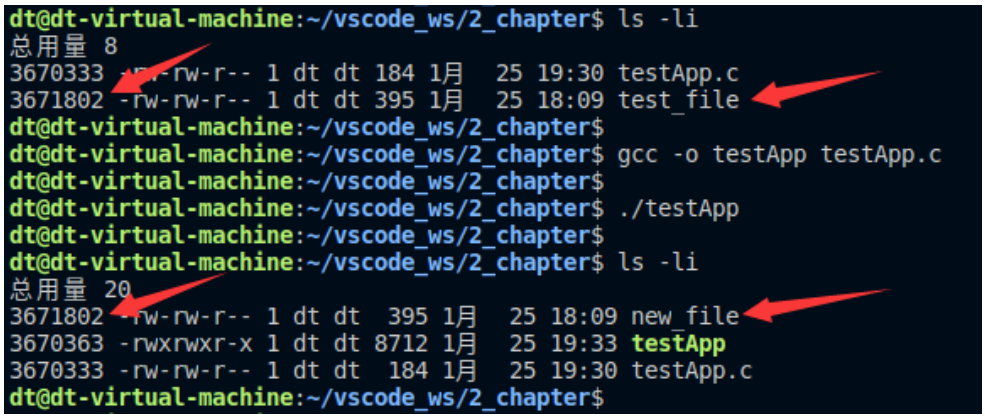


图 5.10.1 rename 重命名

从图中可以知道，使用 rename 进行文件重命名之后，其 inode 号并未改变。

5.11 总结

本章所介绍的内容比较多，主要是围绕文件属性以及目录展开的一系列相关话题，本章开头先给大家介绍 Linux 系统下的 7 种文件类型，包括普通文件、目录、设备文件（字符设备文件、块设备文件）、符号链接文件（软链接文件）、管道文件以及套接字文件。

接着围绕 stat 系统调用，详细给大家介绍了 struct stat 结构体中的每一个成员，这使得我们对 Linux 下文件的各个属性都有所了解。接着分别给大家详细介绍了文件属主、文件访问权限、文件时间戳、软链接与硬链接以及目录等相关内容，让大家知道在应用编程中如何去修改文件的这些属性以及它们所需要满足的条件。

至此，本章内容到这里就结束了，相信大家已经学习到了不少知识内容，大家加油！

第六章 字符串处理

字符串处理在几乎所有的编程语言中都是一个绕不开的话题，在一些高级语言当中，对字符串的处理支持度更是完善，譬如 C++、C#、Python 等。若在 C 语言中想要对字符串进行相关的处理，譬如将两个字符串进行拼接、字符串查找、两个字符串进行比较等操作，几乎是需要程序员自己编写字符串处理相关逻辑代码来实现字符串处理功能。

好在 C 语言库函数中已经给我们提供了丰富的字符串处理相关函数，基本常见的字符串处理需求都可以直接使用这些库函数来实现，而不需要自己编写代码，使用这些库函数可以大大减轻编程负担。这些库函数大致可以分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索等几类，本章将向大家介绍这些库函数的使用方法。

本章将会讨论如下主题内容。

- 字符串输入/输出；
- C 库中提供的字符串处理函数；
- 给应用程序传参；
- 正则表达式。

6.1 字符串输入/输出

在程序当中，经常需要在程序运行过程中打印出一些信息，将其输出显示到标准输出设备 stdout（譬如屏幕）或标准错误设备 stderr（譬如屏幕），譬如调试信息、报错信息、中间产生的变量的值等等，以实现对程序运行状态的掌控和分析。除了向 stdout 或 stderr 输出打印信息之外，有时程序在运行过程中还需要从标准输入设备 stdin（譬如键盘）中读取字符串，将读取到的字符串进行解析，以指导程序的下一步动作、控制程序执行流程。

6.1.1 字符串输出

常用的字符串输出函数有 putchar()、puts()、fputc()、fputs()，前面我们经常使用 printf()函数来输出字符串信息，而并没有使用到 putchar()、puts()、fputc()、fputs()这些函数，原因在于 printf()可以按照自己规定的格式输出字符串信息，一般称为格式化输出；而 putchar()、puts()、fputc()、fputs()这些函数只能输出字符串，不能进行格式转换。所以由此可知，printf()在功能上要比 putchar()、puts()、fputc()、fputs()这些函数更加强大，往往在实际编程中，printf()用的也会更多，但是 putchar()、puts()、fputc()、fputs()这些库函数相比与 printf，在使用上方便、简单。

与 printf()一样，putchar()、puts()、fputc()、fputs()这些函数也是标准 I/O 函数，属于标准 C 库函数，所以需要包含头文件<stdio.h>，并且它们也使用 stdio 缓冲。

puts 函数

puts()函数用来向标准输出设备（屏幕、显示器）输出字符串并自行换行。把字符串输出到标准输出设备，将' \0 '转换为换行符' \n '。puts 函数原型如下所示（可通过"man 3 puts"命令查看）：

```
#include <stdio.h>

int puts(const char *s);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

s：需要进行输出的字符串。

返回值：成功返回一个非负数；失败将返回 EOF，EOF 其实就是-1。

使用 puts()函数连换行符' \n '都省了，函数内部会自动在其后添加一个换行符。所以，如果只是单纯输出字符串到标准输出设备，而不包含数字格式化转换操作，那么使用 puts()会更加方便、简洁；puts()虽然方便、简单，但也仅限于输出字符串，功能还是没有 printf()强大。

puts 函数测试

示例代码 6.1.1 puts 函数使用示例

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

int main(void)
{
    char str[50] = "Linux app puts test";

    puts("Hello World!");
    puts(str);
    exit(0);
```

```
}
```

编译运行结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
Linux app puts test
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.1 puts 输出字符串

putchar 函数

putchar()函数可以把参数 c 指定的字符（一个无符号字符）输出到标准输出设备，其输出可以是一个字符，可以是介于 0~127 之间的一个十进制整型数（包含 0 和 127，输出其对应的 ASCII 码字符），也可以是用 char 类型定义好的一个字符型变量。putchar 函数原型如下所示（可通过"man 3 putchar"命令查看）：

```
#include <stdio.h>

int putchar(int c);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

c：需要进行输出的字符。

返回值：出错将返回 EOF。

putchar 函数测试

示例代码 6.1.2 putchar 函数使用示例

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

int main(void)
{
    putchar('A');
    putchar('B');
    putchar('C');
    putchar('D');
    putchar('\n');
    exit(0);
}
```

编译运行结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
ABCD
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.2 putchar 输出字符

fputc 函数

fputc()与 putchar()类似，也用于输出参数 c 指定的字符（一个无符号字符），与 putchar()区别在于，putchar()只能输出到标准输出设备，而 fputc()可把字符输出到指定的文件中，既可以是标准输出、标准错误设备，也可以是一个普通文件。

fputc 函数原型如下所示：

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

c：需要进行输出的字符。

stream：文件指针。

返回值：成功时返回输出的字符；出错将返回 EOF。

fputc 测试

(1)使用 fputc 函数将字符输出到标准输出设备。

示例代码 6.1.3 fputc 输出字符到标准输出设备

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

int main(void)
{
    fputc('A', stdout);
    fputc('B', stdout);
    fputc('C', stdout);
    fputc('D', stdout);
    fputc('\n', stdout);
    exit(0);
}
```

编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
ABCD
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.3 fputc 测试结果 1

(2)使用 fputc 函数将字符输出到一个普通文件。

示例代码 6.1.4 fputc 输出字符到普通文件

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

int main(void)
{
    FILE *fp = NULL;

    /* 创建一个文件 */
    fp = fopen("./new_file", "a");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    /* 输入字符到文件 */
    fputc('A', fp);
    fputc('B', fp);
    fputc('C', fp);
    fputc('D', fp);
    fputc('\n', fp);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

编译运行，结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
new_file  testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat new_file
ABCD
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.4 fputc 测试结果 2

fputs 函数

同理，fputs()与 puts()类似，也用于输出一条字符串，与 puts()区别在于，puts()只能输出到标准输出设备，而 fputs()可把字符串输出到指定的文件中，既可以是标准输出、标准错误设备，也可以是一个普通文件。

函数原型如下所示：

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
```

函数参数和返回值含义如下：

s：需要输出的字符串。

stream：文件指针。

返回值：成功返回非负数；失败将返回 EOF。

fputs 测试

(1)使用 fputs 输出字符串到标注输出设备。

示例代码 6.1.5 fputs 输出字符串到标准输出设备

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

int main(void)
{
    fputs("Hello World! 1\n", stdout);
    fputs("Hello World! 2\n", stdout);
    exit(0);
}
```

编译运行，结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World! 1
Hello World! 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.5 fputs 测试结果 1

(2)使用 fputs 输出字符串到一个普通文件。

示例代码 6.1.6 fputs 输出字符串到普通文件

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

int main(void)
{
    FILE *fp = NULL;

    /* 创建一个文件 */
    fp = fopen("./new_file", "a");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    fputs("Hello World! 1\n", fp);
    fputs("Hello World! 2\n", fp);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

编译运行，结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat new_file
Hello World! 1
Hello World! 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.6 fputs 测试结果 2

6.1.2 字符串输入

常用的字符串输入函数有 gets()、getchar()、fgetc()、fgets()。与 printf()对应，在 C 库函数中同样也提供了格式化输入函数 scanf()。scanf()与 gets()、getchar()、fgetc()、fgets()这些函数相比，在功能上确实有它的优势，但是在使用上不如它们方便、简单、更易于使用。

与 scanf()一样，gets()、getchar()、fgetc()、fgets()这些函数也是标准 I/O 函数，属于标准 C 库函数，所以需要包含头文件<stdio.h>，并且它们也使用 stdio 缓冲。

gets 函数

gets()函数用于从标准输入设备（譬如键盘）中获取用户输入的字符串，gets()函数原型如下所示：

```
#include <stdio.h>

char *gets(char *s);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

s：指向字符数组的指针，用于存储字符串。

返回值：如果成功，该函数返回指向 s 的指针；如果发生错误或者到达末尾时还未读取任何字符，则返回 NULL。

用户从键盘输入的字符串数据首先会存放在一个输入缓冲区中，gets()函数会从输入缓冲区中读取字符串存储到字符指针变量 s 所指向的内存空间，当从输入缓冲区中读走字符后，相应的字符便不存在于缓冲区了。

输入的字符串中就算是有空格也可以直接输入，字符串输入完成之后按回车即可，gets()函数不检查缓冲区溢出。

使用示例

使用 gets()函数获取用户输入字符串。

示例代码 6.1.7 gets 函数使用示例

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

int main(void)
{
    char str[100] = {0};
    char *ptr = NULL;

    ptr = gets(str);
    if (NULL == ptr)
        exit(-1);
    puts(str);

    exit(0);
}
```

当在 Ubuntu 系统编译代码时，会出现如下警告信息：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
testApp.c: In function ‘main’:
testApp.c:9:8: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
  ptr = gets(str);
        ^
testApp.c:9:6: warning: assignment makes pointer from integer without a cast [-Wint-conversion]
  ptr = gets(str);
      ^
/tmp/ccOspAbM.o: 在函数‘main’中：
testApp.c:(.text+0x49): 警告： the `gets' function is dangerous and should not be used.
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.7 编译代码出现警告信息

出现如上警告信息，其实是建议我们不要使用 gets()函数，因为程序中使用 gets()函数是非常不安全的，可能会出现 bug、出现不可靠性，gets()在某些意外情况下会导致程序陷入不可控状态，所以一般建议大家不要使用这个函数，可以使用后面将给大家介绍的 fgets()代替。

这里先不管这个警告信息，我们直接运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入字符串：aaa bbb ccc ddd
aaa bbb ccc ddd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入字符串：'adsada' "dasdasdas"
'adsada' "dasdasdas"
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.8 gets 测试结果

由此可知，不管我们输入的是空格、单引号、双引号都会作为 gets()获取到的字符串的一部分，直到用户输入回车换行符结束。

gets()与 scanf()的区别

gets()除了在功能上不及 scanf 之外，它们在一些细节上也存在着不同：

- gets()函数不仅比 scanf 简洁，而且，就算输入的字符串中有空格也可以，因为 gets()函数允许输入的字符串带有空格、制表符，输入的空格和制表符也是字符串的一部分，仅以回车换行符作为字符串的分割符；而对于 scanf 以%s 格式输入的时候，空格、换行符、TAB 制表符等都是作为字符串分割符存在，即分隔符前后是两个字符串，读取字符串时并不会将分隔符读取出来作为字符串的组成部分，一个%s 只能读取一个字符串，若要多去多个字符串，则需要使用多个%s、并且需要使用多个字符数组存储。
- gets()会将回车换行符从输入缓冲区中取出来，然后将其丢弃，所以使用 gets()读走缓冲区中的字符串数据之后，缓冲区中将不会遗留下回车换行符；而对于 scanf()来说，使用 scanf()读走缓冲区中的字符串数据时，并不会将分隔符（空格、TAB 制表符、回车换行符等）读走将其丢弃，所以使用 scanf()读走缓冲区中的字符串数据之后，缓冲区中依然还存在用户输入的分隔符。

针对上面所提出的两个区别点，下面我们将进行一些列的代码测试。

(1)测试 1

示例代码 6.1.8 测试代码 1

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

int main(void)
{
    char s1[100] = {0};
    char s2[100] = {0};

    scanf("%s", s1);
    printf("s1: %s\n", s1);

    scanf("%s", s2);
    printf("s2: %s\n", s2);

    exit(0);
}
```

当输入 123_456 回车时，输出结果如下（_表示空格）：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123 456
s1: 123
s2: 456
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.9 测试结果 1

代码中我们调用了两次 scanf()，而事实上我们只输入了一次，输入“123”之后输入空格、再输入“456”，然后按回车，由打印结果可知，字符串 s1 等于“123”，字符串 s2 等于“456”；当输入完成按回车之后，输入缓冲区中此时存在如下字符：

'1'、'2'、'3'、'空格'、'4'、'5'、'6'、'\n'

第一个 scanf()读取缓冲区时，将'1'、'2'、'3'读走，读走之后，它们将不存在于缓冲区中了，空格被视为字符串分割符，分割符及后面的字符将不会读取（以%s 格式输入情况下）。

第二个 scanf()读取缓冲区时，'4'、'5'、'6'会被读走，分割符依然不读取。

再次执行测试程序：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
123
s1: 123
456
s2: 456
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.10 测试结果 2

当输入“123”回车，之后输出了“123”；之后需要再次输入，接着输入“456”回车，输出“456”。这里输入了两次字符串，原因在于第一次 scanf()读走“123”之后，缓冲区中只剩下回车换行字符，第二次 scanf()不读取换行符，所以需要用户再次输入字符串。

(2)测试 2

示例代码 6.1.9 测试代码 2

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

int main(void)
{
    char s[100] = {0};
    char c;

    scanf("%s", s);
    printf("s: %s\n", s);

    scanf("%c", &c);
    printf("c: %d\n", c);

    exit(0);
}
```

执行测试：

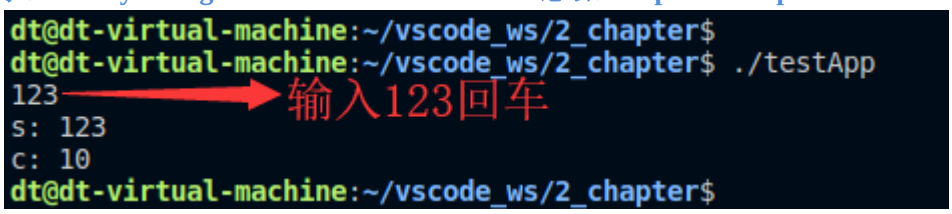


图 6.1.11 测试结果 3

同样这段代码也是调用了两次 scanf()，但只是输入了一次字符串，当第一个 scanf()读取之后，缓冲区中只剩下回车换行符；从打印信息可以发现，第二次 scanf()读取时，把换行符也读取出来了（换行符'\n'对应的 ASCII 编码值等于 10），因为这里 scanf 用的是%c 格式，而不是%s，对于%c 读入时，空格、换行符、TAB 这些都是正常字符。

(3)测试 3

示例代码 6.1.10 测试代码 3

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

int main(void)
{
    char s1[100] = {0};
    char s2[100] = {0};

    scanf("%s", s1);
    printf("s1: %s\n", s1);

    gets(s2);
    printf("s2: %s\n", s2);

    exit(0);
}
```

执行测试：

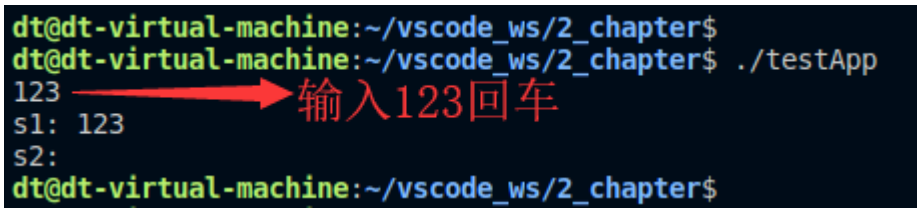


图 6.1.12 测试结果 4

这段代码先是调用了 scanf()，之后调用了 gets()，但只输入了一次字符串。scanf()读取之后，缓冲区中只剩下换行符，但是 gets()会将换行符读取出来并将其丢弃，所以说字符串便是一个空字符串。

再次执行测试程序：

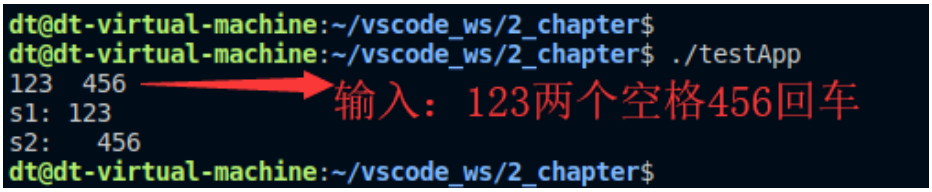


图 6.1.13 测试结果 5

字符串 s1 依然是“123”，scanf 读取完之后，缓冲区此时剩下如下字符串：

'空格'、'空格'、'4'、'5'、'6'、'\n'

gets()读取时将两个空格以及“456”、换行符全部读取出来，其中换行符会被丢弃、不作为字符串的组成字符，所以字符串 s2 前面就会存在两个空格。

getchar 函数

getchar()函数用于从标准输入设备中读取一个字符（一个无符号字符），函数原型如下所示：

```
#include <stdio.h>

int getchar(void);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

无需传参。

返回值：该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

同样 getchar()函数也是从输入缓冲区读取字符数据，但只读取一个字符，包括空格、TAB 制表符、换行回车符等。

测试

示例代码 6.1.11 getchar 函数使用示例

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

int main(void)
{
    int ch;

    ch = getchar();
    printf("ch: %c\n", ch);
    exit(0);
}
```

执行测试：

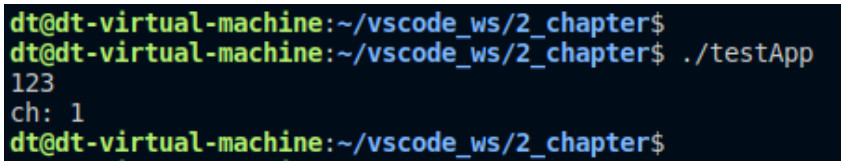


图 6.1.14 测试结果 6

getchar()只从输入缓冲区中读取一个字符，与 scanf 以%c 格式读取一样，空格、TAB 制表符、回车符都将是正常的字符。即使输入了多个字符，但 getchar()仅读取一个字符。

fgets 函数

fgets()与 gets()一样用于获取输入的字符串，fgets()函数原型如下所示

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

s：指向字符数组的指针，用于存储字符串。

size：这是要读取的最大字符数。

stream：文件指针。

fgets()与 gets()的区别主要是三点：

- gets()只能从标准输入设备中获取输入字符串，而 fgets()既可以从标准输入设备获取字符串、也可以从一个普通文件中获取输入字符串。
- fgets()可以设置获取字符串的最大字符数。
- gets()会将缓冲区中的换行符'\n'读取出来、将其丢弃、将'\n'替换为字符串结束符'\0'；fgets()也会将缓冲区中的换行符读取出来，但并不丢弃，而是作为字符串组成字符存在，读取完成之后自动在最后添加字符串结束字符'\0'。

其它方面与 gets()函数一样，包括前面给大家所介绍的与 scanf(%s)在一些细节方面的区别。

测试

示例代码 6.1.12 fgets 从标准输入设备获取字符串

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

int main(void)
{
    char str[100] = {0};

    printf("请输入字符串: ");
    fgets(str, sizeof(str), stdin);
    printf("%s", str);
    exit(0);
}
```

执行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
请输入字符串: 12345
12345
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.15 测试结果

此段代码中，使用 printf 打印字符串 str 时并没有在%s 后面添加'\n'，但是结果显示，打印出来的字符串已经换行，也就意味着 str 字符串本身就包含了换行符'\n'。

示例代码 6.1.13 fgets 从普通文件中输入字符串

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

int main(void)
{
    char str[100] = {0};
    FILE *fp = NULL;

    /* 打开文件 */
    fp = fopen("./test_file", "r");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    /* 从文件中输入字符串 */
    fgets(str, sizeof(str), fp);
    printf("%s", str);

    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

使用 fgets()读取文件中输入的字符串，文件指针会随着读取的字节数向前移动。

执行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.16 测试结果

fgetc 函数

fgetc()与 getchar()一样，用于读取一个输入字符，函数原型如下所示：

```
#include <stdio.h>

int fgetc(FILE *stream);
```

使用该函数需要包含头文件<stdio.h>。

函数参数和返回值含义如下：

stream：文件指针。

返回值：该函数以无符号 char 强制转换为 int 的形式返回读取的字符，如果到达文件末尾或发生读错误，则返回 EOF。

fgetc()与 getchar()的区别在于，fgetc 可以指定输入字符的文件，既可以从标准输入设备输入字符，也可以从一个普通文件中输入字符，其它方面与 getchar 函数相同。

测试

示例代码 6.1.14 fgetc 从标准输入设备中输入字符

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

int main(void)
{
    int ch;

    ch = fgetc(stdin);
    printf("%c\n", ch);
    exit(0);
}
```

执行测试：

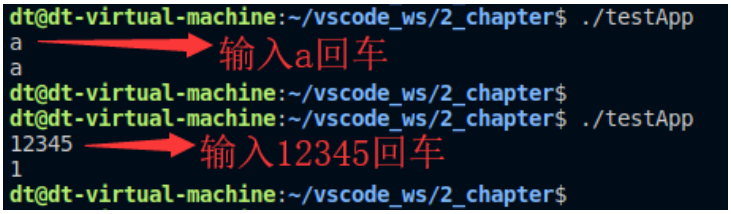


图 6.1.17 测试结果

示例代码 6.1.15 fgetc 从普通文件中输入字符

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

int main(void)
{
    int ch;
    FILE *fp = NULL;

    /* 打开文件 */
    fp = fopen("./test_file", "r");
    if (NULL == fp) {
        perror("fopen error");
        exit(-1);
    }

    /* 从文件中输入一个字符 */
    ch = fgetc(fp);
    printf("%c\n", ch);
```

```
    /* 关闭文件 */
    fclose(fp);
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test_file
Hello World
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
H
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.1.18 测试结果

6.1.3 总结

本小节给大家介绍了一些字符串输入、输出相关的 C 库函数，涉及到的函数比较多，在实际的编程当中，需要根据自己的实际需求以及函数的适用情况来进行选择。

6.2 字符串长度

C 语言函数库中提供了一个用于计算字符串长度的函数 strlen()，其函数原型如下所示：

```
#include <string.h>

size_t strlen(const char *s);
```

使用该函数需要包含头文件<string.h>。

函数参数和返回值含义如下：

s：需要进行长度计算的字符串，字符串必须包含结束字符'\0'。

返回值：返回字符串长度（以字节为单位），字符串结束字符'\0'不计算在内。

测试

示例代码 6.2.1 strlen 使用示例

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

int main(void)
{
    char str[] = "Linux app strlen test!";

    printf("String: \"%s\"\n", str);
    printf("Length: %ld\n", strlen(str));
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
String: "Linux app strlen test!"
Length: 22
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.2.1 strlen 计算字符串长度

sizeof 和 strlen 的区别

在程序当中，我们通常也会使用 sizeof 来计算长度，那 strlen 和 sizeof 有什么区别呢？

- sizeof 是 C 语言内置的操作符关键字，而 strlen 是 C 语言库函数；
- sizeof 仅用于计算数据类型的大小或者变量的大小，而 strlen 只能以结尾为'\0'的字符串作为参数；
- 编译器在编译时就计算出了 sizeof 的结果，而 strlen 必须在运行时才能计算出来；
- sizeof 计算数据类型或变量会占用内存的大小，strlen 计算字符串实际长度。

sizeof 和 strlen 测试

示例代码 6.2.2 strlen 和 sizeof 对比测试

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

int main(void)
{
    char str[50] = "Linux app strlen test!";
    char *ptr = str;

    printf("sizeof: %ld\n", sizeof(str));
    printf("strlen: %ld\n", strlen(str));
    puts("~~~~~~~~~~");
    printf("sizeof: %ld\n", sizeof(ptr));
    printf("strlen: %ld\n", strlen(ptr));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
sizeof: 50
strlen: 22
~~~~~~~~~~
sizeof: 8
strlen: 22
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.2.2 strlen 和 sizeof 对比测试结果

从打印信息可知，第一个 sizeof 计算的是数组变量 str 的大小，所以等于 50；而第二个 sizeof 计算的是指针变量 ptr 的大小，这里等于 8 个字节，因为这里笔者是在 Ubuntu 64 位系统下进行的测试，所以指针占用的内存大小就等于 8 个字节；而 strlen 始终计算的都是字符串的长度。

6.3 字符串拼接

C 语言函数库中提供了 strcat()函数或 strncat()函数用于将两个字符串连接（拼接）起来，strcat 函数原型如下所示：

```
#include <string.h>

char *strcat(char *dest, const char *src);
```

使用该函数需要包含头文件<string.h>。

函数参数和返回值含义如下：

dest：目标字符串。

src：源字符串。

返回值：返回指向目标字符串 dest 的指针。

strcat()函数会把 src 所指向的字符串追加到 dest 所指向的字符串末尾，所以必须要保证 dest 有足够的存储空间来容纳两个字符串，否则会导致溢出错误；dest 末尾的' \0 '结束字符会被覆盖，src 末尾的结束字符' \0 '会一起被复制过去，最终的字符串只有一个' \0 '。

strcat 测试

使用 strcat 函数将字符串 str2 连接到字符串 str1 末尾，并将其打印出来。

示例代码 6.3.1 strcat 使用示例

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

int main(void)
{
    char str1[100] = "Linux app strcat test, ";
    char str2[] = "Hello World!";

    strcat(str1, str2);
    puts(str1);
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Linux app strcat test, Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.3.1 strcat 测试结果

strncat 函数

strncat()与 strcat()的区别在于，strncat 可以指定源字符串追加到目标字符串的字符数量，strncat 函数原型如下所示：

```
#include <string.h>

char *strncat(char *dest, const char *src, size_t n);
```

函数参数和返回值含义如下：

dest：目标字符串。

src：源字符串。

n：要追加的最大字符数。

返回值：返回指向目标字符串 dest 的指针。

如果源字符串 src 包含 n 个或更多个字符，则 strncat()将 n+1 个字节追加到 dest 目标字符串（src 中的 n 个字符加上结束字符' \0 '）。

strncat 测试

示例代码 6.3.2 strncat 函数使用示例

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

int main(void)
{
    char str1[100] = "Linux app strcat test, ";
    char str2[] = "Hello World!";

    strncat(str1, str2, 5);
    puts(str1);
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Linux app strcat test, Hello
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.3.2 strncat 测试结果

6.4 字符串拷贝

C 语言函数库中提供了 strcpy()函数和 strncpy()函数用于实现字符串拷贝，strcpy 函数原型如下所示：

```
#include <string.h>

char *strcpy(char *dest, const char *src);
```

函数参数和返回值含义如下：

dest：目标字符串。

src：源字符串。

返回值：返回指向目标字符串 dest 的指针。

strcpy()会把 src（必须包含结束字符'\0'）指向的字符串复制（包括字符串结束字符'\0'）到 dest，所以必须保证 dest 指向的内存空间足够大，能够容纳下 src 字符串，否则会导致溢出错误。

strcpy 测试

示例代码 6.4.1 strcpy 函数使用示例

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

int main(void)
{
    char str1[100] = {0};
    char str2[] = "Hello World!";

    strcpy(str1, str2);
    puts(str1);
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.4.1 strcpy 测试结果

strncpy 函数

strncpy()与 strcpy()的区别在于，strncpy()可以指定从源字符串 src 复制到目标字符串 dest 的字符数量，strncpy 函数原型如下所示：

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

函数参数和返回值含义如下：

dest：目标字符串。

src：源字符串。

n：从 src 中复制的最大字符数。

返回值：返回指向目标字符串 dest 的指针。

把 src 所指向的字符串复制到 dest，最多复制 n 个字符。当 n 小于或等于 src 字符串长度（不包括结束字符的长度）时，则复制过去的字符串中没有包含结束字符'\0'；当 n 大于 src 字符串长度时，则会将 src 字

符串的结束字符' \0 '也一并拷贝过去，必须保证 dest 指向的内存空间足够大，能够容纳下拷贝过来的字符串，否则会导致溢出错误。

strncpy 函数测试

示例代码 6.4.2 strncpy 函数使用示例

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

int main(void)
{
    char str1[100] = "AAAAAAAAAAAAAAAAAAAAAAAA";
    char str2[] = "Hello World!";

    strncpy(str1, str2, 5);
    puts(str1);
    puts("~~~~~~~~~~~~~~~~");
    strncpy(str1, str2, 20);
    puts(str1);
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
HelloAAAAAAAAAAAAAAAAA
~~~~~~~~~~~~~~~~
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.4.2 strncpy 测试结果

memcpy、memmove、bcopy

除了 strcpy()和 strncpy()之外，其实还可以使用 memcpy()、memmove()以及 bcopy()这些库函数实现拷贝操作，字符串拷贝本质上也只是内存数据的拷贝，所以这些库函数同样也是适用的，在实际的编程当中，这些库函数也是很常用的，关于这三个库函数，这里不再给大家介绍，用法也非常简单，需要注意的就是目标内存空间与源内存空间是否有重叠的问题。

关于三个库函数的使用方法，大家可以使用 man 手册进行查询。

6.5 内存填充

在编程中，经常需要将某一块内存中的数据全部设置为指定的值，譬如在定义数组、结构体这种类型变量时，通常需要对其进行初始化操作，而初始化操作一般都是将其占用的内存空间全部填充为 0。

memset 函数

memset()函数用于将某一块内存的数据全部设置为指定的值，其函数原型如下所示：

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

使用该函数需要包含头文件<string.h>。

函数参数和返回值含义如下：

s：需要进行数据填充的内存空间起始地址。

c：要被设置的值，该值以 int 类型传递。

n：填充的字节数。

返回值：返回指向内存空间 s 的指针。

参数 c 虽然是以 int 类型传递，但 memset()函数在填充内存块时是使用该值的无符号字符形式，也就是函数内部会将该值转换为 unsigned char 类型的数据，以字节为单位进行数据填充。

memset 测试

对数组 str 进行初始化操作，将其存储的数据全部设置为 0。

示例代码 6.5.1 memset 函数使用示例

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

int main(void)
{
    char str[100];

    memset(str, 0x0, sizeof(str));
    exit(0);
}
```

bzero 函数

bzero()函数用于将一段内存空间中的数据全部设置为 0，函数原型如下所示：

```
#include <strings.h>

void bzero(void *s, size_t n);
```

函数参数和返回值含义如下：

s：内存空间的起始地址。

n：填充的字节数。

返回值：无返回值。

bzero 测试

对数组 str 进行初始化操作，将其存储的数据全部设置为 0。

示例代码 6.5.2 bzero 函数使用示例

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

```
int main(void)
{
    char str[100];

    bzero(str, sizeof(str));
    exit(0);
}
```

6.6 字符串比较

C 语言函数库提供了用于字符串比较的函数 strcmp()和 strncmp()，strcmp()函数原型如下所示：

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

函数参数和返回值含义如下：

s1：进行比较的字符串 1。

s2：进行比较的字符串 2。

返回值：

- 如果返回值小于 0，则表示 str1 小于 str2
- 如果返回值大于 0，则表示 str1 大于 str2
- 如果返回值等于 0，则表示字符串 str1 等于字符串 str2

strcmp 进行字符串比较，主要是通过比较字符串中的字符对应的 ASCII 码值，strcmp 会根据 ASCII 编码依次比较 str1 和 str2 的每一个字符，直到出现了不同的字符，或者某一字符串已经到达末尾（遇见了字符串结束字符'\0'）。

strcmp 测试

示例代码 6.6.1 strcmp 函数使用示例

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

int main(void)
{
    printf("%d\n", strcmp("ABC", "ABC"));
    printf("%d\n", strcmp("ABC", "a"));
    printf("%d\n", strcmp("a", "ABC"));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0
-1
1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.6.1 strcmp 测试结果

strncmp 函数

strncmp()与 strcmp()函数一样，也用于对字符串进行比较操作，但最多比较前 n 个字符，strncmp()函数原型如下所示：

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

函数参数和返回值含义如下：

s1：参与比较的第一个字符串。

s2：参与比较的第二个字符串。

n：最多比较前 n 个字符。

返回值：返回值含义与 strcmp()函数相同。

strncmp 测试

示例代码 6.6.2 strncmp 函数使用示例

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

int main(void)
{
    printf("%d\n", strncmp("ABC", "ABC", 3));
    printf("%d\n", strncmp("ABC", "ABCD", 3));
    printf("%d\n", strncmp("ABC", "ABCD", 4));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0
0
-1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.6.2 strncmp 测试结果

6.7 字符串查找

字符串查找在平时的编程当中也是一种很常见的操作，譬如从一个给定的字符串当中查找某一个字符或者一个字符串，并获取它的位置。C 语言函数库中也提供了一些用于字符串查找的函数，包括 strchr()、strrchr()、strstr()、strpbrk()、index()以及 rindex()等。

strchr 函数

使用 strchr()函数可以查找到给定字符串当中的某一个字符，函数原型如下所示：

```
#include <string.h>

char *strchr(const char *s, int c);
```

函数参数和返回值含义如下：

s：给定的目标字符串。

c：需要查找的字符。

返回值：返回字符 c 第一次在字符串 s 中出现的位置，如果未找到字符 c，则返回 NULL。

字符串结束字符' \0 '也将作为字符串的一部分，因此，如果将参数 c 指定为' \0 '，则函数将返回指向结束字符的指针。strchr 函数在字符串 s 中从前到后（或者称为从左到右）查找字符 c，找到字符 c 第一次出现的位置就返回，返回值指向这个位置，如果找不到字符 c 就返回 NULL。

strchr 测试

从字符串中查找一个字符，并将其在字符串数组中的下标打印出来。

示例代码 6.7.1 strchr 函数使用示例

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

int main(void)
{
    char *ptr = NULL;
    char str[] = "Hello World!";

    ptr = strchr(str, 'W');
    if (NULL != ptr) {
        printf("Character: %c\n", *ptr);
        printf("Offset: %ld\n", ptr - str);
    }

    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Character: W
Offset: 6
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.7.1 strchr 测试结果

strrchr 函数

strrchr()与 strchr()函数一样，它同样表示在字符串中查找某一个字符，返回字符第一次在字符串中出现的位置，如果没找到该字符，则返回值 NULL，但两者唯一不同的是，strrchr()函数在字符串中是从后到前（或者称为从右向左）查找字符，找到字符第一次出现的位置就返回，返回值指向这个位置，strrchr()函数原型如下所示：

```
#include <string.h>

char *strrchr(const char *s, int c);
```

函数参数和返回值含义与 strchr()函数相同。

strrchr 测试

编写程序测试 strrchr()与 strchr()之间的区别。

示例代码 6.7.2 strrchr()与 strchr()之间的区别

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

int main(void)
{
    char *ptr = NULL;
    char str[] = "I love my home";

    ptr = strchr(str, 'o');
    if (NULL != ptr)
        printf("strchr: %ld\n", ptr - str);

    ptr = strrchr(str, 'o');
    if (NULL != ptr)
        printf("strrchr: %ld\n", ptr - str);

    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strchr: 3
strrchr: 11
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.7.2 strrchr 与 strchr 对比测试结果

strstr 函数

与 strchr()函数不同的是，strstr()可在给定的字符串 haystack 中查找第一次出现子字符串 needle 的位置，不包含结束字符'\0'，函数原型如下所示：

```
#include <string.h>

char *strstr(const char *haystack, const char *needle);
```

函数参数和返回值含义如下：

haystack：目标字符串。

needle：需要查找的子字符串。

返回值：如果目标字符串 haystack 中包含了子字符串 needle，则返回该字符串首次出现的位置；如果未能找到子字符串 needle，则返回 NULL。

strstr 测试

在给定的字符串"I love my home"中，查找"home"在该字符串中首次出现的位置。

示例代码 6.7.3 strstr 函数使用示例

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

int main(void)
{
    char *ptr = NULL;
    char str[] = "I love my home";

    ptr = strstr(str, "home");
    if (NULL != ptr) {
        printf("String: %s\n", ptr);
        printf("Offset: %ld\n", ptr - str);
    }

    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
String: home
Offset: 10
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.7.3 strstr 测试结果

其它函数

除了上面介绍的三个函数之外，C 函数库中还提供其它的字符串（或字符）查找函数，譬如 strpbrk()、index()以及 rindex()等，这里便不再给大家一一介绍了，这些函数的用法都比较简单，大家通过 man 手册便可以快速了解到它们的使用方法。

6.8 字符串与数字互转

在编程中，经常会需要将数字组成的字符串转换为相应的数字、或者将数字转换为字符串，在 C 函数库中同样也提供了相应的函数，本小节就向大家介绍这些函数的用法。

6.8.1 字符串转整形数据

C 函数库中提供了一系列函数用于实现将一个字符串转为整形数据，主要包括 atoi()、atol()、atoll()以及 strtol()、strtoll()、strtoul()、strtoull()等，它们之间的区别主要包括以下两个方面：

- 数据类型（int、long int、unsigned long 等）。
- 不同进制方式表示的数字字符串（八进制、十六进制、十进制）。

atoi、atol、atoll 函数

atoi()、atol()、atoll()三个函数可用于将字符串分别转换为 int、long int 以及 long long 类型的数据，它们的函数原型如下：

```
#include <stdlib.h>

int atoi(const char *nptr);
long atol(const char *nptr);
long long atoll(const char *nptr);
```

使用这些函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

nptr：需要进行转换的字符串。

返回值：分别返回转换之后得到的 int 类型数据、long int 类型数据以及 long long 类型数据。

目标字符串 nptr 中可以包含非数字字符，转换时跳过前面的空格字符（如果目标字符串开头存在空格字符），直到遇上数字字符或正负符号才开始做转换，而再遇到非数字或字符串结束时(' /0 ')才结束转换，并将结果返回。

使用 atoi()、atol()、atoll()函数只能转换十进制表示的数字字符串，即 0~9。

测试

使用 atoi()、atol()、atoll()这三个函数将一个数字字符串转为十进制数据。

示例代码 6.8.1 atoi、atol、atoll 使用示例

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

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

int main(void)
{
    printf("atoi:   %d\n", atoi("500"));
    printf("atol:   %ld\n", atol("500"));
    printf("atoll:  %lld\n", atoll("500"));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
atoi:   500
atol:   500
atoll:  500
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.1 atoi、atol、atoll 测试结果

strtol、strtoll 函数

strtol()、strtoll()两个函数可分别将字符串转为 long int 类型数据和 long long ing 类型数据，与 atol()、atoll()之间的区别在于，strtol()、strtoll()可以实现将多种不同进制数（譬如二进制表示的数字字符串、八进制表示的数字字符串、十六进制表示的数数字符串）表示的字符串转换为整形数据，其函数原型如下所示：

```
#include <stdlib.h>

long int strtol(const char *nptr, char **endptr, int base);
long long int strtoll(const char *nptr, char **endptr, int base);
```

使用这两个函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

nptr：需要进行转换的目标字符串。

endptr：char **类型的指针，如果 endptr 不为 NULL，则 strtol()或 strtoll()会将字符串中第一个无效字符的地址存储在*endptr 中。如果根本没有数字，strtol()或 strtoll()会将 nptr 的原始值存储在*endptr 中（并返回 0）。也可将参数 endptr 设置为 NULL，表示不接收相应信息。

base：数字基数，参数 base 必须介于 2 和 36（包含）之间，或者是特殊值 0。参数 base 决定了字符串转换为整数时合法字符的取值范围，譬如，当 base=2 时，合法字符为' 0 '、' 1 '（表示是一个二进制表示的数字字符串）；当 base=8 时，合法字符为' 0 '、' 1 '、' 2 '、' 3 '……' 7 '（表示是一个八进制表示的数字字符串）；当 base=16 时，合法字符为' 0 ' 、' 1 '、' 2 '、' 3 '……' 9 '、' a '……' f '（表示是一个十六进制表示的数字字符串）；当 base 大于 10 的时候，' a '代表 10、' b '代表 11、' c '代表 12，依次类推，' z '代表 35（不区分大小写）。

返回值：分别返回转换之后得到的 long int 类型数据以及 long long int 类型数据。

需要进行转换的目标字符串可以以任意数量的空格或者 0 开头，转换时跳过前面的空格字符，直到遇上数字字符或正负符号（'+'或'-'）才开始做转换，而再遇到非数字或字符串结束时(' /0 ')才结束转换，并将结果返回。

在 base=0 的情况下，如果字符串包含一个了“0x”前缀，表示该数字将以 16 为基数；如果包含的是“0”前缀，表示该数字将以 8 为基数。

当 base=16 时，字符串可以使用“0x”前缀。

测试

示例代码 6.8.2 strtol 函数使用示例

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

int main(void)
{
    printf("strtol: %ld\n", strtol("0x500", NULL, 16));
    printf("strtol: %ld\n", strtol("0x500", NULL, 0));
    printf("strtol: %ld\n", strtol("500", NULL, 16));
    printf("strtol: %ld\n", strtol("0777", NULL, 8));
    printf("strtol: %ld\n", strtol("0777", NULL, 0));
    printf("strtol: %ld\n", strtol("1111", NULL, 2));
    printf("strtol: %ld\n", strtol("-1111", NULL, 2));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strtol: 1280
strtol: 1280
strtol: 1280
strtol: 511
strtol: 511
strtol: 15
strtol: -15
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.2 strtol 测试结果

strtoul、strtoull 函数

这两个函数使用方法与 strtol()、strtoll()一样，区别在于返回值的类型不同，strtoul()返回值类型是 unsigned long int，strtoull()返回值类型是 unsigned long long int，函数原型如下所示：

```
#include <stdlib.h>

unsigned long int strtoul(const char *nptr, char **endptr, int base);
unsigned long long int strtoull(const char *nptr, char **endptr, int base);
```

函数参数与 strtol()、strtoll()一样，这里不再重述！

测试

示例代码 6.8.3 strtoul 函数使用示例

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

```
#include <string.h>

int main(void)
{
    printf("strtoul:    %lu\n", strtoul("0x500", NULL, 16));
    printf("strtoul:    %lu\n", strtoul("0x500", NULL, 0));
    printf("strtoul:    %lu\n", strtoul("500", NULL, 16));
    printf("strtoul:    %lu\n", strtoul("0777", NULL, 8));
    printf("strtoul:    %lu\n", strtoul("0777", NULL, 0));
    printf("strtoul:    %lu\n", strtoul("1111", NULL, 2));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strtoul:        1280
strtoul:        1280
strtoul:        1280
strtoul:        511
strtoul:        511
strtoul:        15
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.3 strtoul 函数测试结果

6.8.2 字符串转浮点型数据

C 函数库中用于字符串转浮点型数据的函数有 atof()、strtod()、strtof()、strtold()。

atof 函数

atof()用于将字符串转换为一个 double 类型的浮点数据，函数原型如下所示：

```
#include <stdlib.h>

double atof(const char *nptr);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

nptr：需要进行转换的字符串。

返回值：返回转换得到的 double 类型数据。

测试

示例代码 6.8.4 atof 函数使用示例

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

int main(void)
{
```

```
    printf("atof:    %lf\n", atof("0.123"));
    printf("atof:    %lf\n", atof("-1.1185"));
    printf("atof:    %lf\n", atof("100.0123"));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
atof:    0.123000
atof:    -1.118500
atof:    100.012300
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.4 atof 函数测试结果

strtod、strtof、strtold 函数

strtof()、strtod()以及 strtold()三个库函数可分别将字符串转换为 float 类型数据、double 类型数据、long double 类型数据，函数原型如下所示：

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

使用这些函数需要包含头文件<stdlib.h>。

函数参数与 strtol()含义相同，但是少了 base 参数。

测试

示例代码 6.8.5 strtof、strtod、strtold 使用示例

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

int main(void)
{
    printf("strtof:    %f\n", strtof("0.123", NULL));
    printf("strtod:    %lf\n", strtod("-1.1185", NULL));
    printf("strtold:   %Lf\n", strtold("100.0123", NULL));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
strtof:         0.123000
strtod:         -1.118500
strtold:        100.012300
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.5 字符串转浮点型数据

6.8.3 数字转字符串

数字转换为字符串推荐大家使用前面介绍的格式化 IO 相关库函数，譬如使用 printf()将数字转字符串、并将其输出到标准输出设备或者使用 sprintf()或 snprintf()将数字转换为字符串并存储在缓冲区中，具体的使用方法，3.11 内容中已经给大家进行了详细介绍，这里不再重述。

示例代码 6.8.6 sprintf 数字转字符串

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

int main(void)
{
    char str[20] = {0};

    sprintf(str, "%d", 500);
    puts(str);

    memset(str, 0x0, sizeof(str));
    sprintf(str, "%f", 500.111);
    puts(str);

    memset(str, 0x0, sizeof(str));
    sprintf(str, "%u", 500);
    puts(str);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
500
500.111000
500
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.8.6 测试结果

6.9 给应用程序传参

一个能够接受外部传参的应用程序往往使用上会比较灵活，根据参入不同的参数实现不同的功能，前面给大家编写的示例代码中，信息都是硬编码在代码中的，譬如 open 打开的文件路径是固定的，意味着如果需要打开另一个文件则需要修改代码、修改文件路径，然后再重新编译、运行，非常麻烦、不够灵活。其实可以将这些可变的信息通过参数形式传递给应用程序，譬如，当执行应用程序的时候，把需要打开的文件路径作为参数传递给应用程序，就可以在不重新编译源码的情况下，通过传递不同的参数打开不同的文件。当然这里只是举个例子，不同应用程序需根据其需要来设计。

在第一章内容中便给大家介绍了 main 函数的两种常用写法，如果在执行应用程序时，需要向应用程序传递参数，则写法如下：

```
int main(int argc, char **argv)
{
    /* 代码 */
}
```

或者写成如下形式：

```
int main(int argc, char *argv[])
{
    /* 代码 */
}
```

传递进来的参数以字符串的形式存在，字符串的起始地址存储在 argv 数组中，参数 argc 表示传递进来的参数个数，包括应用程序自身路径名，多个不同的参数之间使用空格分隔开来，如果参数本身带有空格、则可以使用双引号" "或者单引号' '的形式来表示。

测试

获取执行应用程序时，向应用程序传递的参数。

示例代码 6.9.1 打印传递给应用程序的参数

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

int main(int argc, char *argv[])
{
    int i = 0;

    printf("Number of parameters: %d\n", argc);
    for (i = 0; i < argc; i++)
        printf("    %s\n", argv[i]);

    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 0 1 2
Number of parameters: 4
  ./testApp
  0
  1
  2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp a b c
Number of parameters: 4
  ./testApp
  a
  b
  c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 6.9.1 给应用程序传参

6.10 正则表达式

上面给大家介绍了 C 语言函数库中提供的用于处理字符串相关的一些函数，这些库函数能够满足基本常见的字符串处理需求，其库函数内部实现并不复杂，无非使用到了 for 循环进行处理，大家可以尝试自己去实现这些函数的功能。

本小节给大家介绍一个新的内容---正则表达式，在许多的应用程序当中，通常会有这样的需要：给定一个字符串，检查该字符串是否符合某种条件或规则、或者从给定的字符串中找出符合某种条件或规则的子字符串，将匹配到的字符串提取出来。这种需要在很多的应用程序当中是存在的，例如，很多应用程序都有这种校验功能，譬如检验用户输入的账号或密码是否符合它们定义的规则，如果不符合规则通常会提示用户按照正确的规则输入用户名或密码。

譬如给定一个字符串，在程序当中判断该字符串是否是一个 IP 地址，对于实现这个功能，大家可能首先想到的是，使用万能的 for 循环，当然，笔者首先肯定的是，使用 for 循环自然是可以解决这个问题，但是在程序代码处理上会比较麻烦，有兴趣的朋友可以自己试一下。

对于这些需求，其实只需要通过一个正则表达式就可以搞定了，下一小节开始将向大家介绍正则表达式。

6.10.1 初识正则表达式

正则表达式，又称为规则表达式（英语: Regular Expression），正则表达式通常被用来检索、替换那些符合某个模式（规则）的字符串，正则表达式描述了一种字符串的匹配模式（pattern），可以用来检查一个给定的字符串中是否含有某种子字符串、将匹配的字符串替换或者从某个字符串中取出符合某个条件的子字符串。

在 Linux 系统下运行命令的时候，相信大家都使用过?或*通配符来查找硬盘上的文件或者文本中的某个字符串，?通配符匹配 0 个或 1 个字符，而*通配符匹配 0 个或多个字符，譬如"data?.txt"这样的匹配模式可以将下列文件查找出来：

```
data.dat
data1.dat
data2.dat
datax.dat
dataN.dat
```

尽管使用通配符的方法很有用，但它还是很有限，正则表达式则更加强大、更加灵活。

正则表达式其实也是一个字符串，该字符串由普通字符（譬如，数字 0~9、大小写字母以及其它字符）和特殊字符（称为“元字符”）所组成，由这些字符组成一个“规则字符串”，这个“规则字符串”用来表达对给定字符串的一种查找、匹配逻辑。

许多程序设计语言都支持正则表达式。譬如，在 Perl 中就内建了一个功能强大的正则表达式引擎、Python 提供了内置模块 re 用于处理正则表达式，正则表达式这个概念最初是由 Unix 中的工具软件（例如 sed 和 grep）普及开的，使用过 sed 命令的朋友想必对正则表达式并不陌生。同样，在 C 语言函数库中也提供了用于处理正则表达式的接口供程序员使用。

6.11 C 语言中使用正则表达式

编译正则表达式
匹配正则表达式
释放正则表达式
匹配 URL 的正则表达式：

```
^((ht|f)tps?)://[-A-Za-z0-9_]+(\.[-A-Za-z0-9_]+)+([-A-Za-z0-9_.,@?^=%&:/~+#]*[-A-Za-z0-9_@?^=%&/~+#])?$
```

示例代码 6.11.1 C 应用程序中使用正则表达式

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <regex.h>
#include <string.h>

int main(int argc, char *argv[])
{
    regmatch_t pmatch = {0};
    regex_t reg;
    char errbuf[64];
    int ret;
    char *sptr;
    int length;
    int nmatch; //最多匹配出的结果

    if (4 != argc) {
        /***********************************
         * 执行程序时需要传入两个参数:
         * arg1: 正则表达式
         * arg2: 待测试的字符串
         * arg3: 最多匹配出多少个结果
         ***********************************/
        fprintf(stderr, "usage: %s <regex> <string> <nmatch>\n", argv[0]);
        exit(0);
    }
```

```
    /* 编译正则表达式 */
    if(ret = regcomp(&reg, argv[1], REG_EXTENDED)) {
        regerror(ret, &reg, errbuf, sizeof(errbuf));
        fprintf(stderr, "regcomp error: %s\n", errbuf);
        exit(0);
    }

    /* 赋值操作 */
    sptr = argv[2];             //待测试的字符串
    length = strlen(argv[2]);//获取字符串长度
    nmatch = atoi(argv[3]); //获取最大匹配数

    /* 匹配正则表达式 */
    for (int j = 0; j < nmatch; j++) {

        char temp_str[100];

        /* 调用 regexec 匹配正则表达式 */
        if(ret = regexec(&reg, sptr, 1, &pmatch, 0)) {
            regerror(ret, &reg, errbuf, sizeof(errbuf));
            fprintf(stderr, "regexec error: %s\n", errbuf);
            goto out;
        }

        if(-1 != pmatch.rm_so) {
            if (pmatch.rm_so == pmatch.rm_eo) {//空字符串
                sptr += 1;
                length -= 1;
                printf("\n");     //打印出空字符串

                if (0 >= length)//如果已经移动到字符串末尾、则退出
                    break;
                continue;     //从 for 循环开始执行
            }

            memset(temp_str, 0x00, sizeof(temp_str));//清零缓冲区
            memcpy(temp_str, sptr + pmatch.rm_so,
                    pmatch.rm_eo - pmatch.rm_so);//将匹配出来的子字符串拷贝到缓冲区
            printf("%s\n", temp_str);           //打印字符串

            sptr += pmatch.rm_eo;
            length -= pmatch.rm_eo;
            if (0 >= length)
```

```
                    break;
                }
            }

            /* 释放正则表达式 */
out:
            regfree(&reg);
            exit(0);
}
```

第七章 系统信息与系统资源

在应用程序当中，有时往往需要去获取到一些系统相关的信息，譬如时间、日期、以及其它一些系统相关信息，本章将向大家介绍如何通过 Linux 系统调用或 C 库函数获取系统信息，譬如获取系统时间、日期以及设置系统时间、日期等；除此之外，还会向大家介绍 Linux 系统下的/proc 虚拟文件系统，包括/proc 文件系统是什么以及如何从/proc 文件系统中读取系统、进程有关信息。

除了介绍系统信息内容外，本章还会向大家介绍有关系统资源的使用，譬如系统内存资源的申请与使用等。好了，废话不多少，开始本章内容的学习吧！

- 用于获取系统相关信息的函数；
- 时间、日期；
- 进程时间；
- 使程序进入休眠；
- 在堆中申请内存；
- proc 文件系统介绍；
- 定时器。

7.1 系统信息

7.1.1 系统标识 uname

系统调用 uname()用于获取有关当前操作系统内核的名称和信息，函数原型如下所示（可通过"man 2 uname"命令查看）：

```
#include <sys/utsname.h>

int uname(struct utsname *buf);
```

使用该函数需要包含头文件<sys/utsname.h>。

函数参数和返回值含义如下：

buf：struct utsname 结构体类型指针，指向一个 struct utsname 结构体类型对象。

返回值：成功返回 0；失败将返回-1，并设置 errno。

uname()函数用法非常简单，先定义一个 struct utsname 结构体变量，调用 uname()函数时传入变量的地址即可，struct utsname 结构体如下所示：

示例代码 7.1.1 struct utsname 结构体

```
struct utsname {
    char sysname[];         /* 当前操作系统的名称 */
    char nodename[];        /* 网络上的名称（主机名） */
    char release[];         /* 操作系统内核版本 */
    char version[];         /* 操作系统发行版本 */
    char machine[];         /* 硬件架构类型 */
    #ifdef _GNU_SOURCE
        char domainname[];/* 当前域名 */
    #endif
};
```

可以看到，struct utsname 结构体中的所有成员变量都是字符数组，所以获取到的信息都是字符串。

测试

编写一个简单地程序，获取并打印出当前操作系统名称、主机名、内核版本、操作系统发行版本以及处理器硬件架构类型等信息，测试代码如下：

示例代码 7.1.2 uname 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/utsname.h>

int main(void)
{
    struct utsname os_info;
    int ret;

    /* 获取信息 */
    ret = uname(&os_info);
    if (-1 == ret) {
```

```
        perror("uname error");
        exit(-1);
    }

    /* 打印信息 */
    printf("操作系统名称: %s\n", os_info.sysname);
    printf("主机名: %s\n", os_info.nodename);
    printf("内核版本: %s\n", os_info.release);
    printf("发行版本: %s\n", os_info.version);
    printf("硬件架构: %s\n", os_info.machine);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
操作系统名称: Linux
主机名: dt-virtual-machine
内核版本: 4.15.0-132-generic
发行版本: #136~16.04.1-Ubuntu SMP Tue Jan 12 18:22:20 UTC 2021
硬件架构: x86_64
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.1 运行结果

7.1.2 sysinfo 函数

sysinfo 系统调用可用于获取一些系统统计信息，其函数原型如下所示：

```
#include <sys/sysinfo.h>

int sysinfo(struct sysinfo *info);
```

函数参数和返回值含义如下：

info：struct sysinfo 结构体类型指针，指向一个 struct sysinfo 结构体类型对象。

返回值：成功返回 0；失败将返回-1，并设置 errno。

同样 sysinfo()函数用法也非常简单，先定义一个 struct sysinfo 结构体变量，调用 sysinfo()函数时传入变量的地址即可，struct sysinfo 结构体如下所示：

示例代码 7.1.3 struct sysinfo 结构体

```
struct sysinfo {
    long uptime;              /* 自系统启动之后所经过的时间（以秒为单位） */
    unsigned long loads[3];   /* 1, 5, and 15 minute load averages */
    unsigned long totalram;   /* 总的可用内存大小 */
    unsigned long freeram;    /* 还未被使用的内存大小 */
    unsigned long sharedram;  /* Amount of shared memory */
    unsigned long bufferram;  /* Memory used by buffers */
    unsigned long totalswap;  /* Total swap space size */
    unsigned long freeswap;   /* swap space still available */
```

```
    unsigned short procs;          /* 系统当前进程数量 */
    unsigned long totalhigh;       /* Total high memory size */
    unsigned long freehigh;        /* Available high memory size */
    unsigned int mem_unit;         /* 内存单元大小（以字节为单位） */
    char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding to 64 bytes */
};
```

测试

示例代码 7.1.4 sysinfo 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysinfo.h>

int main(void)
{
    struct sysinfo sys_info;
    int ret;

    /* 获取信息 */
    ret = sysinfo(&sys_info);
    if (-1 == ret) {
        perror("sysinfo error");
        exit(-1);
    }

    /* 打印信息 */
    printf("uptime: %ld\n", sys_info.uptime);
    printf("totalram: %lu\n", sys_info.totalram);
    printf("freeram: %lu\n", sys_info.freeram);
    printf("procs: %u\n", sys_info.procs);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
uptime: 258083
totalram: 4112277504
freeram: 165687296
procs: 620
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.2 sysinfo 测试结果

7.1.3 gethostname 函数

此函数可用于单独获取 Linux 系统主机名，与 struct utsname 数据结构体中的 nodename 变量一样，gethostname 函数原型如下所示（可通过"man 2 gethostname"命令查看）：

```
#include <unistd.h>

int gethostname(char *name, size_t len);
```

使用此函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

name：指向用于存放主机名字符串的缓冲区。

len：缓冲区长度。

返回值：成功返回 0,；失败将返回-1，并会设置 errno。

测试

使用 gethostname 函数获取系统主机名：

示例代码 7.1.5 gethostname 函数使用示例

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

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

    memset(hostname, 0x0, sizeof(hostname));
    ret = gethostname(hostname, sizeof(hostname));
    if (-1 == ret) {
        perror("gethostname error");
        exit(ret);
    }

    puts(hostname);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt-virtual-machine
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.3 获取主机名

7.1.4 sysconf()函数

sysconf()函数是一个库函数，可在运行时获取系统的一些配置信息，譬如页大小（page size）、主机名的最大长度、进程可以打开的最大文件数、每个用户 ID 的最大并发进程数等。其函数原型如下所示：

```
#include <unistd.h>

long sysconf(int name);
```

使用该函数需要包含头文件<unistd.h>。

调用 sysconf()函数获取系统的配置信息，参数 name 指定了要获取哪个配置信息，参数 name 可取以下任何一个值（都是宏定义，可通过 man 手册查询）：

- **_SC_ARG_MAX：**exec 族函数的参数的最大长度，exec 族函数后面会介绍，这里先不管！
- **_SC_CHILD_MAX：**每个用户的最大并发进程数，也就是同一个用户可以同时运行的最大进程数。
- **_SC_HOST_NAME_MAX：**主机名的最大长度。
- **_SC_LOGIN_NAME_MAX：**登录名的最大长度。
- **_SC_CLK_TCK：**每秒时钟滴答数，也就是系统节拍率。
- **_SC_OPEN_MAX：**一个进程可以打开的最大文件数。
- **_SC_PAGESIZE：**系统页大小（page size）。
- **_SC_TTY_NAME_MAX：**终端设备名称的最大长度。
- ……

除以上之外，还有很多，这里就不再一一列举了，可以通过 man 手册进行查看，用的比较多的是_SC_PAGESIZE 和_SC_CLK_TCK，在后面章节示例代码中有使用到。

若指定的参数 name 为无效值，则 sysconf()函数返回-1，并会将 errno 设置为 EINVAL。否则返回的值便是对应的配置值。注意，返回值是一个 long 类型的数据。

使用示例

获取每个用户的最大并发进程数、系统节拍率和系统页大小。

示例代码 7.1.6 sysconf()函数使用示例

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

int main(void)
{
    printf("每个用户的最大并发进程数: %ld\n", sysconf(_SC_CHILD_MAX));
    printf("系统节拍率: %ld\n", sysconf(_SC_CLK_TCK));
    printf("系统页大小: %ld\n", sysconf(_SC_PAGESIZE));
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
每个用户的最大并发进程数： 15446
系统节拍率： 100
系统页大小： 4096
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.1.4 测试结果

7.2 时间、日期

本小节向大家介绍下时间、日期相关的系统调用或 C 库函数以及它们的使用方法。

7.2.1 时间的概念

在正式介绍这些时间、日期相关的系统调用或 C 库函数之前，需要向大家介绍一些时间相关的基本概念，譬如 GMT 时间、UTC 时间以及时区等。

地球总是自西向东自转，东边总比西边先看到太阳，东边的时间也总比西边的早。东边时刻与西边时刻的差值不仅要以时计，而且还要以分和秒来计算，这给人们的日常生活和工作都带来许多不便。

GMT 时间

GMT（Greenwich Mean Time）中文全称是格林威治标准时间，这个时间系统的概念在 1884 年被确立，由英国伦敦的格林威治皇家天文台计算并维护，并在之后的几十年向欧陆其它国家扩散。

由于从 19 实际开始，因为世界各国往来频繁，而欧洲大陆、美洲大陆以及亚洲大陆都有各自的时区，所以为了避免时间混乱，1884 年，各国代表在美国华盛顿召开国际大会，通过协议选出英国伦敦的格林威治作为全球时间的中心点，决定以通过格林威治的子午线作为划分东西两半球的经线零度线（本初子午线、零度经线），由此格林威治标准时间因而诞生！

所以 GMT 时间就是英国格林威治当地时间，也就是零时区（中时区）所在时间，譬如 GMT 12:00 就是指英国伦敦的格林威治皇家天文台当地的中午 12:00，与我国的标准时间北京时间（东八区）相差 8 个小时，即早八个小时，所以 GMT 12:00 对应的北京时间是 20:00。

UTC 时间

UTC（Coordinated Universal Time）指的是世界协调时间（又称世界标准时间、世界统一时间），是经过平均太阳时(以格林威治时间 GMT 为准)、地轴运动修正后的新时标以及以「秒」为单位的国际原子时所综合精算而成的时间，计算过程相当严谨精密，因此若以「世界标准时间」的角度来说，UTC 比 GMT 来得更加精准。

GMT 与 UTC 这两者几乎是同一概念，它们都是指格林威治标准时间，也就是国际标准时间，只不过 UTC 时间比 GMT 时间更加精准，所以在我们的编程当中不用刻意去区分它们之间的区别。

在 Ubuntu 系统下，可以使用"date -u"命令查看到当前的 UTC 时间，如下所示：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ date -u
2021年 02月 24日 星期三 02:34:13 UTC
dt@dt-virtual-machine:~$
```

图 7.2.1 查看 UTC 时间

后面显示的 UTC 字样就表示当前查看到的时间是 UTC 时间，也就是国际标准时间。

时区

全球被划分为 24 个时区，每一个时区横跨经度 15 度，以英国格林威治的本初子午线作为零度经线，将全球划分为东西两半球，分为东一区、东二区、东三区……东十二区以及西一区、西二区、西三区……西十二区，而本初子午线所在时区被称为中时区（或者叫零时区），划分图如下所示：

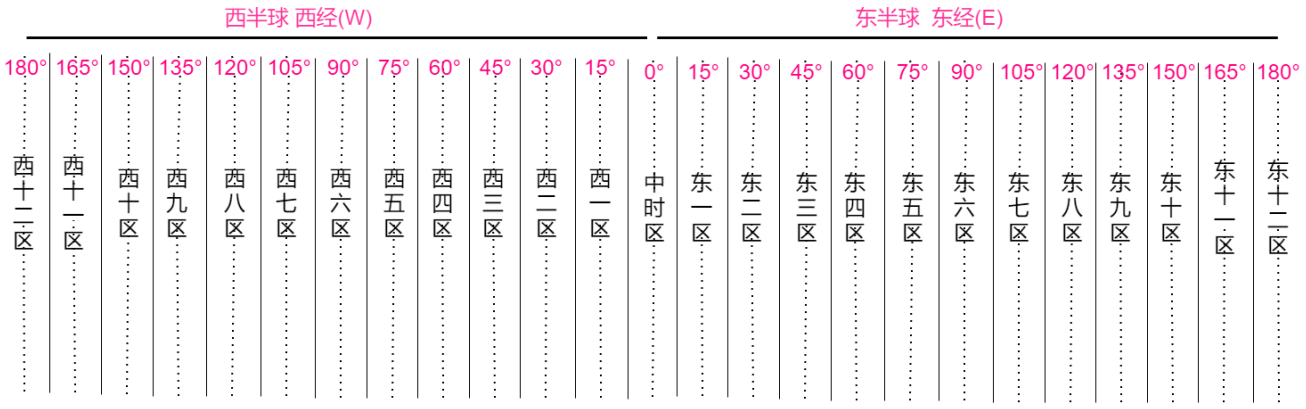


图 7.2.2 全球 24 时区划分

东十二区和西十二区其实是一个时区，就是十二区，东十二区与西十二区各横跨经度 7.5 度，以 180 度经线作为分界线。每个时区的中央经线上的时间就是这个时区内统一采用的时间，称为区时。相邻两个时区的时间相差 1 小时。例如，我国东 8 区的时间总比泰国东 7 区的时间早 1 小时，而比日本东 9 区的时间晚 1 小时。因此，出国旅行的人，必须随时调整自己的手表，才能和当地时间相一致。凡向西走，每过一个时区，就要把表向前拨 1 小时(比如 2 点拨到 1 点)；凡向东走，每过一个时区，就要把表向后拨 1 小时（比如 1 点拨到 2 点）。

实际上，世界上不少国家和地区都不严格按时区来计算时间。为了在全国范围内采用统一的时间，一般都把某一个时区的时间作为全国统一采用的时间。例如，我国把首都北京所在的东 8 区的时间作为全国统一的时间，称为北京时间，北京时间就作为我国使用的本地时间，譬如我们电脑上显示的时间就是北京时间，我国国土面积广大，由东到西横跨了 5 个时区，也就意味着我国最东边的地区与最西边的地区实际上相差了 4、5 个小时。又例如，英国、法国、荷兰和比利时等国，虽地处中时区，但为了和欧洲大多数国家时间相一致，则采用东 1 区的时间。

譬如在 Ubuntu 系统下，可以使用 date 命令查看系统当前的本地时间，如下所示：

```
dt@dt-virtual-machine:~$ date
2021年 02月 23日 星期二 16:35:45 CST
dt@dt-virtual-machine:~$
```

图 7.2.3 date 查看本地时间

可以看到显示出来的字符串后面有一个"CST"字样，CST 在这里其实指的是 China Standard Time（中国标准时间）的缩写，表示当前查看到的时间是中国标准时间，也就是我国所使用的标准时间--北京时间，一般在安装 Ubuntu 系统的时候会提示用户设置所在城市，那么系统便会根据你所设置的城市来确定系统的本地时间对应的时区，譬如设置的城市为上海，那么系统的本地时间就是北京时间，因为我国统一使用北京时间作为本国的标准时间。

在 Ubuntu 系统下，时区信息通常以标准格式保存在一些文件当中，这些文件通常位于/usr/share/zoneinfo 目录下，该目录下的每一个文件（包括子目录下的文件）都包含了一个特定国家或地区内时区制度的相关信息，且往往根据其所描述的城市或地区缩写来加以命名，譬如 EST（美国东部标准时间）、CET（欧洲中部时间）、UTC（世界标准时间）、Hongkong、Iran、Japan（日本标准时间）等，也把这些文件称为时区配置文件，如下图所示：

```
dt@dt-virtual-machine:/usr/share/zoneinfo$ pwd
/usr/share/zoneinfo
dt@dt-virtual-machine:/usr/share/zoneinfo$ ls
Africa      Cuba      GMT0          Japan             Pacific     Turkey
America     EET       GMT-0         Kwajalein         Poland      UCT
Antarctica  Egypt     GMT+0         leap-seconds.list Portugal    Universal
Arctic      Eire      Greenwich     Libya             posix       US
Asia        EST       Hongkong      localtime         posixrules  UTC
Atlantic    EST5EDT   HST           MET               PRC         WET
Australia   Etc       Iceland       Mexico            PST8PDT     W-SU
Brazil      Europe    Indian        MST               right       zone1970.tab
Canada      Factory   Iran          MST7MDT           ROC         zone.tab
CET         GB        iso3166.tab   Navajo            ROK         Zulu
Chile       GB-Eire   Israel        NZ                Singapore
CST6CDT     GMT       Jamaica       NZ-CHAT           SystemV
dt@dt-virtual-machine:/usr/share/zoneinfo$
```

图 7.2.4 时区信息配置文件

系统的本地时间由时区配置文件/etc/localtime 定义，通常链接到/usr/share/zoneinfo 目录下的某一个文件（或其子目录下的某一个文件）：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ls -l /etc/localtime
lrwxrwxrwx 1 root root 33 2月  23 17:14 /etc/localtime -> /usr/share/zoneinfo/Asia/Shanghai
dt@dt-virtual-machine:~$
```

图 7.2.5 /etc/localtime 配置文件

如果我们要修改 Ubuntu 系统本地时间的时区信息，可以直接将/etc/localtime 链接到/usr/share/zoneinfo 目录下的任意一个时区配置文件，譬如 EST（美国东部标准时间），首先进入到/etc 目录下，执行下面的命令：

```
sudo rm -rf localtime                    #删除原有链接文件
sudo ln -s /usr/share/zoneinfo/EST localtime    #重新建立链接文件
```

```
dt@dt-virtual-machine:/etc$
dt@dt-virtual-machine:/etc$ sudo rm -rf localtime
dt@dt-virtual-machine:/etc$
dt@dt-virtual-machine:/etc$ sudo ln -s /usr/share/zoneinfo/EST localtime
dt@dt-virtual-machine:/etc$
```

图 7.2.6 修改本地时间对应的时区配置文件

接下来再使用 date 命令查看下系统当前的时间，如下所示：

```
dt@dt-virtual-machine:~$ date
2021年 02月 23日 星期二 04:33:39 EST
dt@dt-virtual-machine:~$
```

图 7.2.7 date 查看时间

可以发现后面的标识变成了 EST，也就意味着当前系统的本地时间变成了 EST 时间（美国东部标准时间）。

关于时区的信息就给大家介绍这么多了。

世界标准时间的意义

世界标准时间指的就是格林威治时间，也就是中时区对应的时间，用格林威治当地时间作为全球统一时间，用以描述全球性的事件，方便大家记忆、以免混淆。

本小节关于时间的概念就给大家介绍这么多，其中涉及到的很多内容都是初中地理或高中地理中的知识，譬如本初子午线、经线、时区等，相信大家都学过，这里笔者便不再啰嗦！

7.2.2 Linux 系统中的时间

点时间和段时间

通常描述时间有两种方式：点时间和段时间；点时间顾名思义指的是某一个时间点，譬如当前时间是 2021 年 2 月 22 日星期一 11:12 分 35 秒，所以这里指的就是某一个时间点；而对于段时间来说，顾名思义指的是某一个时间段，譬如早上 8:00 到中午 12:00 这段时间。

实时时钟 RTC

操作系统中一般会有两个时钟，一个系统时钟（system clock），一个实时时钟（Real time clock），也叫 RTC；系统时钟由系统启动之后由内核来维护，譬如使用 date 命令查看到的就是系统时钟，所以在系统关机情况下是不存在的；而实时时钟一般由 RTC 时钟芯片提供，RTC 芯片有相应的电池为其供电，以保证系统在关机情况下 RTC 能够继续工作、继续计时。

Linux 系统如何记录时间

Linux 系统在开机启动之后首先会读取 RTC 硬件获取实时时钟作为系统时钟的初始值，之后内核便开始维护自己的系统时钟。所以由此可知，RTC 硬件只有在系统开机启动时会读取一次，目的是用于对系统时钟进行初始化操作，之后的运行过程中便不会再对其进行读取操作了。

而在系统关机时，内核会将系统时钟写入到 RTC 硬件、已进行同步操作。

jiffies 的引入

jiffies 是内核中定义的一个全局变量，内核使用 jiffies 来记录系统从启动以来的系统节拍数，所以这个变量用来记录以系统节拍时间为单位的时间长度，Linux 内核在编译配置时定义了一个节拍时间，使用节拍率（一秒钟多少个节拍数）来表示，譬如常用的节拍率为 100Hz（一秒钟 100 个节拍数，节拍时间为 1s / 100）、200Hz（一秒钟 200 个节拍，节拍时间为 1s / 200）、250Hz（一秒钟 250 个节拍，节拍时间为 1s / 250）、300Hz（一秒钟 300 个节拍，节拍时间为 1s / 300）、500Hz（一秒钟 500 个节拍，节拍时间为 1s / 500）等。由此可以发现配置的节拍率越低，每一个系统节拍的时间就越短，也就意味着 jiffies 记录的时间精度越高，当然，高节拍率会导致系统中断的产生更加频繁，频繁的中断会加剧系统的负担，一般默认情况下都是采用 100Hz 作为系统节拍率。

内核其实通过 jiffies 来维护系统时钟，全局变量 jiffies 在系统开机启动时会设置一个初始值，上面也给大家提到过，RTC 实时时钟会在系统开机启动时读取一次，目的是用于对系统时钟进行初始化，这里说的初始化其实指的就是对内核的 jiffies 变量进行初始化操作，具体如何将读取到的实时时钟换算成 jiffies 数值，这里便不再给大家介绍了。

所以由此可知，操作系统使用 jiffies 这个全局变量来记录当前时间，当我们需要获取到系统当前时间点时，就可以使用 jiffies 变量去计算，当然并不需要我们手动去计算，Linux 系统提供了相应的系统调用或 C 库函数用于获取当前时间，譬如系统调用 time()、gettimeofday()，其实质上就是通过 jiffies 变量换算得到。

7.2.3 获取时间 time/gettimeofday

(1)time 函数

系统调用 time()用于获取当前时间，以秒为单位，返回得到的值是自 1970-01-01 00:00:00 +0000 (UTC)以来的秒数，其函数原型如下所示（可通过"man 2 time"命令查看）：

```
#include <time.h>

time_t time(time_t *tloc);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下：

tloc：如果 tloc 参数不是 NULL，则返回值也存储在 tloc 指向的内存中。

返回值：成功则返回自 1970-01-01 00:00:00 +0000 (UTC)以来的时间值（以秒为单位）；失败则返回-1，并会设置 errno。

所以由此可知，time 函数获取得到的是一个时间段，也就是从 1970-01-01 00:00:00 +0000 (UTC)到现在这段时间所经过的秒数，所以你要计算现在这个时间点，只需要使用 time()得到的秒数加 1970-01-01 00:00:00 即可！当然，这并不需要我们手动去计算，可以直接使用相关系统调用或 C 库函数来得到当前时间，后面再给大家介绍。

自 1970-01-01 00:00:00 +0000 (UTC)以来经过的总秒数，我们把这个称之为日历时间或 time_t 时间。

测试

使用系统调用 time()获取自 1970-01-01 00:00:00 +0000 (UTC)以来的时间值：

示例代码 7.2.1 time 函数使用示例

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

int main(void)
{
    time_t t;

    t = time(NULL);
    if (-1 == t) {
        perror("time error");
        exit(-1);
    }

    printf("时间值: %ld\n", t);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
时间值: 1613968524
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.8 time 函数测试结果

(2)gettimeofday 函数

time()获取到的时间只能精确到秒，如果想要获取更加精确的时间可以使用系统调用 gettimeofday 来实现，gettimeofday()函数提供微秒级时间精度，函数原型如下所示（可通过"man 2 gettimeofday"命令查看）：

```
#include <sys/time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

使用该函数需要包含头文件<sys/time.h>。

函数参数和返回值含义如下：

tv：参数 tv 是一个 struct timeval 结构体指针变量，struct timeval 结构体在前面章节内容中已经给大家介绍过，具体参考示例代码 5.6.3。

tz：参数 tz 是个历史产物，早期实现用其来获取系统的时区信息，目前已遭废弃，在调用 gettimeofday()函数时应将参数 tz 设置为 NULL。

返回值：成功返回 0；失败将返回-1，并设置 errno。

获取得到的时间值存储在参数 tv 所指向的 struct timeval 结构体变量中，该结构体包含了两个成员变量 tv_sec 和 tv_usec，分别用于表示秒和微秒，所以获取得到的时间值就是 tv_sec（秒）+tv_usec（微秒），同样获取得到的秒数与 time()函数一样，也是自 1970-01-01 00:00:00 +0000 (UTC)到现在这段时间所经过的秒数，也就是日历时间，所以由此可知 time()返回得到的值和函数 gettimeofday()所返回的 tv 参数中 tv_sec 字段的数值相同。

测试

使用 gettimeofday 获取自 1970-01-01 00:00:00 +0000 (UTC)以来的时间值：

示例代码 7.2.2 gettimeofday 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
    struct timeval tval;
    int ret;

    ret = gettimeofday(&tval, NULL);
    if (-1 == ret) {
        perror("gettimeofday error");
        exit(-1);
    }

    printf("时间值: %ld 秒+%ld 微秒\n", tval.tv_sec, tval.tv_usec);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
时间值: 1613976912秒+99626微秒
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.9 gettimeofday 函数测试结果

7.2.4 时间转换函数

通过 time()或 gettimeofday()函数可以获取到当前时间点相对于 1970-01-01 00:00:00 +0000 (UTC)这个时间点所经过时间（日历时间），所以获取得到的是一个时间段的长度，但是这并不利于我们查看当前时间，

这个结果对于我们来说非常不友好，那么本小节将向大家介绍一些系统调用或 C 库函数，通过这些 API 可以将 time()或 gettimeofday()函数获取到的秒数转换为利于查看和理解的形式。

(1)ctime 函数

ctime()是一个 C 库函数，可以将日历时间转换为可打印输出的字符串形式，ctime()函数原型如下所示：

```
#include <time.h>

char *ctime(const time_t *timep);
char *ctime_r(const time_t *timep, char *buf);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下：

timep： time_t 时间变量指针。

返回值： 成功将返回一个 char *类型指针，指向转换后得到的字符串；失败将返回 NULL。

所以由此可知，使用 ctime 函数非常简单，只需将 time_t 时间变量的指针传入即可，调用成功便可返回字符串指针，拿到字符串指针之后，可以使用 printf 将其打印输出。但是 ctime()是一个不可重入函数，存在一些安全上面的隐患，ctime_r()是 ctime()的可重入版本，一般推荐大家使用可重入函数 ctime_r()，可重入函数 ctime_r()多了一个参数 buf，也就是缓冲区首地址，所以 ctime_r()函数需要调用者提供用于存放字符串的缓冲区。

Tips：关于可重入函数与不可重入函数将会在后面章节内容中进行介绍，这里暂时先不去管这个问题，在 Linux 系统中，有一些系统调用或 C 库函数提供了可重入版本与不可重入版本的函数接口，可重入版本函数所对应的函数名一般都会有一个"_r"后缀来表明它是一个可重入函数。

ctime（或 ctime_r）转换得到的时间是计算机所在地对应的本地时间（譬如在中国对应的便是北京时间），并不是 UTC 时间，接下来编写一段简单地代码进行测试。

测试

示例代码 7.2.3 time/time_r 函数使用示例

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

int main(void)
{
    char tm_str[100] = {0};
    time_t tm;

    /* 获取时间 */
    tm = time(NULL);
    if (-1 == tm) {
        perror("time error");
        exit(-1);
    }

    /* 将时间转换为字符串形式 */
    ctime_r(&tm, tm_str);
```

```
    /* 打印输出 */
    printf("当前时间: %s", tm_str);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
当前时间: Mon Feb 22 17:10:46 2021
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.10 打印当前时间

从图中可知，打印出来的时间为"Mon Feb 22 17:10:46 2021"，Mon 表示星期一，这是一个英文单词的缩写，Feb 表示二月份，这也是一个英文单词的缩写，22 表示 22 日，所以整个打印信息显示的时间就是 2021 年 2 月 22 日星期一 17 点 10 分 46 秒。

(2)localtime 函数

localtime()函数可以把 time()或 gettimeofday()得到的秒数（time_t 时间或日历时间）变成一个 struct tm 结构体所表示的时间，该时间对应的是本地时间。localtime 函数原型如下：

```
#include <time.h>

struct tm *localtime(const time_t *timep);
struct tm *localtime_r(const time_t *timep, struct tm *result);
```

使用该函数需要包含头文件<time.h>，localtime()的可重入版本为 localtime_r()。

函数参数和返回值含义如下：

timep：需要进行转换的 time_t 时间变量对应的指针，可通过 time()或 gettimeofday()获取得到。

result：是一个 struct tm 结构体类型指针，稍后给大家介绍 struct tm 结构体，参数 result 是可重入函数 localtime_r()需要额外提供的参数。

返回值：对于不可重入版本 localtime()来说，成功则返回一个有效的 struct tm 结构体指针，而对于可重入版本 localtime_r()来说，成功执行情况下，返回值将会等于参数 result；失败则返回 NULL。

使用不可重入函数 localtime()并不需要调用者提供 struct tm 变量，而是它会直接返回出来一个 struct tm 结构体指针，然后直接通过该指针访问里边的成员变量即可！虽然很方便，但是存在一些安全隐患，所以一般不推荐使用不可重入版本。

使用可重入版本 localtime_r()调用者需要自己定义 struct tm 结构体变量、并将该变量指针赋值给参数 result，在函数内部会对该结构体变量进行赋值操作。

struct tm 结构体如下所示：

示例代码 7.2.4 struct tm 结构体

```
struct tm {
    int tm_sec;         /* 秒(0-60) */
    int tm_min;         /* 分(0-59) */
    int tm_hour;        /* 时(0-23) */
    int tm_mday;        /* 日(1-31) */
    int tm_mon;         /* 月(0-11) */
```

```
    int tm_year;        /* 年(这个值表示的是自 1900 年到现在经过的年数) */
    int tm_wday;        /* 星期(0-6, 星期日 Sunday = 0、星期一=1…) */
    int tm_yday;        /* 一年里的第几天(0-365, 1 Jan = 0) */
    int tm_isdst;       /* 夏令时 */
};
```

从 struct tm 结构体内容可知，该结构体中包含了年月日时分秒星期等信息，使用 localtime/localtime_r()便可以将 time_t 时间总秒数分解成了各个独立的时间信息，易于我们查看和理解。

测试

示例代码 7.2.5 localtime/localtime_r 函数使用示例

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

int main(void)
{
    struct tm t;
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

    /* 转换得到本地时间 */
    localtime_r(&sec, &t);

    /* 打印输出 */
    printf("当前时间: %d 年%d 月%d 日 %d:%d:%d\n",
                t.tm_year + 1900, t.tm_mon, t.tm_mday,
                t.tm_hour, t.tm_min, t.tm_sec);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
当前时间: 2021年1月22日 18:38:32
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.11 localtime/localtime_r 测试结果

(3)gmtime 函数

gmtime()函数也可以把 time_t 时间变成一个 struct tm 结构体所表示的时间，与 localtime()所不同的是，gmtime()函数所得到的是 UTC 国际标准时间，并不是计算机的本地时间，这是它们之间的唯一区别。gmtime()函数原型如下所示：

```
#include <time.h>

struct tm *gmtime(const time_t *timep);
struct tm *gmtime_r(const time_t *timep, struct tm *result);
```

同样使用 gmtime()函数需要包含头文件<time.h>。

gmtime_r()是 gmtime()的可重入版本，同样也是推荐大家使用可重入版本函数 gmtime_r。关于该函数的参数和返回值，这里便不再介绍，与 localtime()是一样的。

测试

使用 localtime 获取本地时间、使用 gmtime 获取 UTC 国际标准时间，并进行对比：

示例代码 7.2.6 gmtime 函数使用示例

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

int main(void)
{
    struct tm local_t;
    struct tm utc_t;
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

    /* 转换得到本地时间 */
    localtime_r(&sec, &local_t);

    /* 转换得到国际标准时间 */
    gmtime_r(&sec, &utc_t);

    /* 打印输出 */
    printf("本地时间:    %d 年%d 月%d 日 %d:%d:%d\n",
                local_t.tm_year + 1900, local_t.tm_mon, local_t.tm_mday,
                local_t.tm_hour, local_t.tm_min, local_t.tm_sec);
    printf("UTC 时间:   %d 年%d 月%d 日 %d:%d:%d\n",
                utc_t.tm_year + 1900, utc_t.tm_mon, utc_t.tm_mday,
                utc_t.tm_hour, utc_t.tm_min, utc_t.tm_sec);
```

```
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本地时间:       2021年1月22日 19:30:7
UTC时间:        2021年1月22日 11:30:7
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.12 打印本地时间与 UTC 时间

从打印结果可知，本地时间与 UTC 时间（国际标准时间）相差 8 个小时，因为笔者使用的计算机其对应的本地时间指的便是北京时间，而北京时间要早于国际标准时间 8 个小时（东八区）。

(4)mktime 函数

mktime()函数与 localtime()函数相反，mktime()可以将使用 struct tm 结构体表示的分解时间转换为 time_t 时间（日历时间），同样这也是一个 C 库函数，其函数原型如下所示：

```
#include <time.h>

time_t mktime(struct tm *tm);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下：

tm：需要进行转换的 struct tm 结构体变量对应的指针。

返回值：成功返回转换得到 time_t 时间值；失败返回-1。

测试

示例代码 7.2.7 mktime 函数使用示例

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

int main(void)
{
    struct tm local_t;
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

    printf("获取得到的秒数: %ld\n", sec);
    localtime_r(&sec, &local_t);
```

```
    printf("转换得到的秒数: %ld\n", mktime(&local_t));

    exit(0);
}
```

运行结果：

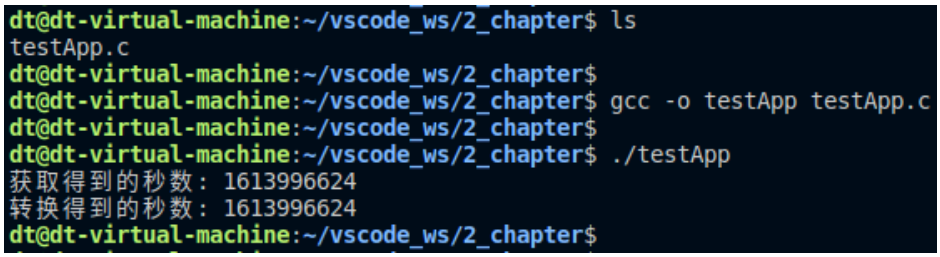


图 7.2.13 测试结果

(5)asctime 函数

asctime()函数与 ctime()函数的作用一样，也可将时间转换为可打印输出的字符串形式，与 ctime()函数的区别在于，ctime()是将 time_t 时间转换为固定格式字符串、而 asctime()则是将 struct tm 表示的分解时间转换为固定格式的字符串。asctime()函数原型如下所示：

```
#include <time.h>

char *asctime(const struct tm *tm);
char *asctime_r(const struct tm *tm, char *buf);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下：

tm：需要进行转换的 struct tm 表示的时间。

buf：可重入版本函数 asctime_r 需要额外提供的参数 buf，指向一个缓冲区，用于存放转换得到的字符串。

返回值：转换失败将返回 NULL；成功将返回一个 char *类型指针，指向转换后得到的时间字符串，对于 asctime_r 函数来说，返回值就等于参数 buf。

测试

示例代码 7.2.8 asctime 函数使用示例

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

int main(void)
{
    struct tm local_t;
    char tm_str[100] = {0};
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
```

```
        perror("time error");
        exit(-1);
    }

    localtime_r(&sec, &local_t);
    asctime_r(&local_t, tm_str);
    printf("本地时间: %s", tm_str);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本地时间: Mon Feb 22 20:40:25 2021
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.14 测试结果

(6)strftime 函数

除了 asctime()函数之外，这里再给大家介绍一个 C 库函数 strftime()，此函数也可以将一个 struct tm 变量表示的分解时间转换为为格式化字符串，并且在功能上比 asctime()和 ctime()更加强大，它可以根据自己的喜好自定义时间的显示格式，而 asctime()和 ctime()转换得到的字符串时间格式的固定的。

strftime()函数原型如下所示：

```
#include <time.h>

size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下：

s：指向一个缓存区的指针，该缓冲区用于存放生成的字符串。

max：字符串的最大字节数。

format：这是一个用字符串表示的字段，包含了普通字符和特殊格式说明符，可以是这两种字符的任意组合。特殊格式说明符将会被替换为 struct tm 结构体对象所指时间的相应值，这些特殊格式说明符如下：

表 7.2.1 strftime 函数特殊格式说明符

说明符	表示含义	实例
%a	星期的缩写	Sun
%A	星期的完整名称	Sunday
%b	月份的缩写	Mar
%B	月份的完整名称	March
%c	系统当前语言环境对应的首选日期和时间表示形式	
%C	世纪（年/100）	20
%d	十进制数表示一个月中的第几天（01-31）	15、05
%D	相当于%m/%d/%y	01/14/21
%e	与%d 相同，但是单个数字时，前导 0 会被去掉	15、5

%F	相当于%Y-%m-%d	2021-01-14
%h	相当于%b	Jan
%H	十进制数表示的 24 小时制的小时（范围 00-23）	01、22
%I	十进制数表示的 12 小时制的小时（范围 01-12）	01、11
%j	十进制数表示的一年中的某天（范围 001-366）	050、285
%k	与%H 相同，但是单个数字时，前导 0 会被去掉（范围 0-23）	1、22
%l	与%I 相同，但是单个数字时，前导 0 会被去掉（范围 1-12）	1、11
%m	十进制数表示的月份（范围 01-12）	01、10
%M	十进制数表示的分钟（范围 00-59）	01、55
%n	换行符	
%p	根据给定的时间值，添加“AM”或“PM”	PM
%P	与%p 相同，但会使用小写字母表示	pm
%r	相当于%I:%M:%S %p	12:15:31 PM
%R	相当于%H:%M	12:16
%S	十进制数表示的秒数（范围 00-60）	05、30
%T	相当于%H:%M:%S	12:20:03
%u	十进制数表示的星期（范围 1-7，星期一为 1）	1、5
%U	十进制数表示，当前年份的第几个星期（范围 00-53），从第一个星期日作为 01 周的第一天开始	
%W	十进制数表示，当前年份的第几个星期（范围 00-53），从第一个星期一作为第 01 周的第一天开始	
%w	十进制数表示的星期，范围为 0-6，星期日为 0	
%x	系统当前语言环境的首选日期表示形式，没有时间	01/14/21
%X	系统当前语言环境的首选时间表示形式，没有日期	12:30:16
%y	十进制数表示的年份（后两字数字）	21
%Y	十进制数表示的年份（4 个数字）	2021
%%	输出%符号	%

strftime 函数的特殊格式说明符还是比较多的，不用去记它，需要用的时候再去查即可！

通过上表可知，譬如我要想输出"2021-01-14 16:30:25<PM> January Thursday"这样一种形式表示的时间日期，那么就可以这样来设置 format 参数：

```
"%Y-%m-%d %H:%M:%S<%p> %B %A"
```

tm：指向 struct tm 结构体对象的指针。

返回值：如果转换得到的目标字符串不超过最大字节数（也就是 max），则返回放置到 s 数组中的字节数；如果超过了最大字节数，则返回 0。

测试

示例代码 7.2.9 strftime 函数使用示例

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

int main(void)
{
```

```
    struct tm local_t;
    char tm_str[100] = {0};
    time_t sec;

    /* 获取时间 */
    sec = time(NULL);
    if (-1 == sec) {
        perror("time error");
        exit(-1);
    }

    localtime_r(&sec, &local_t);
    strftime(tm_str, sizeof(tm_str), "%Y-%m-%d %A %H:%M:%S", &local_t);
    printf("本地时间: %s\n", tm_str);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本地时间: 2021-02-22 Monday 21:07:46
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.2.15 测试结果

7.2.5 设置时间 settimeofday

使用 settimeofday()函数可以设置时间，也就是设置系统的本地时间，函数原型如下所示：

```
#include <sys/time.h>

int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

首先使用该函数需要包含头文件<sys/time.h>。

函数参数和返回值含义如下：

tv：参数 tv 是一个 struct timeval 结构体指针变量，struct timeval 结构体在前面章节内容中已经给大家介绍了，需要设置的时间便通过参数 tv 指向的 struct timeval 结构体变量传递进去。

tz：参数 tz 是个历史产物，早期实现用其来设置系统的时区信息，目前已遭废弃，在调用 settimeofday()函数时应将参数 tz 设置为 NULL。

返回值：成功返回 0；失败将返回-1，并设置 errno。

使用 settimeofday 设置系统时间时内核会进行权限检查，只有超级用户（root）才可以设置系统时间，普通用户将无操作权限。

7.2.6 总结

本小节给大家介绍了时间相关的基本概念，譬如 GMT 时间、UTC 时间以及全球 24 个时区的划分等，并且给大家介绍了 Linux 系统下常用的时间相关的系统调用和库函数，主要有 9 个：

time/ctime/localtime/gmtime/mktime/asctime/strftime/gettimeofday/settimeofday，对这些函数的功能、作用总结如下：

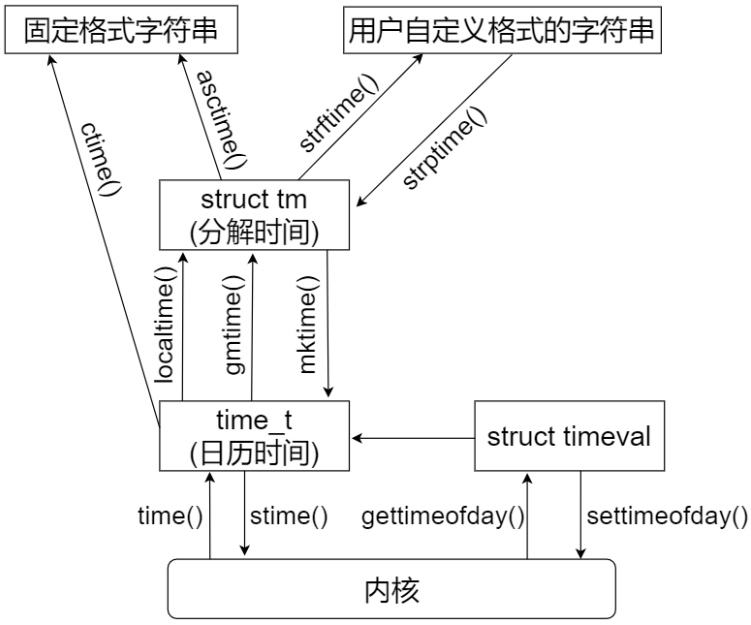


图 7.2.16 时间相关 API 总结

通过上图可以帮助大家快速理解各个函数的功能、作用，大家加油！

本小节到这里就结束了。

7.3 进程时间

进程时间指的是进程从创建后（也就是程序运行后）到目前为止这段时间内使用 CPU 资源的时间总数，出于记录的目的，内核把 CPU 时间（进程时间）分为以下两个部分：

- 用户 CPU 时间：进程在用户空间（用户态）下运行所花费的 CPU 时间。有时也成为虚拟时间（virtual time）。
- 系统 CPU 时间：进程在内核空间（内核态）下运行所花费的 CPU 时间。这是内核执行系统调用或代表进程执行的其它任务（譬如，服务页错误）所花费的时间。

一般来说，进程时间指的是用户 CPU 时间和系统 CPU 时间的总和，也就是总的 CPU 时间。

Tips：进程时间不等于程序的整个生命周期所消耗的时间，如果进程一直处于休眠状态（进程被挂起、不会得到系统调度），那么它并不会使用 CPU 资源，所以休眠的这段时间并不计算在进程时间中。

7.3.1 times 函数

times()函数用于获取当前进程时间，其函数原型如下所示：

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

使用该函数需要包含头文件<sys/times.h>。

函数参数和返回值含义如下：

buf：times()会将当前进程时间信息存在一个 struct tms 结构体数据中，所以我们需要提供 struct tms 变量，使用参数 buf 指向该变量，关于 struct tms 结构体稍后给大家介绍。

返回值：返回值类型为 clock_t（实质是 long 类型），调用成功情况下，将返回从过去任意的一个时间点（譬如系统启动时间）所经过的时钟滴答数（其实就是系统节拍数），将(节拍数 / 节拍率)便可得到秒数，返回值可能会超过 clock_t 所能表示的范围（溢出）；调用失败返回-1，并设置 errno。

如果我们想查看程序运行到某一个位置时的进程时间，或者计算出程序中的某一段代码执行过程所花费的进程时间，都可以使用 times()函数来实现。

struct tms 结构体内容如下所示：

示例代码 7.3.1 struct tms 结构体

```
struct tms {
    clock_t tms_utime;  /* user time, 进程的用户 CPU 时间, tms_utime 个系统节拍数 */
    clock_t tms_stime;  /* system time, 进程的系统 CPU 时间, tms_stime 个系统节拍数 */
    clock_t tms_cutime; /* user time of children, 已死掉子进程的 tms_utime + tms_cutime 时间总和 */
    clock_t tms_cstime; /* system time of children, 已死掉子进程的 tms_stime + tms_cstime 时间总和 */
};
```

测试

以下我们演示了通过 times()来计算程序中某一段代码执行所耗费的进程时间和总的时间，测试程序如下所示：

示例代码 7.3.2 times 函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    struct tms t_buf_start;
    struct tms t_buf_end;
    clock_t t_start;
    clock_t t_end;
    long tck;
    int i, j;

    /* 获取系统的节拍率 */
    tck = sysconf(_SC_CLK_TCK);

    /* 开始时间 */
    t_start = times(&t_buf_start);
    if (-1 == t_start) {
        perror("times error");
        exit(-1);
    }

    /* *****需要进行测试的代码段***** */
    for (i = 0; i < 20000; i++)
```

```
        for (j = 0; j < 20000; j++)
            ;

    sleep(1);           //休眠挂起
    /* **************end*************** */

    /* 结束时间 */
    t_end = times(&t_buf_end);
    if (-1 == t_end) {
        perror("times error");
        exit(-1);
    }

    /* 打印时间 */
    printf("时间总和: %f 秒\n", (t_end - t_start) / (double)tck);
    printf("用户 CPU 时间: %f 秒\n", (t_buf_end.tms_utime - t_buf_start.tms_utime) / (double)tck);
    printf("系统 CPU 时间: %f 秒\n", (t_buf_end.tms_stime - t_buf_start.tms_stime) / (double)tck);
    exit(0);
}
```

首先，笔者先对测试程序做一个简单地介绍，程序中使用 sysconf(_SC_CLK_TCK)获取到系统节拍率，程序还使用了一个库函数 sleep()，该函数也是本章将要向大家介绍的函数，具体参考 7.5.1 小节中的介绍。

示例代码 7.3.2 中对如下代码段进行了测试：

```
for (i = 0; i < 20000; i++)
    for (j = 0; j < 20000; j++)
        ;

sleep(1);       //休眠挂起
```

接下来编译运行，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
时间总和: 2.910000秒
用户CPU时间: 1.900000秒
系统CPU时间: 0.000000秒
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.3.1 测试结果

可以看到用户 CPU 时间为 1.9 秒，系统 CPU 时间为 0 秒，也就是说测试的这段代码并没有进入内核态运行，所以总的进程时间 = 用户 CPU 时间 + 系统 CPU 时间 = 1.9 秒。

图 7.3.1 中显示的时间总和并不是总的进程时间，前面也给大家解释过，这个时间总和指的是从起点到终点锁经过的时间，并不是进程时间，这里大家要理解。时间总和包括了进程处于休眠状态时消耗的时间（sleep 等会让进程挂起、进入休眠状态），可以发现时间总和比进程时间多 1 秒，其实这一秒就是进程处于休眠状态的时间。

7.3.2 clock 函数

库函数 clock()提供了一个更为简单的方式用于进程时间，它的返回值描述了进程使用的总的 CPU 时间（也就是进程时间，包括用户 CPU 时间和系统 CPU 时间），其函数原型如下所示：

```
#include <time.h>

clock_t clock(void);
```

使用该函数需要包含头文件<time.h>。

函数参数和返回值含义如下：

无参数。

返回值：返回值是到目前为止程序的进程时间，为 clock_t 类型，注意 clock()的返回值并不是系统节拍数，如果想要获得秒数，请除以 CLOCKS_PER_SEC（这是一个宏）。如果返回的进程时间不可用或其值无法表示，则该返回值是-1。

clock()函数虽然可以很方便的获取总的进程时间，但并不能获取到单独的用户 CPU 时间和系统 CPU 时间，在实际编程当中，根据自己的需要选择。

测试

对示例代码 7.3.2 进行简单地修改，使用 clock()获取到待测试代码段所消耗的进程时间，如下：

示例代码 7.3.3 clock 函数使用示例

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

int main(int argc, char *argv[])
{
    clock_t t_start;
    clock_t t_end;
    int i, j;

    /* 开始时间 */
    t_start = clock();
    if (-1 == t_start)
        exit(-1);

    /* *****需要进行测试的代码段***** */
    for (i = 0; i < 20000; i++)
        for (j = 0; j < 20000; j++)
            ;
    /* **************end*************** */

    /* 结束时间 */
    t_end = clock();
    if (-1 == t_end)
        exit(-1);
```

```
    /* 打印时间 */
    printf("总的 CPU 时间: %f\n", (t_end - t_start) / (double)CLOCKS_PER_SEC);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
总的CPU时间： 1.806213
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.3.2 测试结果

7.4 产生随机数

在应用编程当中可能会用到随机数，譬如老板让你编写一个抽奖的小程序，编号 0~100，分为特等奖 1 个、一等奖 2 个、二等级 3 以及三等级 4 个，也就是说需要从 0~100 个编号中每次随机抽取一个号码，这就需要用到随机数。那在 Linux 应用编程中如何去产生随机数呢？本小节就来学习生成随机数。

随机数与伪随机数

随机数是随机出现，没有任何规律的一组数列。在我们编程当中，是没有办法获得真正意义上的随机数列的，这是一种理想的情况，在我们的程序当中想要使用随机数列，只能通过算法得到一个伪随机数序列，那在编程当中说到的随机数，基本都是指伪随机数。

C 语言函数库中提供了很多函数用于产生伪随机数，其中最常用的是通过 rand()和 srand()产生随机数，本小节就以这两个函数为例向大家介绍如何在我们的程序中获得随机数列。

rand 函数

rand()函数用于获取随机数，多次调用 rand()可得到一组随机数序列，其函数原型如下：

```
#include <stdlib.h>

int rand(void);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

返回值：返回一个介于 0 到 RAND_MAX（包含）之间的值，也就是数学上的[0, RAND_MAX]。

程度当中调用 rand()可以得到[0, RAND_MAX]之间的伪随机数，多次调用 rand()便可以生成一组伪随机树序列，但是这里有个问题，就是每一次运行程序所得到的随机数序列都是相同的，那如何使得每一次启动应用程序所得到的随机数序列是不一样的呢？那就通过设置不同的随机数种子，可通过 srand()设置随机数种子。

如果没有调用 srand()设置随机数种子的情况下，rand()会将 1 作为随机数种子，如果随机数种子相同，那么每一次启动应用程序所得到的随机数序列就是一样的，所以每次启动应用程序需要设置不同的随机数种子，这样就可以使得程序每次运行所得到随机数序列不同。

srand 函数

使用 srand()函数为 rand()设置随机数种子，其函数原型如下所示：

```
#include <stdlib.h>

void srand(unsigned int seed);
```

函数参数和返回值含义如下：

seed：指定一个随机数中，int 类型的数据，一般尝尝将当前时间作为随机数种子赋值给参数 seed，譬如 time(NULL)，因为每次启动应用程序时间上是一样的，所以就能够使得程序中设置的随机数种子在每次启动程序时是不一样的。

返回值：void

常用的用法 srand(time(NULL));

测试

使用 rand()和 srand()产生一组伪随机数，数值范围为[0~100]，将其打印出来：

示例代码 7.4.1 生成随机数示例代码

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

int main(int argc, char *argv[])
{
    int random_number_arr[8];
    int count;

    /* 设置随机数种子 */
    srand(time(NULL));

    /* 生成伪随机数 */
    for (count = 0; count < 8; count++)
        random_number_arr[count] = rand() % 100;

    /* 打印随机数数组 */
    printf("[");
    for (count = 0; count < 8; count++) {
        printf("%d", random_number_arr[count]);
        if (count != 8 - 1)
            printf(", ");
    }

    printf("]\n");
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[48, 58, 25, 54, 67, 51, 96, 79]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[40, 35, 61, 55, 71, 86, 44, 72]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[46, 10, 1, 26, 84, 45, 76, 74]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
[61, 15, 68, 29, 87, 53, 3, 37]
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.4.1 测试结果

从图中可以发现，每一次得到的[0~100]之间的随机数数组都是不同的（数组不同，不是产生的随机数不同），因为程序中将 rand()的随机数种子设置为 srand(time(NULL))，直接等于 time_t 时间值，意味着每次启动种子都不一样，所以能够产生不同的随机数数组。

本小节关于在 Linux 下使用随机数就给大家介绍这么多，产生随机数的 API 函数并不仅仅只有这些，除此之外，譬如还有 random()、srandom()、initstate()、setstate()等，这里便不再给大家一一介绍了，在我们使用 man 手册查看系统调用或 C 库函数帮助信息时，在帮助信息页面 SEE ALSO 栏会列举出与本函数有关联的一些命令、系统调用或 C 库函数等，如下所示（譬如执行 man 3 srand 查看）：

```
SEE ALSO
       drand48(3), random(3)

COLOPHON
       This page is part of release 4.04 of the Linux man-pages project.  A descripti
       be found at http://www.kernel.org/doc/man-pages/.
```

图 7.4.2 see also

7.5 休眠

有时需要将进程暂停或休眠一段时间，进入休眠状态之后，程序将暂停运行，直到休眠结束。常用的系统调用和 C 库函数有 sleep()、usleep()以及 nanosleep()，这些函数在应用程序当中通常作为延时使用，譬如延时 1 秒钟，本小节将一一介绍。

7.5.1 秒级休眠: sleep

sleep()是一个 C 库函数，从函数名字面意思便可以知道该函数的作用了，简单地说，sleep()就是让程序“休息”一会，然后再继续工作。其函数原型如下所示：

```
#include <unistd.h>


unsigned int sleep(unsigned int seconds);
```

使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

seconds：休眠时长，以秒为单位。

返回值：如果休眠时长为参数 seconds 所指定的秒数，则返回 0；若被信号中断则返回剩余的秒数。

sleep()是一个秒级别休眠函数，程序在休眠过程中，是可以被其它信号所打断的，关于信号这些内容，将会在后面章节向大家介绍。

测试

编写一个简单地程序，调用 sleep()函数让程序暂停运行（休眠）3 秒钟。

示例代码 7.5.1 sleep 函数使用示例

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

int main(void)
{
    puts("Sleep Start!");

    /* 让程序休眠 3 秒钟 */
    sleep(3);

    puts("Sleep End!");
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Sleep Start!
Sleep End!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.5.1 sleep 测试结果

7.5.2 微秒级休眠: usleep

usleep()同样也是一个 C 库函数，与 sleep()的区别在于休眠时长精度不同，usleep()支持微秒级程序休眠，其函数原型如下所示：

```
#include <unistd.h>

int usleep(useconds_t usec);
```

函数参数和返回值含义如下：

usec：休眠时长，以微秒为单位。

返回值：成功返回 0；失败返回-1，并设置 errno。

测试

使用 usleep()函数让程序休眠 3 秒钟。

示例代码 7.5.2 usleep 函数使用示例

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

int main(void)
{
    puts("Sleep Start!");
```

```
    /* 让程序休眠 3 秒钟(3*1000*1000 微秒) */
    usleep(3 * 1000 * 1000);

    puts("Sleep End!");
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Sleep Start!
Sleep End!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.5.2 usleep 休眠

7.5.3 高精度休眠: nanosleep

nanosleep()与 sleep()以及 usleep()类似，都用于程序休眠，但 nanosleep()具有更高精度来设置休眠时间长度，支持纳秒级时长设置。与 sleep()、usleep()不同的是，nanosleep()是一个 Linux 系统调用，其函数原型如下所示：

```
#include <time.h>

int nanosleep(const struct timespec *req, struct timespec *rem);
```

使用该函数需要包含头文件<time.h>。

函数参数与返回值含义如下：

req： 一个 struct timespec 结构体指针，指向一个 struct timespec 变量，用于设置休眠时间长度，可精确到纳秒级别。

rem： 也是一个 struct timespec 结构体指针，指向一个 struct timespec 变量，也可设置 NULL。

返回值： 在成功休眠达到请求的时间间隔后，nanosleep()返回 0；如果中途被信号中断或遇到错误，则返回-1，并将剩余时间记录在参数 rem 指向的 struct timespec 结构体变量中（参数 rem 不为 NULL 的情况下，如果为 NULL 表示不接收剩余时间），还会设置 errno 标识错误类型。

在 5.2.3 小节中介绍了 struct timespec 结构体，该结构体包含了两个成员变量，秒（tv_sec）和纳秒（tv_nsec），具体定义可参考示例代码 5.2.2。

测试

示例代码 7.5.3 nanosleep 函数使用示例

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

int main(void)
{
    struct timespec request_t;
```

```
    puts("Sleep Start!");

    /* 让程序休眠 3 秒钟 */
    request_t.tv_sec = 3;
    request_t.tv_nsec = 0;
    nanosleep(&request_t, NULL);

    puts("Sleep End!");
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Sleep Start!
Sleep End!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.5.3 nanosleep 休眠

前面说到，在应用程序当中，通常使用这些函数作为延时功能，譬如在程序当中需要延时一秒钟、延时 5 毫秒等应用场景时，那么就可以使用这些函数来实现；但是大家需要注意，休眠状态下，该进程会失去 CPU 使用权，退出系统调度队列，直到休眠结束。在一个裸机程序当中，通常使用 for 循环（或双重 for 循环）语句来实现延时等待，譬如在 for 循环当中执行 nop 空指令，也就意味着即使在延时等待情况下，CPU 也是一直都在工作；由此可知，应用程序当中使用休眠用作延时功能，并不是裸机程序中的 nop 空指令延时，一旦执行 sleep()，进程便主动交出 CPU 使用权，暂时退出系统调度队列，在休眠结束前，该进程的指令将得不到执行。

7.6 申请堆内存

在操作系统下，内存资源是由操作系统进行管理、分配的，当应用程序想要内存时（这里指的是堆内存），可以向操作系统申请内存，然后使用内存；当不再需要时，将申请的内存释放、归还给操作系统；在许多的应用程序当中，往往都会有这种需求，譬如为一些数据结构动态分配/释放内存空间，本小节向大家介绍应用程序如何向操作系统申请堆内存。

7.6.1 在堆上分配内存：malloc 和 free

Linux C 程序当中一般使用 malloc()函数为程序分配一段堆内存，而使用 free()函数来释放这段内存，先来看下 malloc()函数原型，如下所示：

```
#include <stdlib.h>

void *malloc(size_t size);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

size：需要分配的内存大小，以字节为单位。

返回值：返回值为 void *类型，如果申请分配内存成功，将返回一个指向该段内存的指针，void *并不是说没有返回值或者返回空指针，而是返回的指针类型未知,所以在调用 malloc()时通常需要进行强制类型转换，将 void *指针类型转换成我们希望的类型；如果分配内存失败（譬如系统堆内存不足）将返回 NULL，如果参数 size 为 0，返回值也是 NULL。

malloc()在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的，所以通常需要程序员对 malloc()分配的堆内存进行初始化操作。

在堆上分配的内存，需要开发者自己手动释放掉，通常使用 free()函数释放堆内存，free()函数原型如下所示：

```
#include <stdlib.h>

void free(void *ptr);
```

使用该函数同样需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

ptr：指向需要被释放的堆内存对应的指针。

返回值：无返回值。

测试

示例代码 7.6.1 malloc()和 free()申请/释放堆内存

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

#define    MALLOC_MEM_SIZE    (1 * 1024 * 1024)

int main(int argc, char *argv[])
{
    char *base = NULL;

    /* 申请堆内存 */
    base = (char *)malloc(MALLOC_MEM_SIZE);
    if (NULL == base) {
        printf("malloc error\n");
        exit(-1);
    }

    /* 初始化申请到的堆内存 */
    memset(base, 0x0, MALLOC_MEM_SIZE);

    /* 使用内存 */
    /* ..... */

    /* 释放内存 */
    free(base);
```

```
    exit(0);
}
```

调用 free()还是不调用 free()

在学习文件 IO 基础章节内容时曾向大家介绍过，Linux 系统中，当一个进程终止时，内核会自动关闭它没有关闭的所有文件（该进程打开的文件，但是在进程终止时未调用 close()关闭它）。同样，对于内存来说，也是如此！当进程终止时，内核会将其占用的所有内存都返还给操作系统，这包括在堆内存中由 malloc()函数所分配的内存空间。基于内存的这一自动释放机制，很多应用程序通常会省略对 free()函数的调用。

这在程序中分配了多块内存的情况下可能会特别有用，因为加入多次对 free()的调用不但会消耗品大量的 CPU 时间，而且可能会使代码趋于复杂。

虽然依靠终止进程来自动释放内存对大多数程序来说是可以接受的，但最好能够在程序中显式调用 free()释放内存，首先其一，显式调用 free()能使程序具有更好的可读性和可维护性；其二，对于很多程序来说，申请的内存并不是在程序的生命周期中一直需要，大多数情况下，都是根据代码需求动态申请、释放的，如果申请的内存对程序来说已经不再需要了，那么就已经把它释放、归还给操作系统，如果持续占用，将会导致内存泄漏，也就是人们常说的“你的程序在吃内存”！

7.6.2 在堆上分配内存的其它方法

除了 malloc()外，C 函数库中还提供了一系列在堆上分配内存的其它函数，本小节将逐一介绍。

用 calloc()分配内存

calloc()函数用来动态地分配内存空间并初始化为 0，其函数原型如下所示：

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
```

使用该函数同样也需要包含头文件<stdlib.h>。

calloc()在堆中动态地分配 nmemb 个长度为 size 的连续空间，并将每一个字节都初始化为 0。所以它的结果是分配了 nmemb * size 个字节长度的内存空间，并且每个字节的值都是 0。

返回值：分配成功返回指向该内存的地址，失败则返回 NULL。

calloc()与 malloc()的一个重要区别是：calloc()在动态分配完内存后，自动初始化该内存空间为零，而 malloc()不初始化，里边数据是未知的垃圾数据。下面的两种写法是等价的：

```
// calloc()分配内存空间并初始化
char *buf1 = (char *)calloc(10, 2);

// malloc()分配内存空间并用 memset()初始化
char *buf2 = (char *)malloc(10 * 2);
memset(buf2, 0, 20);
```

测试

编写测试代码，将用户输入的一组数字存放到堆内存中，并打印出来。

示例代码 7.6.2 calloc 函数使用示例

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

int main(int argc, char *argv[])
{
```

```
    int *base = NULL;
    int i;

    /* 校验传参 */
    if (2 > argc)
        exit(-1);

    /* 使用 calloc 申请内存 */
    base = (int *)calloc(argc - 1, sizeof(int));
    if (NULL == base) {
        printf("calloc error\n");
        exit(-1);
    }

    /* 将字符串转为 int 型数据存放在 base 指向的内存中 */
    for (i = 0; i < argc - 1; i++)
        base[i] = atoi(argv[i+1]);

    /* 打印 base 数组中的数据 */
    printf("你输入的数据是: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", base[i]);
    putchar('\n');

    /* 释放内存 */
    free(base);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 1020 154545 545 201122 7845
你输入的数据是: 1020 154545 545 201122 7845
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 1 2 3 4 5
你输入的数据是: 1 2 3 4 5
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 100 200 300 500
你输入的数据是: 100 200 300 500
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 7.6.1 测试结果

7.6.3 分配对其内存

C 函数库中还提供了一系列在堆上分配对齐内存的函数，对齐内存在某些应用场合非常有必要，常用于分配对其内存的库函数有：posix_memalign()、aligned_alloc()、memalign()、valloc()、pvalloc()，它们的函数原型如下所示：

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

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
void *aligned_alloc(size_t alignment, size_t size);
void *valloc(size_t size);

#include <malloc.h>

void *memalign(size_t alignment, size_t size);
void *pvalloc(size_t size);
```

使用 posix_memalign()、aligned_alloc()、valloc()这三个函数时需要包含头文件<stdlib.h>，而使用 memalign()、pvalloc()这两个函数时需要包含头文件<malloc.h>。前面介绍的 malloc()、calloc()分配内存返回的地址其实也是对齐的，但是它俩的对齐都是固定的，并且对其的字节边界比较小，譬如在 32 位系统中，通常是以 8 字节为边界进行对其，在 64 位系统中是以 16 字节进行对其。如果想实现更大字节的对齐，则需要使用本小节介绍的函数。

posix_memalign()函数

posix_memalign()函数用于在堆上分配 size 个字节大小的对齐内存空间，将*memptr 指向分配的空间，分配的内存地址将是参数 alignment 的整数倍。参数 alignment 表示对齐字节数，alignment 必须是 2 的幂次方（譬如 2^4、2^5、2^8 等），同时也要是 sizeof(void *)的整数倍，对于 32 位系统来说，sizeof(void *)等于 4，如果是 64 位系统 sizeof(void *)等于 8。

函数参数和返回值含义如下：

memptr：void **类型的指针，内存申请成功后会将分配的内存地址存放在*memptr 中。

alignment：设置内存对其的字节数，alignment 必须是 2 的幂次方（譬如 2^4、2^5、2^8 等），同时也要是 sizeof(void *)的整数倍。

size：设置分配的内存大小，以字节为单位，如果参数 size 等于 0，那么*memptr 中的值是 NULL。

返回值：成功将返回 0；失败返回非 0 值。

示例代码

示例代码 7.6.3 posix_memalign 函数使用示例

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

int main(int argc, char *argv[])
{
    int *base = NULL;
    int ret;

    /* 申请内存: 256 字节对齐 */
    ret = posix_memalign((void **)&base, 256, 1024);
    if (0 != ret) {
        printf("posix_memalign error\n");
        exit(-1);
    }
```

```
    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

aligned_alloc()函数

aligned_alloc()函数用于分配 size 个字节大小的内存空间，返回指向该空间的指针。

函数参数和返回值含义如下：

alignment：用于设置对齐字节大小，alignment 必须是 2 的幂次方（譬如 2^4、2^5、2^8 等）。

size：设置分配的内存大小，以字节为单位。参数 size 必须是参数 alignment 的整数倍。

返回值：成功将返回内存空间的指针，内存空间的起始地址是参数 alignment 的整数倍；失败返回 NULL。

使用示例

示例代码 7.6.4 aligned_alloc 函数使用示例

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

int main(int argc, char *argv[])
{
    int *base = NULL;

    /* 申请内存: 256 字节对齐 */
    base = (int *)aligned_alloc(256, 256 * 4);
    if (base == NULL) {
        printf("aligned_alloc error\n");
        exit(-1);
    }

    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

memalign()函数

memalign()与 aligned_alloc()参数是一样的，它们之间的区别在于：对于参数 size 必须是参数 alignment 的整数倍这个限制条件，memalign()并没有这个限制条件。

Tips：memalign()函数已经过时了，并不提倡使用！

使用示例

示例代码 7.6.5 memalign 函数使用示例

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

int main(int argc, char *argv[])
{
    int *base = NULL;

    /* 申请内存: 256 字节对齐 */
    base = (int *)memalign(256, 1024);
    if (base == NULL) {
        printf("memalign error\n");
        exit(-1);
    }

    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

valloc()函数

valloc()分配 size 个字节大小的内存空间，返回指向该内存空间的指针，内存空间的地址是页大小（pagesize）的倍数。

valloc()与 memalign()类似，只不过 valloc()函数内部实现中，使用了页大小作为对齐的长度，在程序当中，可以通过系统调用 getpagesize()来获取内存的页大小。

Tips：valloc()函数已经过时了，并不提倡使用！

使用示例

示例代码 7.6.6 valloc 函数使用示例

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

```
int main(int argc, char *argv[])
{
    int *base = NULL;

    /* 申请内存: 1024 个字节 */
    base = (int *)valloc(1024);
    if (base == NULL) {
        printf("valloc error\n");
        exit(-1);
    }

    /* 使用内存 */
    // base[0] = 0;
    // base[1] = 1;
    // base[2] = 2;
    // base[3] = 3;

    /* 释放内存 */
    free(base);
    exit(0);
}
```

7.7 proc 文件系统

proc 文件系统是一个虚拟文件系统，它以文件系统的方式为应用层访问系统内核数据提供了接口，用户和应用程序可以通过 proc 文件系统得到系统信息和进程相关信息，对 proc 文件系统的读写作为与内核进行通信的一种手段。但是与普通文件不同的是，proc 文件系统是动态创建的，文件本身并不存在于磁盘当中、只存在于内存当中,与 devfs 一样，都被称为虚拟文件系统。

最初构建 proc 文件系统是为了提供有关系统中进程相关的信息，但是由于这个文件系统非常有用，因此内核中的很多信息也开始使用它来报告，或启用动态运行时配置。内核构建 proc 虚拟文件系统，它会将内核运行时的一些关键数据信息以文件的方式呈现在 proc 文件系统下的一些特定文件中，这样相当于将一些不可见的内核中的数据结构以可视化的方式呈现给应用层。

proc 文件系统挂载在系统的/proc 目录下，对于内核开发者（譬如驱动开发工程师）来说，proc 文件系统给了开发者一种调试内核的方法：通过查看/proc/xxx 文件来获取到内核特定数据结构的值，在添加了新功能前后进行对比，就可以判断此功能所产生的影响是否合理。

/proc 目录下中包含了一些目录和虚拟文件，如下所示：

```
dt@dt-virtual-machine:/proc$ pwd
/proc
dt@dt-virtual-machine:/proc$ ls
1       13    1996  2137  2236   234   2461  26    31    37     44   74898  990          kallsyms      self
10      133   2     214   224    2346  247   260   312   37425  45   8      99686        kcore         slabinfo
100038  14    20    2141  225    235   2472  261   3160  37426  46   885    acpi         keys          softirqs
1002    1490  2008  215   2250   236   2476  262   3165  37431  495  887    asound       key-users     stat
100513  15    2029  2158  2255   237   248   263   317   37432  50   888    buddyinfo    kmsg          swaps
101898  152   2030  216   226    2373  249   264   3179  37433  51   891    bus          kpagecgroup   sys
101924  1524  2035  2169  22627  238   25    265   3183  37434  52   89334  cgroups      kpagecount    sysrq-trigger
101994  1560  2056  217   227    239   250   266   3184  37435  53   9      cmdline      kpageflags    sysvipc
102033  1591  206   2170  228    2393  2501  267   32    37436  54   90101  consoles     loadavg       thread-self
102039  16    2068  2173  229    24    251   268   3208  37437  55   90102  cpuinfo      locks         timer_list
1046    1607  2069  2176  2294   240   2513  269   3222  37438  56   904    crypto       mdstat        tty
1057    1612  207   2179  2295   2403  252   27    3235  378    57   905    devices      meminfo       uptime
1071    1621  2073  218   2298   241   253   270   3271  38     58   906    diskstats    misc          version
11      1716  2081  2180  2299   2410  254   271   33    387    6    917    dma          modules       version_signature
113     1735  2086  2182  230    2416  255   272   3304  39     62   92540  driver       mounts        vmallocinfo
114     1739  21    2184  2304   242   2550  273   3315  4      63   93945  execdomains  mpt           vmstat
115     18    2100  219   2306   2421  2554  274   3330  40     64   943    fb           mtrr          zoneinfo
116     19    2104  22    231    243   256   28    3372  400    65   98557  filesystems  net
117     1903  2115  220   232    2430  2564  2813  34    405    66   98560  fs           pagetypeinfo
118     1904  2117  2209  2320   244   2568  290   3403  406    69   98562  interrupts   partitions
12      1909  213   221   2322   2440  257   292   351   42     7    98563  iomem        sched_debug
1229    1911  2130  222   233    245   258   30    352   43     70   98564  ioports      schedstat
124     1993  2131  223   2332   246   259   3090  36    438    71   98639  irq          scsi
dt@dt-virtual-machine:/proc$
```

图 7.7.1 proc 文件系统下的目录和文件

可以看到/proc 目录下有很多以数字命名的文件夹，譬如 100038、2299、98560，这些数字对应的其实就是一个一个的进程 PID 号，每一个进程在内核中都会存在一个编号，通过此编号来区分不同的进程，这个编号就是 PID 号，关于 PID、以及进程相关的信息将会在后面章节内容中向大家介绍。

所以这些以数字命名的文件夹中记录了这些进程相关的信息，不同的信息通过不同的虚拟文件呈现出来，关于这些信息将会在后面章节内容中向大家介绍。

/proc 目录下除了文件夹之外，还有很多的虚拟文件，譬如 buddyinfo、cgroups、cmdline、version 等等，不同的文件记录了不同信息，关于这些文件记录的信息和意思如下：

- cmdline：内核启动参数；
- cpuinfo：CPU 相关信息；
- iomem：IO 设备的内存使用情况；
- interrupts：显示被占用的中断号和占用者相关的信息；
- ioports：IO 端口的使用情况；
- kcore：系统物理内存映像，不可读取；
- loadavg：系统平均负载；
- meminfo：物理内存和交换分区使用情况；
- modules：加载的模块列表；
- mounts：挂载的文件系统列表；
- partitions：系统识别的分区表；
- swaps：交换分区的利用情况；
- version：内核版本信息；
- uptime：系统运行时间；

7.7.1 proc 文件系统的使用

proc 文件系统的使用就是去读取/proc 目录下的这些文件，获取文件中记录的信息，可以直接使用 cat 命令读取，也可以在应用程序中调用 open()打开、然后再使用 read()函数读取。

使用 cat 命令读取

在 Linux 系统下直接使用 cat 命令查看/proc 目录下的虚拟文件，譬如"cat /proc/version"查看内核版本相关信息：

```
dt@dt-virtual-machine:~$ cat /proc/version
Linux version 4.15.0-132-generic (buildd@lgw01-amd64-034) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #136~16.04.1-Ubuntu SMP Tue Jan 12 18:22:20 UTC 2021
dt@dt-virtual-machine:~$
```

图 7.7.2 cat 命令查看/proc 目录下的文件

使用 read()函数读取

编写一个简单地程序，使用 read()函数读取/proc/version 文件。

示例代码 7.7.1 应用程序中读取 proc 文件系统

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

int main(int argc, char *argv[])
{
    char buf[512] = {0};
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("/proc/version", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 读取文件 */
    ret = read(fd, buf, sizeof(buf));
    if (-1 == ret) {
        perror("read error");
        exit(-1);
    }

    /* 打印信息 */
    puts(buf);

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

运行结果：


dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Linux version 4.15.0-132-generic (buildd@lgw01-amd64-034) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #136~16.04.1-Ubuntu SMP Tue Jan 12 18:22:20 UTC 2021
dt@dt-virtual-machine:~/vscode_ws/2_chapter$

图 7.7.3 测试结果

第八章 信号：基础

本章将讨论信号，虽然信号的基本概念比较简单，但是其所涉及到的细节内容比较多，所以本章篇幅也会相对比较长。事实上，在很多应用程序当中，都会存在处理异步事件这种需求，而信号提供了一种处理异步事件的方法，所以信号机制在 Linux 早期版本中就已经提供了支持，随着 Linux 内核版本的更新迭代，其对信号机制的支持更加完善。

本章将会讨论如下主题内容。

- 信号的基本概念；
- 信号的分类、Linux 提供的各种不同的信号及其作用；
- 发出信号以及响应信号，信号由“谁”发送、由“谁”处理以及如何处理；
- 进程在默认情况下对信号的响应方式；
- 使用进程信号掩码来阻塞信号、以及等待信号等相关概念；
- 如何暂停进程的执行，并等待信号的到达。

8.1 基本概念

信号是事件发生时对进程的通知机制，也可以把它称为软件中断。信号与硬件中断的相似之处在于能够打断程序当前执行的正常流程，其实是在软件层次上对中断机制的一种模拟。大多数情况下，是无法预测信号达到的准确时间，所以，信号提供了一种处理异步事件的方法。

信号的目的是用来通信的

一个具有合适权限的进程能够向另一个进程发送信号，信号的这一用法可作为一种同步技术，甚至是进程间通信（IPC）的原始形式。信号可以由“谁”发出呢？以下列举的很多情况均可以产生信号：

- 硬件发生异常，即硬件检测到错误条件并通知内核，随即再由内核发送相应的信号给相关进程。硬件检测到异常的例子包括执行一条异常的机器语言指令，诸如，除数为 0、数组访问越界导致引用了无法访问的内存区域等，这些异常情况都会被硬件检测到，并通知内核、然后内核为该异常情况发生时正在运行的进程发送适当的信号以通知进程。
- 用于在终端下输入了能够产生信号的特殊字符。譬如在终端上按下 CTRL + C 组合按键可以产生中断信号（SIGINT），通过这个方法可以终止在前台运行的进程；按下 CTRL + Z 组合按键可以产生暂停信号（SIGCONT），通过这个方法可以暂停当前前台运行的进程。
- 进程调用 kill()系统调用可将任意信号发送给另一个进程或进程组。当然对此是有所限制的，接收信号的进程和发送信号的进程的所有者必须相同，亦或者发送信号的进程的所有者是 root 超级用户。
- 用户可以通过 kill 命令将信号发送给其它进程。kill 命令想必大家都会使用，通常我们会通过 kill 命令来“杀死”（终止）一个进程，譬如在终端下执行"kill -9 xxx"来杀死 PID 为 xxx 的进程。kill 命令其内部的实现原理便是通过 kill()系统调用来完成的。
- 发生了软件事件，即当检测到某种软件条件已经发生。这里指的不是硬件产生的条件（如除数为 0、引用无法访问的内存区域等），而是软件的触发条件、触发了某种软件条件（进程所设置的定时器已经超时、进程执行的 CPU 时间超限、进程的某个子进程退出等等情况）。

进程同样也可以向自身发送信号，然而发送给进程的诸多信号中，大多数都是来自于内核。

以上便是可以产生信号的多种不同的条件，总的来看，信号的目的都是用于通信的，当发生某种情况下，通过信号将情况“告知”相应的进程，从而达到同步、通信的目的。

信号由谁处理、怎么处理

信号通常是发送给对应的进程，当信号到达后，该进程需要做出相应的处理措施，通常进程会视具体信号执行以下操作之一：

- 忽略信号。也就是说，当信号到达进程后，该进程并不会去理会它、直接忽略，就好像是没有出该信号，信号对该进程不会产生任何影响。事实上，大多数信号都可以使用这种方式进行处理，但有两种信号却决不能被忽略，它们是 SIGKILL 和 SIGSTOP，这两种信号不能被忽略的原因是：它们向内核和超级用户提供了使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号，则进程的运行行为是未定义的。
- 捕获信号。当信号到达进程后，执行预先绑定好的信号处理函数。为了做到这一点，要通知内核在某种信号发生时，执行用户自定义的处理函数，该处理函数中将会对该信号事件作出相应的处理，Linux 系统提供了 signal()系统调用可用于注册信号的处理函数，将会在后面向大家介绍。
- 执行系统默认操作。进程不对该信号事件作出处理，而是交由系统进行处理，每一种信号都会有其对应的系统默认的处理方式，8.3 小节中对此有进行介绍。需要注意的是，对大多数信号来说，系统默认的处理方式就是终止该进程。

信号是异步的

信号是异步事件的经典实例，产生信号的事件对进程而言是随机出现的，进程无法预测该事件产生的准确时间，进程不能够通过简单地测试一个变量或使用系统调用来判断是否产生了一个信号，这就如同硬件中断事件，程序是无法得知中断事件产生的具体时间，只有当产生中断事件时，才会告知程序、然后打断当前程序的正常执行流程、跳转去执行中断服务函数，这就是异步处理方式。

信号本质上是 int 类型数字编号

信号本质上是 int 类型的数字编号，这就好比硬件中断所对应的中断号。内核针对每个信号，都给其定义了一个唯一的整数编号，从数字 1 开始顺序展开。并且每一个信号都有其对应的名字（其实就是一个宏），信号名字与信号编号乃是一一对应关系，但是由于每个信号的实际编号随着系统的不同可能会不一样，所以在程序当中一般都使用信号的符号名（也就是宏定义）。

这些信号在<signum.h>头文件中定义，每个信号都是以 SIGxxx 开头，如下所示：

示例代码 8.1.1 信号定义

```
/* Signals.  */
#define SIGHUP      1       /* Hangup (POSIX).  */
#define SIGINT      2       /* Interrupt (ANSI).  */
#define SIGQUIT     3       /* Quit (POSIX).  */
#define SIGILL      4       /* Illegal instruction (ANSI).  */
#define SIGTRAP     5       /* Trace trap (POSIX).  */
#define SIGABRT     6       /* Abort (ANSI).  */
#define SIGIOT      6       /* IOT trap (4.2 BSD).  */
#define SIGBUS      7       /* BUS error (4.2 BSD).  */
#define SIGFPE      8       /* Floating-point exception (ANSI).  */
#define SIGKILL     9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1     10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV     11      /* Segmentation violation (ANSI).  */
#define SIGUSR2     12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE     13      /* Broken pipe (POSIX).  */
#define SIGALRM     14      /* Alarm clock (POSIX).  */
#define SIGTERM     15      /* Termination (ANSI).  */
#define SIGSTKFLT   16      /* Stack fault.  */
#define SIGCLD      SIGCHLD     /* Same as SIGCHLD (System V).  */
#define SIGCHLD     17      /* Child status has changed (POSIX).  */
#define SIGCONT     18      /* Continue (POSIX).  */
#define SIGSTOP     19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP     20      /* Keyboard stop (POSIX).  */
#define SIGTTIN     21      /* Background read from tty (POSIX).  */
#define SIGTTOU     22      /* Background write to tty (POSIX).  */
#define SIGURG      23      /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU     24      /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ     25      /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM   26      /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF     27      /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH    28      /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL     SIGIO   /* Pollable event occurred (System V).  */
```

```
#define SIGIO       29      /* I/O now possible (4.2 BSD).    */
#define SIGPWR      30      /* Power failure restart (System V).    */
#define SIGSYS      31      /* Bad system call.    */
#define SIGUNUSED   31
```

不存在编号为 0 的信号，从示例代码 8.1.1 中也可以看到，信号编号是从 1 开始的，事实上 kill()函数对信号编号 0 有着特殊的应用，关于这个文件将会在后面的内容向大家介绍。

8.2 信号的分类

Linux 系统下可对信号从两个不同的角度进行分类，从可靠性方面将信号分为可靠信号与不可靠信号；而从实时性方面将信号分为实时信号与非实时信号，本小节将对这些信号的分类进行简单地介绍。

8.2.1 可靠信号与不可靠信号

Linux 信号机制基本上是从 UNIX 系统中继承过来的，早期 UNIX 系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，它的主要问题是：

- 进程每次处理信号后，就将对信号的响应设置为系统默认操作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用 signal()，重新为该信号绑定相应的处理函数。
- 因此导致，早期 UNIX 下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失（处理信号时又来了新的信号，则导致信号丢失）。

Linux 支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用 signal()。因此，Linux 下的不可靠信号问题主要指的是信号可能丢失。在 Linux 系统下，信号值小于 SIGRTMIN（34）的信号都是不可靠信号，这就是"不可靠信号"的来源，所以示例代码 8.1.1 中所列举的信号都是不可靠信号。

随着时间的发展，实践证明，有必要对信号的原始机制加以改进和扩充，所以，后来出现的各种 UNIX 版本分别在这方面进行了研究，力图实现"可靠信号"。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号(SIGRTMIN~SIGRTMAX)，并在一开始就把它们定义为可靠信号，在 Linux 系统下使用"kill -l"命令可查看到所有信号，如下所示：

```
dt@dt-virtual-machine:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
dt@dt-virtual-machine:~$
```

图 8.2.1 kill 命令查看所有信号

Tips：括号")"前面的数字对应该信号的编号，编号 1~31 所对应的是不可靠信号，编号 34~64 对应的是可靠信号，从图中可知，可靠信号并没有一个具体对应的名字，而是使用了 SIGRTMIN+N 或 SIGRTMAX-N 的方式来表示。

可靠信号支持排队，不会丢失，同时，信号的发送和绑定也出现了新版本，信号发送函数 sigqueue()及信号绑定函数 sigaction()。

早期 UNIX 系统只定义了 31 种信号，而 Linux 3.x 支持 64 种信号，编号 1-64(SIGRTMIN=34，SIGRTMAX=64)，将来可能进一步增加，这需要得到内核的支持。前 31 种信号已经有了预定义值，每个信号有了确定的用途、含义以及对应的名字，并且每种信号都有各自的系统默认操作。如按键盘的 CTRL+C 时，会产生 SIGINT 信号，对该信号的系统默认操作就是终止进程，后 32 个信号表示可靠信号。

8.2.2 实时信号与非实时信号

实时信号与非实时信号其实是从时间关系上进行的分类，与可靠信号与不可靠信号是相互对应的，非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号。实时信号保证了发送的多个信号都能被接收，实时信号是 POSIX 标准的一部分，可用于应用进程。

一般我们也把非实时信号（不可靠信号）称为标准信号，如果文档中用到了这个词，那么大家要知道，这里指的就是非实时信号（不可靠信号）。关于更多实时信号相关内容将会在 8.10 小节中介绍。

8.3 常见信号与默认行为

前面说到，Linux 下对标准信号（不可靠信号、非实时信号）的编号为 1~31，如示例代码 8.1.1 所示，接下来将介绍这些信号以及这些信号所对应的系统默认操作。

- **SIGINT**

当用户在终端按下中断字符（通常是 CTRL + C）时，内核将发送 SIGINT 信号给前台进程组中的每一个进程。该信号的系统默认操作是终止进程的运行。所以通常我们都会使用 CTRL + C 来终止一个占用前台的进程，原因在于大部分的进程会将该信号交给系统去处理，从而执行该信号的系统默认操作。

- **SIGQUIT**

当用户在终端按下退出字符（通常是 CTRL + \）时，内核将发送 SIGQUIT 信号给前台进程组中的每一个进程。该信号的系统默认操作是终止进程的运行、并生成可用于调试的核心转储文件。进程如果陷入无限循环、或不再响应时，使用 SIGQUIT 信号就很合适。所以对于一个前台进程，既可以在终端按下中断字符 CTRL + C、也可以按下退出字符 CTRL + \来终止，当然前提条件是，此进程会将 SIGINT 信号或 SIGQUIT 信号交给系统处理（也就是没有将信号忽略或捕获），进入执行该信号所对应的系统默认操作。

- **SIGILL**

如果进程试图执行非法（即格式不正确）的机器语言指令，系统将向进程发送该信号。该信号的系统默认操作是终止进程的运行。

- **SIGABRT**

当进程调用 abort()系统调用时（进程异常终止），系统会向该进程发送 SIGABRT 信号。该信号的系统默认操作是终止进程、并生成核心转储文件。

- **SIGBUS**

产生该信号（总线错误，bus error）表示发生了某种内存访问错误。该信号的系统默认操作是终止进程。

- **SIGFPE**

该信号因特定类型的算术错误而产生，譬如除以 0。该信号的系统默认操作是终止进程。

- **SIGKILL**

此信号为“必杀（sure kill）”信号，用于杀死进程的终极办法，此信号无法被进程阻塞、忽略或者捕获，故而“一击必杀”，总能终止进程。使用 SIGINT 信号和 SIGQUIT 信号虽然能终止进程，但是前提条件是该进程并没有忽略或捕获这些信号，如果使用 SIGINT 或 SIGQUIT 无法终止进程，那就使用“必杀信号”SIGKILL 吧。Linux 下有一个 kill 命令，kill 命令可用于向进程发送信号，我们会使用"kill -9 xxx"命令来终止一个进程（xxx 表示进程的 pid），这里的-9 其实指的就是发送编号为 9 的信号，也就是 SIGKILL 信号。

- **SIGUSR1**

该信号和 SIGUSR2 信号供程序员自定义使用，内核绝不会为进程产生这些信号，在我们的程序中，可以使用这些信号来互通通知事件的发生，或是进程彼此同步操作。该信号的系统默认操作是终止进程。

- **SIGSEGV**

这一信号非常常见，当应用程序对内存的引用无效时，操作系统就会向该应用程序发送该信号。引起对内存无效引用的原因很多，C 语言中引发这些事件往往是解引用的指针里包含了错误地址（譬如，未初始化的指针），或者传递了一个无效参数供函数调用等。该信号的系统默认操作是终止进程。

- **SIGUSR2**

与 SIGUSR1 信号相同。

- **SIGPIPE**

涉及到管道和 socket，当进程向已经关闭的管道、FIFO 或套接字写入信息时，那么系统将发送该信号给进程。该信号的系统默认操作是终止进程。

- **SIGALRM**

与系统调用 alarm()或 setitimer()有关，应用程序中可以调用 alarm()或 setitimer()函数来设置一个定时器，当定时器定时时间到，那么内核将会发送 SIGALRM 信号给该应用程序，关于 alarm()或 setitimer()函数的使用，后面将会进行讲解。该信号的系统默认操作是终止进程。

- **SIGTERM**

这是用于终止进程的标准信号，也是 kill 命令所发送的默认信号（kill xxx，xxx 表示进程 pid），有时我们会直接使用"kill -9 xxx"显式向进程发送 SIGKILL 信号来终止进程，然而这一做法通常是错误的，精心设计的应用程序应该会捕获 SIGTERM 信号、并为其绑定一个处理函数，当该进程收到 SIGTERM 信号时，会在处理函数中清除临时文件以及释放其它资源，再而退出程序。如果直接使用 SIGKILL 信号终止进程，从而跳过了 SIGTERM 信号的处理函数，通常 SIGKILL 终止进程是不友好的方式、是暴力的方式，这种方式应该作为最后手段，应首先尝试使用 SIGTERM，实在不行再使用最后手段 SIGKILL。

- **SIGCHLD**

当父进程的某一个子进程终止时，内核会向父进程发送该信号。当父进程的某一个子进程因收到信号而停止或恢复时，内核也可能向父进程发送该信号。注意这里说的停止并不是终止，你可以理解为暂停。该信号的系统默认操作是忽略此信号，如果父进程希望被告知其子进程的这种状态改变，则应捕获此信号。

- **SIGCLD**

与 SIGCHLD 信号同义。

- **SIGCONT**

将该信号发送给已停止的进程，进程将会恢复运行。当进程接收到此信号时并不处于停止状态，系统默认操作是忽略该信号，但如果进程处于停止状态，则系统默认操作是使该进程继续运行。

- **SIGSTOP**

这是一个“必停”信号，用于停止进程（注意停止不是终止，停止只是暂停运行、进程并没有终止），应用程序无法将该信号忽略或者捕获，故而总能停止进程。

- **SIGTSTP**

这也是一个停止信号，当用户在终端按下停止字符（通常是 CTRL + Z），那么系统会将 SIGTSTP 信号发送给前台进程组中的每一个进程，使其停止运行。

- **SIGXCPU**

当进程的 CPU 时间超出对应的资源限制时，内核将发送此信号给该进程。

- **SIGVTALRM**

应用程序调用 setitimer()函数设置一个虚拟定时器，当定时器定时时间到时，内核将会发送该信号给进程。

- **SIGWINCH**

在窗口环境中，当终端窗口尺寸发生变化时（譬如用户手动调整了大小，应用程序调用 ioctl()设置了大小等），系统会向前台进程组中的每一个进程发送该信号。

- **SIGPOLL/SIGIO**

这两个信号同义。这两个信号将会在高级 IO 章节内容中使用到，用于提示一个异步 IO 事件的发生，譬如应用程序打开的文件描述符发生了 I/O 事件时，内核会向应用程序发送 SIGIO 信号。

- **SIGSYS**

如果进程发起的系统调用有误，那么内核将发送该信号给对应的进程。

以上就是关于这些信号的简单介绍内容，以上所介绍的这些信号并不包括 Linux 下所有的信号，仅给大家介绍了一下常见信号，表 8.3.1 将对这些信号进行总结。

表 8.3.1 Linux 信号总结

信号名称	编号	描述	系统默认操作
SIGINT	2	终端中断符	term
SIGQUIT	3	终端退出符	term+core
SIGILL	4	非法硬件指令	term+core
SIGABRT	6	异常终止（abort）	term+core
SIGBUS	7	内存访问错误	term+core
SIGFPE	8	算术异常	term+core
SIGKILL	9	终极终止信号	term
SIGUSR1	10	用户自定义信号 1	term
SIGSEGV	11	无效的内存引用	term+core
SIGUSR2	12	用户自定义信号 2	term
SIGPIPE	13	管道关闭	term
SIGALRM	14	定时器超时（alarm）	term
SIGTERM	15	终止进程	term
SIGCHLD/SIGCLD	17	子进程终止或停止	ignore
SIGCONT	18	使停止状态的进程继续运行	cont
SIGSTOP	19	停止进程	stop
SIGTSTP	20	终端停止符	stop
SIGXCPU	24	超过 CPU 限制	term+core
SIGVTALRM	26	虚拟定时器超时	term
SIGWINCH	28	终端窗口尺寸发生变化	ignore
SIGPOLL/SIGIO	29	异步 I/O	term/ignore
SIGSYS	31	无效系统调用	term+core

Tips：上表中，term 表示终止进程；core 表示生成核心转储文件，核心转储文件可用于调试，这个便不再给介绍了；ignore 表示忽略信号；cont 表示继续运行进程；stop 表示停止进程（注意停止不等于终止，而是暂停）。

8.4 进程对信号的处理

当进程接收到内核或用户发送过来的信号之后，根据具体信号可以采取不同的处理方式：忽略信号、捕获信号或者执行系统默认操作。Linux 系统提供了系统调用 signal()和 sigaction()两个函数用于设置信号的处理方式，本小节将向大家介绍这两个系统调用的使用方法。

8.4.1 signal()函数

本节描述系统调用 signal()，signal()函数是 Linux 系统下设置信号处理方式最简单的接口，可将信号的处理方式设置为捕获信号、忽略信号以及系统默认操作，此函数原型如下所示：

```
#include <signal.h>

typedef void (*sig_t)(int);
sig_t signal(int signum, sig_t handler);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

signum：此参数指定需要进行设置的信号，可使用信号名（宏）或信号的数字编号，建议使用信号名。

handler：sig_t 类型的函数指针，指向信号对应的信号处理函数，当进程接收到信号后会自动执行该处理函数；参数 handler 既可以设置为用户自定义的函数，也就是捕获信号时需要执行的处理函数，也可以设置为 SIG_IGN 或 SIG_DFL，SIG_IGN 表示此进程需要忽略该信号，SIG_DFL 则表示设置为系统默认操作。sig_t 函数指针的 int 类型参数指的是，当前触发该函数的信号，可将多个信号绑定到同一个信号处理函数上，此时就可通过此参数来判断当前触发的是哪个信号。

Tips：SIG_IGN、SIG_DFL 分别取值如下：

```
/* Fake signal functions.  */
#define SIG_ERR  ((sig_t) -1)        /* Error return.  */
#define SIG_DFL  ((sig_t) 0)         /* Default action.  */
#define SIG_IGN  ((sig_t) 1)         /* Ignore signal.  */
```

返回值：此函数的返回值也是一个 sig_t 类型的函数指针，成功情况下的返回值则是指向在此之前的信号处理函数；如果出错则返回 SIG_ERR，并会设置 errno。

由此可知，signal()函数可以根据第二个参数 handler 的不同设置情况，可对信号进行不同的处理。

测试

signal()函数的用法其实非常简单，为信号设置相应的处理方式，接下来编写一个简单地示例代码对 signal()函数进行测试。

示例代码 8.4.1 signal()函数使用示例

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

static void sig_handler(int sig)
{
    printf("Received signal: %d\n", sig);
}

int main(int argc, char *argv[])
{
    sig_t ret = NULL;

    ret = signal(SIGINT, (sig_t)sig_handler);
    if (SIG_ERR == ret) {
```

```
        perror("signal error");
        exit(-1);
    }

    /* 死循环 */
    for ( ; ; ) { }

    exit(0);
}
```

在上述示例代码中，我们通过 signal()函数将 SIGINT（2）信号绑定到了一个用户自定的处理函数上 sig_handler(int sig)，当进程收到 SIGINT 信号后会执行该函数然后运行 printf 打印语句。当运行程序之后，程序会卡在 for 死循环处，此时在终端按下中断符 CTRL + C，系统便会给前台进程组中的每一个进程发送 SIGINT 信号，我们测试程序便会收到该信号。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
```

图 8.4.1 测试结果

当运行程序之后，程序会占用终端称为一个前台进程，此时按下中断符便会打印出信息（^C 表示按下了中断符）。平时大家使用 CTRL + C 可以终止一个进程，而这里却不能通过这种方式来终止这个测试程序，原因在于测试程序中捕获了该信号，而对应的处理方式仅仅只是打印一条语句、而并不终止进程。

那此时该怎么关闭这个测试程序呢？前面给大家介绍了“一击必杀”信号 SIGKILL（编号为 9），可向该进程发送 SIGKILL 暴力终止该进程，当然一般不推荐大家这样使用，如果实在没办法才采取这种措施。新打开一个终端，使用 ps 命令找到该进程的 pid 号，再使用 kill 命令，如下所示：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ ps -aux | grep testApp | grep -v grep
dt        13946 99.7  0.0   4352   644 pts/19   R+   16:17  12:29 ./testApp
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ kill -9 13946
dt@dt-virtual-machine:~$
```

进程pid

图 8.4.2 一击必杀

此时测试程序就会强制终止：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
已杀死
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.4.3 测试程序被终止

Tips：普通用户只能杀死该用户自己的进程，无权限杀死其它用户的进程。

我们再执行一次测试程序，这里将测试程序放在后台运行，然后再按下中断符：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[1] 14158
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ^C
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
    PID TTY          TIME CMD
   3304 pts/19   00:00:13 bash
  14158 pts/19   00:07:23 testApp
  14203 pts/19   00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.4.4 测试结果

按下中断符发现进程并没有收到 SIGINT 信号，原因很简单，因为进程并不是前台进程、而是一个后台进程，按下中断符时系统并不会给后台进程发送 SIGINT 信号。可以使用 kill 命令手动发送信号给我们的进程：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
    PID TTY          TIME CMD
   3304 pts/19   00:00:13 bash
  14158 pts/19   00:11:10 testApp
  14226 pts/19   00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 14158
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.4.5 测试结果

两种不同状态下信号的处理方式

通过上面的介绍，以及我们的测试实验，不知大家是否出现了一个疑问？如果程序中没有调用 signal()函数为信号设置相应的处理方式，亦或者程序刚启动起来并未运行到 signal()处，那么这时进程接收到一个信号后是如何处理的呢？带着这个问题来聊一聊。

- **程序启动**

当一个应用程序刚启动的时候（或者程序中没有调用 signal()函数），通常情况下，进程对所有信号的处理方式都设置为系统默认操作。所以如果在我们的程序当中，没有调用 signal()为信号设置处理方式，则默认的处理方式便是系统默认操作。

所以为什么大家平时都可以使用 CTRL + C 中断符来终止一个进程，因为大部分情况下，应用程序中并不会为 SIGINT 信号设置处理方式，所以该信号的处理方式便是系统默认操作，当接收到信号之后便执行系统默认操作，而 SIGINT 信号的系统默认操作便是终止进程。

- **进程创建**

当一个进程调用 fork()创建子进程时，其子进程将会继承父进程的信号处理方式，因为子进程在开始时复制了父进程的内存映像，所以信号捕获函数的地址在子进程中是有意义的。

8.4.2 sigaction()函数

除了 signal()之外，sigaction()系统调用是设置信号处理方式的另一选择，事实上，推荐大家使用 sigaction()函数。虽然 signal()函数简单好用，而 sigaction()更为复杂，但作为回报，sigaction()也更具灵活性以及移植性。

sigaction()允许单独获取信号的处理函数而不是设置，并且还可以设置各种属性对调用信号处理函数时的行为施以更加精准的控制，其函数原型如下所示：

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

signum：需要设置的信号，除了 SIGKILL 信号和 SIGSTOP 信号之外的任何信号。

act：act 参数是一个 struct sigaction 类型指针，指向一个 struct sigaction 数据结构，该数据结构描述了信号的处理方式，稍后介绍该数据结构；如果参数 act 不为 NULL，则表示需要为信号设置新的处理方式；如果参数 act 为 NULL，则表示无需改变信号当前的处理方式。

oldact：oldact 参数也是一个 struct sigaction 类型指针，指向一个 struct sigaction 数据结构。如果参数 oldact 不为 NULL，则会将信号之前的处理方式等信息通过参数 oldact 返回出来；如果无意获取此类信息，那么可将该参数设置为 NULL。

返回值：成功返回 0；失败将返回-1，并设置 errno。

struct sigaction 结构体

示例代码 8.4.2 struct sigaction 结构体

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

结构体成员介绍：

- sa_handler：指定信号处理函数，与 signal()函数的 handler 参数相同。
- sa_sigaction：也用于指定信号处理函数，这是一个替代的信号处理函数，他提供了更多的参数，可以通过该函数获取到更多信息，这些信号通过 siginfo_t 参数获取，稍后介绍该数据结构；sa_handler 和 sa_sigaction 是互斥的，不能同时设置，对于标准信号来说，使用 sa_handler 就可以了，可通过 SA_SIGINFO 标志进行选择。
- sa_mask：参数 sa_mask 定义了一组信号，当进程在执行由 sa_handler 所定义的信号处理函数之前，会先将这组信号添加到进程的信号掩码字段中，当进程执行完处理函数之后再恢复信号掩码，将这组信号从信号掩码字段中删除。当进程在执行信号处理函数期间，可能又收到了同样的信号或其它信号，从而打断当前信号处理函数的执行，这就好点像中断嵌套；通常我们在执行信号处理函数期间不希望被另一个信号所打断，那么怎么做呢？那么就是通过信号掩码来实现，如果进程接收到了信号掩码中的这些信号，那么这个信号将会被阻塞暂时不能得到处理，直到这些信号从进程的信号掩码中移除。在信号处理函数调用时，进程会自动将当前处理的信号添加到信号掩码字段中，这样保证了在处理一个给定的信号时，如果此信号再次发生，那么它将会被阻塞。如果用户还需要在阻塞其它的信号，则可以通过设置参

数 sa_mask 来完成（此参数是 sigset_t 类型变量，关于该类型的介绍信息请看 8.6.1 小节内容，关于信号掩码还会在 8.7.1 小节中进一步介绍），信号掩码可以避免一些信号之间的竞争状态（也称为竞态）。

- sa_restorer：该成员已过时，不要再使用了。
- sa_flags：参数 sa_flags 指定了一组标志，这些标志用于控制信号的处理过程，可设置为如下这些标志（多个标志使用位或"|"组合）：

SA_NOCLDSTOP

如果 signum 为 SIGCHLD，则子进程停止时（即当它们接收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 中的一种时）或恢复（即它们接收到 SIGCONT）时不会收到 SIGCHLD 信号。

SA_NOCLDWAIT

如果 signum 是 SIGCHLD，则在子进程终止时不要将其转变为僵尸进程。

SA_NODEFER

不要阻塞从某个信号自身的信号处理函数中接收此信号。也就是说当进程此时正在执行某个信号的处理函数，默认情况下，进程会自动将该信号添加到进程的信号掩码字段中，从而在执行信号处理函数期间阻塞该信号，默认情况下，我们期望进程在处理一个信号时阻塞同种信号，否则引起一些竞态条件；如果设置了 SA_NODEFER 标志，则表示不对它进行阻塞。

SA_RESETHAND

执行完信号处理函数之后，将信号的处理方式设置为系统默认操作。

SA_RESTART

被信号中断的系统调用，在信号处理完成之后将自动重新发起。

SA_SIGINFO

如果设置了该标志，则表示使用 sa_sigaction 作为信号处理函数、而不是 sa_handler，关于 sa_sigaction 信号处理函数的参数信息。

以上就是关于 struct sigaction 结构体相关的内容介绍了，接下编写程序进行实战测试。

siginfo_t 结构体

示例代码 8.4.3 siginfo_t 结构体

```
siginfo_t {
    int         si_signo;     /* Signal number */
    int         si_errno;     /* An errno value */
    int         si_code;      /* Signal code */
    int         si_trapno;    /* Trap number that caused hardware-generated signal(unused on most architectures) */
    pid_t       si_pid;       /* Sending process ID */
    uid_t       si_uid;       /* Real user ID of sending process */
    int         si_status;    /* Exit value or signal */
    clock_t     si_utime;     /* User time consumed */
    clock_t     si_stime;     /* System time consumed */
    sigval_t    si_value;     /* Signal value */
    int         si_int;       /* POSIX.1b signal */
    void        *si_ptr;      /* POSIX.1b signal */
    int         si_overrun;   /* Timer overrun count; POSIX.1b timers */
    int         si_timerid;   /* Timer ID; POSIX.1b timers */
    void        *si_addr;     /* Memory location which caused fault */
    long        si_band;      /* Band event (was int in glibc 2.3.2 and earlier) */
```

```
    int         si_fd;      /* File descriptor */
    short       si_addr_lsb; /* Least significant bit of address(since Linux 2.6.32) */
    void        *si_call_addr; /* Address of system call instruction(since Linux 3.5) */
    int         si_syscall; /* Number of attempted system call(since Linux 3.5) */
    unsigned int si_arch;   /* Architecture of attempted system call(since Linux 3.5) */
}
```

这个结构体就不给大家介绍了，使用 man 手册查看 sigaction()函数帮助信息时，在下面会有介绍。

测试

这里使用 sigaction()函数实现与示例代码 8.4.1 相同的功能。

示例代码 8.4.4 sigaction()函数使用示例

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

static void sig_handler(int sig)
{
    printf("Received signal: %d\n", sig);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int ret;

    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    ret = sigaction(SIGINT, &sig, NULL);
    if (-1 == ret) {
        perror("sigaction error");
        exit(-1);
    }

    /* 死循环 */
    for ( ; ; ) { }

    exit(0);
}
```

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
^CReceived signal: 2
```

图 8.4.6 测试结果

关于信号处理函数说明

一般而言，将信号处理函数设计越简单越好，这就好比中断处理函数，越快越好，不要在处理函数中做大量消耗 CPU 时间的事情，这一个重要的原因在于，设计的越简单这将降低引发信号竞争条件的风险。

8.5 向进程发送信号

与 kill 命令相类似，Linux 系统提供了 kill()系统调用，一个进程可通过 kill()向另一个进程发送信号；除了 kill()系统调用之外，Linux 系统还提供了系统调用 killpg()以及库函数 raise()，也可用于实现发送信号的功能，本小节将向大家进行介绍。

8.5.1 kill()函数

kill()系统调用可将信号发送给指定的进程或进程组中的每一个进程，其函数原型如下所示：

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

int kill(pid_t pid, int sig);
```

使用该函数需要包含头文件<sys/types.h>和<signal.h>。

函数参数和返回值含义如下：

pid：参数 pid 为正数的情况下，用于指定接收此信号的进程 pid；除此之外，参数 pid 也可设置为 0 或-1 以及小于-1 等不同值，稍后给说明。

sig：参数 sig 指定需要发送的信号，也可设置为 0，如果参数 sig 设置为 0 则表示不发送信号，但任执行错误检查，这通常可用于检查参数 pid 指定的进程是否存在。

返回值：成功返回 0；失败将返回-1，并设置 errno。

参数 pid 不同取值含义：

- 如果 pid 为正，则信号 sig 将发送到 pid 指定的进程。
- 如果 pid 等于 0，则将 sig 发送到当前进程的进程组中的每个进程。
- 如果 pid 等于-1，则将 sig 发送到当前进程有权发送信号的每个进程，但进程 1（init）除外。
- 如果 pid 小于-1，则将 sig 发送到 ID 为-pid 的进程组中的每个进程。

进程中将信号发送给另一个进程是需要权限的，并不是可以随便给任何一个进程发送信号，超级用户 root 进程可以将信号发送给任何进程，但对于非超级用户（普通用户）进程来说，其基本规则是发送者进程的实际用户 ID 或有效用户 ID 必须等于接收者进程的实际用户 ID 或有效用户 ID。

从上面介绍可知，当 sig 为 0 时，仍可进行正常执行的错误检查，但不会发送信号，这通常可用于确定一个特定的进程是否存在，如果向一个不存在的进程发送信号，kill()将会返回-1，errno 将被设置为 ESRCH，表示进程不存在。

测试

(1)使用 kill()函数向一个指定的进程发送信号。

示例代码 8.5.1 kill()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int pid;

    /* 判断传参个数 */
    if (2 > argc)
        exit(-1);

    /* 将传入的字符串转为整形数字 */
    pid = atoi(argv[1]);
    printf("pid: %d\n", pid);

    /* 向 pid 指定的进程发送信号 */
    if (-1 == kill(pid, SIGINT)) {
        perror("kill error");
        exit(-1);
    }

    exit(0);
}
```

以上代码通过 kill()函数向指定进程发送 SIGINT 信号，可通过外部传参将接收信号的进程 pid 传入到程序中，再执行该测试代码之前，需要运行先一个用于接收此信号的进程，接收信号的进程直接使用示例代码 8.4.4 程序。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp1  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp1 &
[1] 21825
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
   PID TTY          TIME CMD
  3304 pts/19   00:00:13 bash
 21825 pts/19   00:00:03 testApp1
 21826 pts/19   00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 21825
pid: 21825
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 21825
pid: 21825
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 21825
pid: 21825
Received signal: 2
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.5.1 测试结果

testApp1 是示例代码 8.4.4 对应的程序，testApp 则是示例代码 8.5.1 对应的程序，首先执行"./testApp1 &"将接收信号的程序置于后台运行（其进程 pid 为 21825），接着执行"./testApp 21825"向接收信号的进程发送 SIGINT 信号。

8.5.2 raise()

有时进程需要向自身发送信号，raise()函数可用于实现这一要求，raise()函数原型如下所示（此函数为 C 库函数）：

```
#include <signal.h>

int raise(int sig);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

sig：需要发送的信号。

返回值：成功返回 0；失败将返回非零值。

raise()其实等价于：

```
kill(getpid(), sig);
```

Tips：getpid()函数用于获取进程自身的 pid。

测试

示例代码 8.5.2 raise()函数使用示例

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

static void sig_handler(int sig)
{
    printf("Received signal: %d\n", sig);
}
```

```
int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int ret;

    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    ret = sigaction(SIGINT, &sig, NULL);
    if (-1 == ret) {
        perror("sigaction error");
        exit(-1);
    }

    for ( ; ; ) {

        /* 向自身发送 SIGINT 信号 */
        if (0 != raise(SIGINT)) {
            printf("raise error\n");
            exit(-1);
        }

        sleep(3);    // 每隔 3 秒发送一次
    }

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
Received signal: 2
```

图 8.5.2 测试结果

8.6 alarm()和 pause()函数

本小节向大家介绍两个系统调用 alarm()和 pause()。

8.6.1 alarm()函数

使用 alarm()函数可以设置一个定时器（闹钟），当定时器定时时间到时，内核会向进程发送 SIGALRM 信号，其函数原型如下所示：

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

函数参数和返回值：

seconds：设置定时时间，以秒为单位；如果参数 seconds 等于 0，则表示取消之前设置的 alarm 闹钟。

返回值：如果在调用 alarm()时，之前已经为该进程设置了 alarm 闹钟还没有超时，则该闹钟的剩余值作为本次 alarm()函数调用的返回值，之前设置的闹钟则被新的替代；否则返回 0。

参数 seconds 的值是产生 SIGALRM 信号需要经过的时钟秒数，当这一刻到达时，由内核产生该信号，每个进程只能设置一个 alarm 闹钟；虽然 SIGALRM 信号的系统默认操作是终止进程，但是如果程序当中设置了 alarm 闹钟，但大多数使用闹钟的进程都会捕获此信号。

需要注意的是 alarm 闹钟并不能循环触发，只能触发一次，若想要实现循环触发，可以在 SIGALRM 信号处理函数中再次调用 alarm()函数设置定时器。

测试

使用 alarm()来设计一个闹钟。

示例代码 8.6.1 alarm()函数使用示例

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

static void sig_handler(int sig)
{
    puts("Alarm timeout");
    exit(0);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int second;

    /* 检验传参个数 */
    if (2 > argc)
        exit(-1);

    /* 为 SIGALRM 信号绑定处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGALRM, &sig, NULL)) {
```

```
        perror("sigaction error");
        exit(-1);
    }

    /* 启动 alarm 定时器 */
    second = atoi(argv[1]);
    printf("定时时长: %d 秒\n", second);
    alarm(second);

    /* 循环 */
    for ( ; ; )
        sleep(1);

    exit(0);
}
```

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 5
定时时长: 5秒
Alarm timeout
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 2
定时时长: 2秒
Alarm timeout
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.6.1 测试结果

8.6.2 pause()函数

pause()系统调用可以使得进程暂停运行、进入休眠状态，直到进程捕获到一个信号为止，只有执行了信号处理函数并从其返回时，pause()才返回，在这种情况下，pause()返回-1，并且将 errno 设置为 EINTR。其函数原型如下所示：

```
#include <unistd.h>

int pause(void);
```

测试

通过 alarm()和 pause()函数模拟 sleep 功能。

示例代码 8.6.2 alarm()和 pause()模拟 sleep

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

static void sig_handler(int sig)
{
```

```
    puts("Alarm timeout");
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};
    int second;

    /* 检验传参个数 */
    if (2 > argc)
        exit(-1);

    /* 为 SIGALRM 信号绑定处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGALRM, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    /* 启动 alarm 定时器 */
    second = atoi(argv[1]);
    printf("定时时长: %d 秒\n", second);
    alarm(second);

    /* 进入休眠状态 */
    pause();
    puts("休眠结束");

    exit(0);
}
```

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 5
定时时长: 5秒
Alarm timeout
休眠结束
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.6.2 测试结果

8.7 信号集

通常我们需要有一个能表示多个信号（一组信号）的数据类型---信号集（signal set），很多系统调用都使用到了信号集这种数据类型来作为参数传递，譬如 sigaction()函数、sigprocmask()函数、sigpending()函数等。本小节向大家介绍信号集这个数据类型。

信号集其实就是 sigset_t 类型数据结构，来看看：

```
# define _SIGSET_NWORDS      (1024 / (8 * sizeof (unsigned long int)))
typedef struct
{
    unsigned long int __val[_SIGSET_NWORDS];
} sigset_t;
```

使用这个结构体可以表示一组信号，将多个信号添加到该数据结构中，当然 Linux 系统了用于操作 sigset_t 信号集的 API，譬如 sigemptyset()、sigfillset()、sigaddset()、sigdelset()、sigismember()，接下来向大家介绍。

8.7.1 初始化信号集

sigemptyset()和 sigfillset()用于初始化信号集。sigemptyset()初始化信号集，使其不包含任何信号；而 sigfillset()函数初始化信号集，使其包含所有信号（包括所有实时信号），函数原型如下：

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

使用这些函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

set：指向需要进行初始化的信号集变量。

返回值：成功返回 0；失败将返回-1，并设置 errno。

使用示例

初始化为空信号集：

```
sigset_t sig_set;

sigemptyset(&sig_set);
```

初始化信号集，使其包含所有信号：

```
sigset_t sig_set;

sigfillset(&sig_set);
```

8.7.2 向信号集中添加/删除信号

分别使用 sigaddset()和 sigdelset()函数向信号集中添加或移除一个信号，函数原型如下：

```
#include <signal.h>

int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

函数参数和返回值含义如下：

set：指向信号集。

signum：需要添加/删除的信号。

返回值：成功返回 0；失败将返回-1，并设置 errno。

使用示例

向信号集中添加信号：

```
sigset_t sig_set;

sigemptyset(&sig_set);
sigaddset(&sig_set, SIGINT);
```

从信号集中移除信号：

```
sigset_t sig_set;

sigfillset(&sig_set);
sigdelset(&sig_set, SIGINT);
```

8.7.3 测试信号是否在信号集中

使用 sigismember()函数可以测试某一个信号是否在指定的信号集中，函数原型如下所示：

```
#include <signal.h>

int sigismember(const sigset_t *set, int signum);
```

函数参数和返回值含义如下：

set：指定信号集。

signum：需要进行测试的信号。

返回值：如果信号 signum 在信号集 set 中，则返回 1；如果不在信号集 set 中，则返回 0；失败则返回-1，并设置 errno。

使用示例

判断 SIGINT 信号是否在 sig_set 信号集中：

```
sigset_t sig_set;
......
if (1 == sigismember(&sig_set, SIGINT))
    puts("信号集中包含 SIGINT 信号");
else if (!sigismember(&sig_set, SIGINT))
    puts("信号集中不包含 SIGINT 信号");
```

8.8 获取信号的描述信息

在 Linux 下，每个信号都有一串与之相对应的字符串描述信息，用于对该信号进行相应的描述。这些字符串位于 sys_siglist 数组中，sys_siglist 数组是一个 char *类型的数组，数组中的每一个元素存放的是一个字符串指针，指向一个信号描述信息。譬如，可以使用 sys_siglist[SIGINT]来获取对 SIGINT 信号的描述。我们编写一个简单地程序进行测试：

Tips：使用 sys_siglist 数组需要包含<signal.h>头文件。

示例代码 8.8.1 从 sys_siglist 数组获取信号描述信息

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

int main(void)
{
    printf("SIGINT 描述信息: %s\n", sys_siglist[SIGINT]);
    printf("SIGQUIT 描述信息: %s\n", sys_siglist[SIGQUIT]);
    printf("SIGBUS 描述信息: %s\n", sys_siglist[SIGBUS]);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
SIGINT描述信息： Interrupt
SIGQUIT描述信息： Quit
SIGBUS描述信息： Bus error
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.8.1 测试结果

从图中打印信息可知，这个描述信息其实非常简洁，没什么太多的信息。

8.8.1 strsignal()函数

除了直接使用 sys_siglist 数组获取描述信息之外，还可以使用 strsignal()函数。较之于直接引用 sys_siglist 数组，更推荐使用 strsignal()函数，其函数原型如下所示：

```
#include <string.h>

char *strsignal(int sig);
```

使用 strsignal()函数需要包含头文件<string.h>，这是一个库函数。

调用 strsignal()函数将会获取到参数 sig 指定的信号对应的描述信息，返回该描述信息字符串的指针；函数会对参数 sig 进行检查，若传入的 sig 无效，则会返回"Unknown signal"信息。

使用示例

示例代码 8.8.2 strsignal()函数使用示例

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

int main(void)
{
    printf("SIGINT 描述信息: %s\n", strsignal(SIGINT));
```

```
    printf("SIGQUIT 描述信息: %s\n", strsignal(SIGQUIT));
    printf("SIGBUS 描述信息: %s\n", strsignal(SIGBUS));
    printf("编号为 1000 的描述信息: %s\n", strsignal(1000));
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
SIGINT描述信息： Interrupt
SIGQUIT描述信息： Quit
SIGBUS描述信息： Bus error
编号为1000的描述信息： Unknown signal 1000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.8.2 测试结果

8.8.2 psignal()函数

psignal()可以在标准错误（stderr）上输出信号描述信息，其函数原型如下所示：

```
#include <signal.h>

void psignal(int sig, const char *s);
```

调用 psignal()函数会将参数 sig 指定的信号对应的描述信息输出到标准错误，并且还允许调用者添加一些输出信息，由参数 s 指定；所以整个输出信息由字符串 s、冒号、空格、描述信号编号 sig 的字符串和尾随的换行符组成。

使用示例

示例代码 8.8.3 psignal()函数使用示例

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

int main(void)
{
    psignal(SIGINT, "SIGINT 信号描述信息");
    psignal(SIGQUIT, "SIGQUIT 信号描述信息");
    psignal(SIGBUS, "SIGBUS 信号描述信息");
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
SIGINT信号描述信息：Interrupt
SIGQUIT信号描述信息：Quit
SIGBUS信号描述信息：Bus error
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.8.3 运行结果

8.9 信号掩码(阻塞信号传递)

内核为每一个进程维护了一个信号掩码（其实就是一个信号集），即一组信号。当进程接收到一个属于信号掩码中定义的信号时，该信号将会被阻塞、无法传递给进程进行处理，那么内核会将其阻塞，直到该信号从信号掩码中移除，内核才会把该信号传递给进程从而得到处理。

向信号掩码中添加一个信号，通常有如下几种方式：

- 当应用程序调用 signal()或 sigaction()函数为某一个信号设置处理方式时，进程会自动将该信号添加到信号掩码中，这样保证了在处理一个给定的信号时，如果此信号再次发生，那么它将会被阻塞；当然对于 sigaction()而言，是否会如此，需要根据 sigaction()函数是否设置了 SA_NODEFER 标志而定；当信号处理函数结束返回后，会自动将该信号从信号掩码中移除。
- 使用 sigaction()函数为信号设置处理方式时，可以额外指定一组信号，当调用信号处理函数时将该组信号自动添加到信号掩码中，当信号处理函数结束返回后，再将这组信号从信号掩码中移除；通过 sa_mask 参数进行设置，参考 8.4.2 小节内容。
- 除了以上两种方式之外，还可以使用 sigprocmask()系统调用，随时可以显式地向信号掩码中添加/移除信号。

sigprocmask()函数原型如下所示：

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

how：参数 how 指定了调用函数时的一些行为。

set：将参数 set 指向的信号集内的所有信号添加到信号掩码中或者从信号掩码中移除；如果参数 set 为 NULL，则表示无需对当前信号掩码作出改动。

oldset：如果参数 oldset 不为 NULL，在向信号掩码中添加新的信号之前，获取到进程当前的信号掩码，存放在 oldset 所指定的信号集中；如果为 NULL 则表示不获取当前的信号掩码。

返回值：成功返回 0；失败将返回-1，并设置 errno。

参数 how 可以设置为以下宏：

- SIG_BLOCK：将参数 set 所指向的信号集内的所有信号添加到进程的信号掩码中。换言之，将信号掩码设置为当前值与 set 的并集。
- SIG_UNBLOCK：将参数 set 指向的信号集内的所有信号从进程信号掩码中移除。
- SIG_SETMASK：进程信号掩码直接设置为参数 set 指向的信号集。

使用示例

将信号 SIGINT 添加到进程的信号掩码中：

```
    int ret;

    /* 定义信号集 */
    sigset_t sig_set;

    /* 将信号集初始化为空 */
    sigemptyset(&sig_set);

    /* 向信号集中添加 SIGINT 信号 */
    sigaddset(&sig_set, SIGINT);

    /* 向进程的信号掩码中添加信号 */
    ret = sigprocmask(SIG_BLOCK, &sig_set, NULL);
    if (-1 == ret) {
        perror("sigprocmask error");
        exit(-1);
    }
```

从信号掩码中移除 SIGINT 信号：

```
    int ret;

    /* 定义信号集 */
    sigset_t sig_set;

    /* 将信号集初始化为空 */
    sigemptyset(&sig_set);

    /* 向信号集中添加 SIGINT 信号 */
    sigaddset(&sig_set, SIGINT);

    /* 从信号掩码中移除信号 */
    ret = sigprocmask(SIG_UNBLOCK, &sig_set, NULL);
    if (-1 == ret) {
        perror("sigprocmask error");
        exit(-1);
    }
```

下面我们编写一个简单地测试代码，验证信号掩码的作用，测试代码如下所示：

示例代码 8.9.1 测试信号掩码的作用

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

static void sig_handler(int sig)
```

```
{
    printf("执行信号处理函数...\n");
}

int main(void)
{
    struct sigaction sig = {0};
    sigset_t sig_set;

    /* 注册信号处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGINT, &sig, NULL))
        exit(-1);

    /* 信号集初始化 */
    sigemptyset(&sig_set);
    sigaddset(&sig_set, SIGINT);

    /* 向信号掩码中添加信号 */
    if (-1 == sigprocmask(SIG_BLOCK, &sig_set, NULL))
        exit(-1);

    /* 向自己发送信号 */
    raise(SIGINT);

    /* 休眠 2 秒 */
    sleep(2);
    printf("休眠结束\n");

    /* 从信号掩码中移除添加的信号 */
    if (-1 == sigprocmask(SIG_UNBLOCK, &sig_set, NULL))
        exit(-1);

    exit(0);
}
```

上述代码中，我们为 SIGINT 信号注册了一个处理函数 sig_handler，当进程接收到该信号之后就会执行它；然后调用 sigprocmask 函数将 SIGINT 信号添加到信号掩码中，然后再调用 raise(SIGINT)向自己发送一个 SIGINT 信号，如果信号掩码没有生效、也就意味着 SIGINT 信号不会被阻塞，那么调用 raise(SIGINT)之后应该就会立马执行 sig_handler 函数，从而打印出"执行信号处理函数..."字符串信息；如果设置的信号掩码生效了，则并不会立马执行信号处理函数，而是在 2 秒后才执行，因为程序中使用 sleep(2)休眠了 2 秒钟之后，才将 SIGINT 信号从信号掩码中移除，故而进程才会处理该信号，在移除之前接收到该信号会将其阻塞。

编译测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
休眠结束
执行信号处理函数...
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.9.1 测试结果

8.10 阻塞等待信号 sigsuspend()

上一小节已经说明，更改进程的信号掩码可以阻塞所选择的信号，或解除对它们的阻塞。使用这种技术可以保护不希望由信号中断的关键代码段。如果希望对一个信号解除阻塞后，然后调用 pause()以等待之前被阻塞的信号的传递，这将如何？譬如有如下代码段：

```
sigset_t new_set, old_set;

/* 信号集初始化 */
sigemptyset(&new_set);
sigaddset(&new_set, SIGINT);

/* 向信号掩码中添加信号 */
if (-1 == sigprocmask(SIG_BLOCK, &new_set, &old_set))
    exit(-1);

/* 受保护的关键代码段 */
......
/**********************/

/* 恢复信号掩码 */
if (-1 == sigprocmask(SIG_SETMASK, &old_set, NULL))
    exit(-1);

pause();/* 等待信号唤醒 */
```

执行受保护的关键代码时不希望被 SIGINT 信号打断，所以在执行关键代码之前将 SIGINT 信号添加到进程的信号掩码中，执行完毕之后再恢复之前的信号掩码。最后调用了 pause()阻塞等待被信号唤醒，如果此时发生了信号则会被唤醒、从 pause 返回继续执行；考虑到这样一种情况，如果信号的传递恰好发生在第二次调用 sigprocmask()之后、pause()之前，如果确实发生了这种情况，就会产生一个问题，信号传递过来就会导致执行信号的处理函数，而从处理函数返回后又回到主程序继续执行，从而进入到 pause()被阻塞，知道下一次信号发生时才会被唤醒，这有违代码的本意。

虽然信号传递发生在这个时间段的可能性并不大，但并不是完全没有可能，这必然是一个缺陷，要避免这个问题，需要将恢复信号掩码和 pause()挂起进程这两个动作封装成一个原子操作，这正是 sigsuspend()系统调用的目的所在，sigsuspend()函数原型如下所示：

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

使用该函数需要包含头文件#include <signal.h>。

函数参数和返回值含义如下：

mask：参数 mask 指向一个信号集。

返回值：sigsuspend()始终返回-1，并设置 errno 来指示错误（通常为 EINTR），表示被信号所中断，如果调用失败，将 errno 设置为 EFAULT。

sigsuspend()函数会将参数 mask 所指向的信号集来替换进程的信号掩码，也就是将进程的信号掩码设置为参数 mask 所指向的信号集，然后挂起进程，直到捕获到信号被唤醒（如果捕获的信号是 mask 信号集中的成员，将不会唤醒、继续挂起）、并从信号处理函数返回，一旦从信号处理函数返回，sigsuspend()会将进程的信号掩码恢复成调用前的值。

调用 sigsuspend()函数相当于以不可中断（原子操作）的方式执行以下操作：

```
sigprocmask(SIG_SETMASK, &mask, &old_mask);
pause();
sigprocmask(SIG_SETMASK, &old_mask, NULL);
```

使用示例

示例代码 8.10.1 sigsuspend()函数使用示例

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

static void sig_handler(int sig)
{
    printf("执行信号处理函数...\n");
}

int main(void)
{
    struct sigaction sig = {0};
    sigset_t new_mask, old_mask, wait_mask;

    /* 信号集初始化 */
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGINT);
    sigemptyset(&wait_mask);

    /* 注册信号处理函数 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGINT, &sig, NULL))
        exit(-1);

    /* 向信号掩码中添加信号 */
    if (-1 == sigprocmask(SIG_BLOCK, &new_mask, &old_mask))
```

```
        exit(-1);

    /* 执行保护代码段 */
    puts("执行保护代码段");
    /******************/

    /* 挂起、等待信号唤醒 */
    if (-1 != sigsuspend(&wait_mask))
        exit(-1);

    /* 恢复信号掩码 */
    if (-1 == sigprocmask(SIG_SETMASK, &old_mask, NULL))
        exit(-1);

    exit(0);
}
```

在上述代码中，我们希望执行受保护代码段时不被 SIGINT 中断信号打断，所以在执行保护代码段之前将 SIGINT 信号添加到进程的信号掩码中，执行完受保护的代码段之后，调用 sigsuspend()挂起进程，等待被信号唤醒，被唤醒之后再解除 SIGINT 信号的阻塞状态。

8.11 实时信号

如果进程当前正在执行信号处理函数，在处理信号期间接收到了新的信号，如果该信号是信号掩码中的成员，那么内核会将其阻塞，将该信号添加到进程的等待信号集（等待被处理，处于等待状态的信号）中，为了确定进程中处于等待状态的是哪些信号，可以使用 sigpending()函数获取。

8.11.1 sigpending()函数

其函数原型如下所示：

```
#include <signal.h>

int sigpending(sigset_t *set);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

set：处于等待状态的信号会存放在参数 set 所指向的信号集中。

返回值：成功返回 0；失败将返回-1，并设置 errno。

使用示例

判断 SIGINT 信号当前是否处于等待状态

```
    /* 定义信号集 */
    sigset_t sig_set;

    /* 将信号集初始化为空 */
    sigemptyset(&sig_set);
```

```
    /* 获取当前处于等待状态的信号 */
    sigpending(&sig_set);

    /* 判断 SIGINT 信号是否处于等待状态 */
    if (1 == sigismember(&sig_set, SIGINT))
        puts("SIGINT 信号处于等待状态");
    else if (!sigismember(&sig_set, SIGINT))
        puts("SIGINT 信号未处于等待状态");
```

8.11.2 发送实时信号

等待信号集只是一个掩码，仅表明一个信号是否发生，而不能表示其发生的次数。换言之，如果一个同一个信号在阻塞状态下产生了多次，那么会将该信号记录在等待信号集中，并在之后仅传递一次（仅当做发生了一次），这是标准信号的缺点之一。

实时信号较之于标准信号，其优势如下：

- 实时信号的信号范围有所扩大，可应用于应用程序自定义的目的，而标准信号仅提供了两个信号可用于应用程序自定义使用：SIGUSR1 和 SIGUSR2。
- 内核对于实时信号所采取的是队列化管理。如果将某一实时信号多次发送给另一个进程，那么将会多次传递此信号。相反，对于某一标准信号正在等待某一进程，而此时即使再次向该进程发送此信号，信号也只会传递一次。
- 当发送一个实时信号时，可为信号指定伴随数据（一整形数据或者指针值），供接收信号的进程在它的信号处理函数中获取。
- 不同实时信号的传递顺序得到保障。如果有多个不同的实时信号处于等待状态，那么将率先传递具有最小编号的信号。换言之，信号的编号越小，其优先级越高，如果是同一类型的多个信号在排队，那么信号（以及伴随数据）的传递顺序与信号发送来时的顺序保持一致。

Linux 内核定义了 31 个不同的实时信号，信号编号范围为 34~64，使用 SIGRTMIN 表示编号最小的实时信号，使用 SIGRTMAX 表示编号最大的实时信号，其它信号编号可使用这两个宏加上一个整数或减去一个整数。

应用程序当中使用实时信号，需要有以下的两点要求：

- 发送进程使用 sigqueue()系统调用向另一个进程发送实时信号以及伴随数据。
- 接收实时信号的进程要为该信号建立一个信号处理函数，使用 sigaction 函数为信号建立处理函数，并加入 SA_SIGINFO，这样信号处理函数才能够接收到实时信号以及伴随数据，也就是要使用 sa_sigaction 指针指向的处理函数，而不是 sa_handler，当然允许应用程序使用 sa_handler，但这样就不能获取到实时信号的伴随数据了。

使用 sigqueue()函数发送实时信号，其函数原型如下所示：

```
#include <signal.h>

int sigqueue(pid_t pid, int sig, const union sigval value);
```

使用该函数需要包含头文件<signal.h>。

函数参数和返回值含义如下：

pid：指定接收信号的进程对应的 pid，将信号发送给该进程。

sig：指定需要发送的信号。与 kill()函数一样，也可将参数 sig 设置为 0，用于检查参数 pid 所指定的进程是否存在。

value：参数 value 指定了信号的伴随数据，union sigval 数据类型。

返回值：成功将返回 0；失败将返回-1，并设置 errno。

union sigval 数据类型（共用体）如下所示：

```
typedef union sigval
{
    int sival_int;
    void *sival_ptr;
} sigval_t;
```

携带的伴随数据，既可以指定一个整形的数据，也可以指定一个指针。

使用示例

(1)发送进程使用 sigqueue()系统调用向另一个进程发送实时信号

示例代码 8.11.1 使用 sigqueue()函数发送信号

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

int main(int argc, char *argv[])
{
    union sigval sig_val;
    int pid;
    int sig;

    /* 判断传参个数 */
    if (3 > argc)
        exit(-1);

    /* 获取用户传递的参数 */
    pid = atoi(argv[1]);
    sig = atoi(argv[2]);
    printf("pid: %d\nsignal: %d\n", pid, sig);

    /* 发送信号 */
    sig_val.sival_int = 10; //伴随数据
    if (-1 == sigqueue(pid, sig, sig_val)) {
        perror("sigqueue error");
        exit(-1);
    }

    puts("信号发送成功!");
    exit(0);
}
```

(2)接收进程使用 sigaction()函数为信号绑定处理函数

示例代码 8.11.2 使用 sigaction()函数为实时信号绑定处理函数

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

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

static void sig_handler(int sig, siginfo_t *info, void *context)
{
	sigval_t sig_val = info->si_value;

	printf("接收到实时信号: %d\n", sig);
	printf("伴随数据为: %d\n", sig_val.sival_int);
}

int main(int argc, char *argv[])
{
	struct sigaction sig = {0};
	int num;

	/* 判断传参个数 */
	if (2 > argc)
		exit(-1);

	/* 获取用户传递的参数 */
	num = atoi(argv[1]);

	/* 为实时信号绑定处理函数 */
	sig.sa_sigaction = sig_handler;
	sig.sa_flags = SA_SIGINFO;
	if (-1 == sigaction(num, &sig, NULL)) {
		perror("sigaction error");
		exit(-1);
	}

	/* 死循环 */
	for ( ; ; )
		sleep(1);

	exit(0);
}
```

8.12 异常退出 abort()函数

在 3.3 小节中给大家介绍了应用程序中结束进程的几种方法，譬如使用 exit()、_exit()或_Exit()这些函数来终止进程，然后这些方法使用于正常退出应用程序，而对于异常退出程序，则一般使用 abort()库函数，使用 abort()终止进程运行，会生成核心转储文件，可用于判断程序调用 abort()时的程序状态。

abort()函数原型如下所示：

```
#include <stdlib.h>

void abort(void);
```

函数 abort()通常产生 SIGABRT 信号来终止调用该函数的进程，SIGABRT 信号的系统默认操作是终止进程运行、并生成核心转储文件；当调用 abort()函数之后，内核会向进程发送 SIGABRT 信号。

使用示例

示例代码 8.12.1 abort()终止进程

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

static void sig_handler(int sig)
{
    printf("接收到信号: %d\n", sig);
}

int main(int argc, char *argv[])
{
    struct sigaction sig = {0};

    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGABRT, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    sleep(2);
    abort();      // 调用 abort
    for ( ; ; )
        sleep(1);

    exit(0);
}
```

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
接收到信号：6
已放弃 (核心已转储)
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 8.12.1 测试结果

从打印信息可知，即使在我们的程序当中捕获了 SIGABRT 信号，但是程序依然会无情的终止，无论阻塞或忽略 SIGABRT 信号，abort()调用均不收到影响，总会成功终止进程。

第九章 进程

本章将讨论进程相关的知识内容，虽然在前面章节内容已经多次向大家提到了进程这个概念，但并未真正地向大家解释这个概念；在本章，我们将一起来学习 Linux 下进程相关的知识内容，虽然进程的基本概念比较简单，但是其所涉及到的细节内容比较多，所以本章篇幅也会相对比较长，所以，大家加油！

本章将会讨论如下主题内容。

- 程序与进程基本概念；
- 程序的开始与结束；
- 进程的环境变量与虚拟地址空间；
- 进程 ID；
- fork()创建子进程；
- 进程的消亡与诞生；
- 僵尸进程与孤儿进程；
- 父进程监视子进程；
- 进程关系与进程的六种状态；
- 守护进程；
- 进程间通信概述。

9.1 进程与程序

9.1.1 main()函数由谁调用？

C 语言程序总是从 main 函数开始执行，main()函数的原型是：

```
int main(void)
```

或

```
int main(int argc, char *argv[])
```

如果需要向应用程序传参，则选择第二种写法。不知大家是否想过“谁”调用了 main()函数？事实上，操作系统下的应用程序在运行 main()函数之前需要先执行一段引导代码，最终由这段引导代码去调用应用程序的 main()函数，我们在编写应用程序的时候，不用考虑引导代码的问题，在编译链接时，由链接器将引导代码链接到我们的应用程序当中，一起构成最终的可执行文件。

当执行应用程序时，在 Linux 下输入可执行文件的相对路径或绝对路径就可以运行该程序，譬如./app 或/home/dt/app，还可根据应用程序是否接受传参在执行命令时在后面添加传入的参数信息，譬如./app arg1 arg2 或/home/dt/app arg1 arg2。程序运行需要通过操作系统的加载器来实现，加载器是操作系统中的程序，当执行程序时，加载器负责将此应用程序加载内存中去执行。

所以由此可知，对于操作系统下的应用程序来说，链接器和加载器都是很重要的角色！

再来看看 argc 和 argv 传参是如何实现的呢？譬如./app arg1 arg2，这两个参数 arg1 和 arg2 是如何传递给应用程序的 main 函数的呢？当在终端执行程序时，命令行参数（command-line argument）由 shell 进程逐一进行解析，shell 进程会将这些参数传递给加载器，加载器加载应用程序时会将其传递给应用程序引导代码，当引导程序调用 main()函数时，在由它最终传递给 main()函数，如此一来，在我们的应用程序当中便可以获取到命令行参数了。

9.1.2 程序如何结束？

程序结束其实就是进程终止，进程终止的方式通常有多种，大体上分为正常终止和异常终止，正常终止包括：

- main()函数中通过 return 语句返回来终止进程；
- 应用程序中调用 exit()函数终止进程；
- 应用程序中调用_exit()或_Exit()终止进程；

以上这些是在前面的课程中给大家介绍的，异常终止包括：

- 应用程序中调用 abort()函数终止进程；
- 进程接收到一个信号，譬如 SIGKILL 信号。

注册进程终止处理函数 atexit()

atexit()库函数用于注册一个进程在**正常终止**时要调用的函数，其函数原型如下所示：

```
#include <stdlib.h>

int atexit(void (*function)(void));
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

function：函数指针，指向注册的函数，此函数无需传入参数、无返回值。

返回值：成功返回 0；失败返回非 0。

测试

编写一个测试程序，使用 atexit()函数注册一个进程在正常终止时需要调用的函数，测试代码如下。

示例代码 9.1.1 atexit()函数使用示例

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

static void bye(void)
{
    puts("Goodbye!");
}

int main(int argc, char *argv[])
{
    if (atexit(bye)) {
        fprintf(stderr, "cannot set exit function\n");
        exit(-1);
    }

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Goodbye!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.1.1 测试结果

需要说明的是，如果程序当中使用了_exit()或_Exit()终止进程而并非是 exit()函数，那么将不会执行注册的终止处理函数。

9.1.3 何为进程？

本小节正式向大家介绍进程这个概念，前面的内容中也已经多次提到了，其实这个概念本身非常简单，进程其实就是一个可执行程序的实例，这句话如何理解呢？可执行程序就是一个可执行文件，文件是一个静态的概念，存放磁盘中，如果可执行文件没有被运行，那它将不会产生什么作用，当它被运行之后，它将会对系统环境产生一定的影响，所以可执行程序的实例就是可执行文件被运行。

进程是一个动态过程，而非静态文件，它是程序的一次运行过程，当应用程序被加载到内存中运行之后它就称为了一个进程，当程序运行结束后也就意味着进程终止，这就是进程的一个生命周期。

9.1.4 进程号

Linux 系统下的每一个进程都有一个进程号（process ID，简称 PID），进程号是一个正数，用于唯一标识系统中的某一个进程。在 Ubuntu 系统下执行 ps 命令可以查到系统中进程相关的一些信息，包括每个进程的进程号，如下所示：

```
dt@dt-virtual-machine:~$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1 185416  4196 ?        Ss   3月09   0:27 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    3月09   0:00 [kthreadd]
root           4  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/0:0H]
root           6  0.0  0.0      0     0 ?        I<   3月09   0:00 [mm_percpu_wq]
root           7  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/0]
root           8  0.0  0.0      0     0 ?        I    3月09  10:22 [rcu_sched]
root           9  0.0  0.0      0     0 ?        I    3月09   0:00 [rcu_bh]
root          10  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/0]
root          11  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/0]
root          12  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/0]
root          13  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/1]
root          14  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/1]
root          15  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/1]
root          16  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/1]
root          18  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/1:0H]
root          19  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/2]
root          20  0.0  0.0      0     0 ?        S    3月09   0:03 [watchdog/2]
root          21  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/2]
root          22  0.0  0.0      0     0 ?        S    3月09   0:01 [ksoftirqd/2]
root          24  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/2:0H]
root          25  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/3]
root          26  0.0  0.0      0     0 ?        S    3月09   0:03 [watchdog/3]
root          27  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/3]
root          28  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/3]
```

图 9.1.2 ps 命令查看进程信息

上图中红框标识显示的便是每个进程所对应的进程号，进程号的作用就是用于唯一标识系统中某一个进程，在某些系统调用中，进程号可以作为传入参数、有时也可作为返回值。譬如系统调用 kill()允许调用者向某一个进程发送一个信号，如何表示这个进程呢？则是通过进程号进行标识。

在应用程序中，可通过系统调用 getpid()来获取本进程的进程号，其函数原型如下所示：

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

pid_t getpid(void);
```

使用该函数需要包含头文件<sys/types.h>和<unistd.h>。

函数返回值为 pid_t 类型变量，便是对应的进程号。

使用示例

使用 getpid()函数获取进程的进程号。

示例代码 9.1.2 getpid()使用示例

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

int main(void)
{
    pid_t pid = getpid();
    printf("本进程的 PID 为: %d\n", pid);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本进程的PID为： 46136
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.1.3 测试结果

除了 getpid()用于获取本进程的进程号之外，还可以使用 getppid()系统调用获取父进程的进程号，其函数原型如下所示：

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

pid_t getppid(void);
```

返回值对应的便是父进程的进程号。

使用示例

获取进程的进程号和父进程的进程号。

示例代码 9.1.3 getppid()使用示例

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

int main(void)
{
    pid_t pid = getpid();     //获取本进程 pid
    printf("本进程的 PID 为: %d\n", pid);

    pid = getppid();      //获取父进程 pid
    printf("父进程的 PID 为: %d\n", pid);

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
本进程的PID为： 46306
父进程的PID为： 3304
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.1.4 测试结果

9.2 进程的环境变量

每一个进程都有一组与其相关的环境变量，这些环境变量以字符串形式存储在一个字符串数组列表中，把这个数组称为环境列表。其中每个字符串都是以“名称=值（name=value）”形式定义，所以环境变量是“名称-值”的成对集合，譬如在 shell 终端下可以使用 env 命令查看到 shell 进程的所有环境变量，如下所示：

```
dt@dt-virtual-machine:~$ env
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dt
GPG_AGENT_INFO=/home/dt/.gnupg/S.gpg-agent:0:1
SHELL=/bin/bash
TERM=xterm-256color
VTE_VERSION=4205
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=58720266
OLDPWD=/proc
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1911
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
JRE_HOME=/opt/jdk1.8.0_271/jre
USER=dt
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:
*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31
1;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31
sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.
=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.m
=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmv
1;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;3
*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/opt/jdk1.8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/g
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=fcitx
QT_QPA_PLATFORMTHEME=appmenu-qt5
XDG_SESSION_TYPE=x11
PWD=/home/dt
JOB=gnome-session
XMODIFIERS=@im=fcitx
GNOME_KEYRING_PID=
LANG=zh_CN.UTF-8
GDM_LANG=zh_CN
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
TZ=:Asia/Harbin
```

图 9.2.1 env 命令查看环境变量

使用 export 命令还可以添加一个新的环境变量或删除一个环境变量：

```
export LINUX_APP=123456        # 添加 LINUX_APP 环境变量
```

```
dt@dt-virtual-machine:~$ export LINUX_APP=123456
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ env
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dt
GPG_AGENT_INFO=/home/dt/.gnupg/S.gpg-agent:0:1
SHELL=/bin/bash
TERM=xterm-256color
VTE_VERSION=4205
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
WINDOWID=58720266
OLDPWD=/proc
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1911
GNOME_KEYRING_CONTROL=
GTK_MODULES=gail:atk-bridge:unity-gtk-module
JRE_HOME=/opt/jdk1.8.0_271/jre
USER=dt
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40
*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;3
1;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;3
sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.
=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.s
=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.n
1;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01
*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
LINUX_APP=123456
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
```

图 9.2.2 export 添加环境变量

使用"export -n LINUX_APP"命令则可以删除 LINUX_APP 环境变量。

```
export -n LINUX_APP              # 删除 LINUX_APP 环境变量
```

9.2.1 应用程序中获取环境变量

在我们的应用程序当中也可以获取当前进程的环境变量，事实上，进程的环境变量是从其父进程中继承过来的，譬如在 shell 终端下执行一个应用程序，那么该进程的环境变量就是从其父进程（shell 进程）中继承过来的。新的进程在创建之前，会继承其父进程的环境变量副本。

环境变量存放在一个字符串数组中，在应用程序中，通过 environ 变量指向它，environ 是一个全局变量，在我们的应用程序中只需申明它即可使用，如下所示：

```
extern char **environ;          // 申明外部全局变量 environ
```

测试

编写应用程序，获取进程的所有环境变量。

示例代码 9.2.1 获取进程环境变量

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

extern char **environ;

int main(int argc, char *argv[])
{
    int i;

    /* 打印进程的环境变量 */
    for (i = 0; NULL != environ[i]; i++)
```

```
        puts(environ[i]);

    exit(0);
}
```

通过字符串数组元素是否等于 NULL 来判断是否已经到了数组的末尾。

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
XDG_VTNR=7
XDG_SESSION_ID=c2
TERM_PROGRAM=vscode
CLUTTER_IM_MODULE=xim
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/dt
GIO_LAUNCHED_DESKTOP_FILE_PID=3179
SESSION=ubuntu
GPG_AGENT_INFO=/home/dt/.gnupg/S.gpg-agent:0:1
TERM=xterm-256color
XDG_MENU_PREFIX=gnome-
SHELL=/bin/bash
QT_LINUX_ACCESSIBILITY_ALWAYS_ON=1
TERM_PROGRAM_VERSION=1.52.1
OLDPWD=/home/dt/vscode_ws
ORIGINAL_XDG_CURRENT_DESKTOP=Unity
UPSTART_SESSION=unix:abstract=/com/ubuntu/upstart-session/1000/1911
GTK_MODULES=gail:atk-bridge:unity-gtk-module
JRE_HOME=/opt/jdk1.8.0_271/jre
USER=dt
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41
z=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=
z=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;
rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bm
35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;
*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.
;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36
p3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
QT_ACCESSIBILITY=1
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
SESSION_MANAGER=local/dt-virtual-machine:@/tmp/.ICE-unix/2131,unix/dt-virtual-machine:/tmp/.ICE-unix/2131
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
GIO_LAUNCHED_DESKTOP_FILE=/home/dt/桌面/code.desktop
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
```

图 9.2.3 测试结果

获取指定环境变量 getenv()

如果只想要获取某个指定的环境变量，可以使用库函数 getenv()，其函数原型如下所示：

```
#include <stdlib.h>

char *getenv(const char *name);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

name：指定获取的环境变量名称。

返回值：如果存放该环境变量，则返回该环境变量的值对应字符串的指针；如果不存在该环境变量，则返回 NULL。

使用 getenv()需要注意，不应该去修改其返回的字符串，修改该字符串意味着修改了环境变量对应的值，Linux 提供了相应的修改函数，如果需要修改环境变量的值应该使用这些函数，不应直接改动该字符串。

使用示例

示例代码 9.2.2 getenv()函数使用示例

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

```
int main(int argc, char *argv[])
{
    const char *str_val = NULL;

    if (2 > argc) {
        fprintf(stderr, "Error:  请传入环境变量名称\n");
        exit(-1);
    }

    /* 获取环境变量 */
    str_val = getenv(argv[1]);
    if (NULL == str_val) {
        fprintf(stderr, "Error:  不存在[%s]环境变量\n", argv[1]);
        exit(-1);
    }

    /* 打印环境变量的值 */
    printf("环境变量的值: %s\n", str_val);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp PATH
环境变量的值: /opt/jdk1.8.0_271/bin:/home/dt/bin:/home/dt/.local/bin:/opt/jdk1.8.0_271/bin:/home/dt/bin:
/home/dt/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/g
ames:/snap/bin
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.2.4 测试结果

9.2.2 添加/删除/修改环境变量

C 语言函数库中提供了用于修改、添加、删除环境变量的函数，譬如 putenv()、setenv()、unsetenv()、clearenv()函数等。

putenv()函数

putenv()函数可向进程的环境变量数组中添加一个新的环境变量，或者修改一个已经存在的环境变量对应的值，其函数原型如下所示：

```
#include <stdlib.h>

int putenv(char *string);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

string：参数 string 是一个字符串指针，指向 name=value 形式的字符串。

返回值：成功返回 0；失败将返回非 0 值，并设置 errno。

该函数调用成功之后，参数 string 所指向的字符串就成为了进程环境变量的一部分了，换言之，putenv()函数将设定 environ 变量（字符串数组）中的某个元素（字符串指针）指向该 string 字符串，而不是指向它的复制副本，这里需要注意！因此，不能随意修改参数 string 所指向的内容，这将影响进程的环境变量，出于这种原因，参数 string 不应为自动变量（即在栈中分配的字符数组）。

测试

使用 putenv()函数为当前进程添加一个环境变量。

示例代码 9.2.3 putenv()函数使用示例

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

int main(int argc, char *argv[])
{
    if (2 > argc) {
        fprintf(stderr, "Error:  传入 name=value\n");
        exit(-1);
    }

    /* 添加/修改环境变量 */
    if (putenv(argv[1])) {
        perror("putenv error");
        exit(-1);
    }

    exit(0);
}
```

setenv()函数

setenv()函数可以替代 putenv()函数，用于向进程的环境变量列表中添加一个新的环境变量或修改现有环境变量对应的值，其函数原型如下所示：

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int overwrite);
```

使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

name：需要添加或修改的环境变量名称。

value：环境变量的值。

overwrite：若参数 name 标识的环境变量已经存在，在参数 overwrite 为 0 的情况下，setenv()函数将不改变现有环境变量的值，也就是说本次调用没有产生任何影响；如果参数 overwrite 的值为非 0，若参数 name 标识的环境变量已经存在，则覆盖，不存在则表示添加新的环境变量。

返回值：成功返回 0；失败将返回-1，并设置 errno。

setenv()函数为形如 name=value 的字符串分配一块内存缓冲区，并将参数 name 和参数 value 所指向的字符串复制到此缓冲区中，以此来创建一个新的环境变量，所以，由此可知，setenv()与 putenv()函数有两个区别：

- putenv()函数并不会为 name=value 字符串分配内存；
- setenv()可通过参数 overwrite 控制是否需要修改现有变量的值而仅以添加变量为目的，显然 putenv()并不能进行控制。

推荐大家使用 setenv()函数，这样使用自动变量作为 setenv()的参数也不会有问题。

使用示例

示例代码 9.2.4 setenv()函数使用示例

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

int main(int argc, char *argv[])
{
    if (3 > argc) {
        fprintf(stderr, "Error:  传入 name value\n");
        exit(-1);
    }

    /* 添加环境变量 */
    if (setenv(argv[1], argv[2], 0)) {
        perror("setenv error");
        exit(-1);
    }

    exit(0);
}
```

除了上面给大家介绍的函数之外，我们还可以通过一种更简单地方式向进程环境变量表中添加环境变量，用法如下：

```
NAME=value ./app
```

在执行程序的时候，在其路径前面添加环境变量，以 name=value 的形式添加，如果是多个环境变量，则在./app 前面放置多对 name=value 即可，以空格分隔。

unsetenv()函数

unsetenv()函数可以从环境变量表中移除参数 name 标识的环境变量，其函数原型如下所示：

```
#include <stdlib.h>

int unsetenv(const char *name);
```

9.2.3 清空环境变量

有时，需要清除环境变量表中的所有变量，然后再进行重建，可以通过将全局变量 environ 赋值为 NULL 来清空所有变量。

```
environ = NULL;
```

也可通过 clearenv()函数来操作，函数原型如下所示：

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

```
int clearenv(void);
```

clearenv()函数内部的做法其实就是将 environ 赋值为 NULL。在某些情况下，使用 setenv()函数和 clearenv()函数可能会导致程序内存泄漏，前面提到过，setenv()函数会为环境变量分配一块内存缓冲区，随之称为进程的一部分；而调用 clearenv()函数时没有释放该缓冲区（clearenv()调用并不知晓该缓冲区的存在，故而也无法将其释放），反复调用者两个函数的程序，会不断产生内存泄漏。

9.2.4 环境变量的作用

环境变量常见的用途之一是在 shell 中，每一个环境变量都有它所表示的含义，譬如 HOME 环境变量表示用户的家目录，USER 环境变量表示当前用户名，SHELL 环境变量表示 shell 解析器名称，PWD 环境变量表示当前所在目录等，在我们自己的应用程序当中，也可以使用进程的环境变量。

9.3 进程的内存布局

历史沿袭至今，C 语言程序一直都是由以下几部分组成的：

- **正文段**。也可称为代码段，这是 CPU 执行的机器语言指令部分，文本段具有只读属性，以防止程序由于意外而修改其指令；正文段是可以共享的，即使在多个进程间也可同时运行同一段程序。
- **初始化数据段**。通常将此段称为数据段，包含了显式初始化的全局变量和静态变量，当程序加载到内存中时，从可执行文件中读取这些变量的值。
- **未初始化数据段**。包含了未进行显式初始化的全局变量和静态变量，通常将此段称为 bss 段，这一名词来源于早期汇编程序中的一个操作符，意思是“由符号开始的块”（block started by symbol），在程序开始执行之前，系统会将本段内所有内存初始化为 0，可执行文件并没有为 bss 段变量分配存储空间，在可执行文件中只需记录 bss 段的位置及其所需大小，直到程序运行时，由加载器来分配这一段内存空间。
- **栈**。函数内的局部变量以及每次函数调用时所需保存的信息都放在此段中，每次调用函数时，函数传递的实参以及函数返回值等也都存放在栈中。栈是一个动态增长和收缩的段，由栈帧组成，系统会为每个当前调用的函数分配一个栈帧，栈帧中存储了函数的局部变量（所谓自动变量）、实参和返回值。
- **堆**。可在运行时动态进行内存分配的一块区域，譬如使用 malloc()分配的内存空间，就是从系统堆内存中申请分配的。

Linux 下的 size 命令可以查看二进制可执行文件的文本段、数据段、bss 段的段大小：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ size testApp
   text    data     bss     dec     hex filename
   1453     560      16    2029     7ed testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.3.1 size 命令

图 9.3.2 显示了这些段在内存中的典型布局方式，当然，并不要求具体的实现一定是以这种方式安排其存储空间，但这是一种便于我们说明的典型方式。

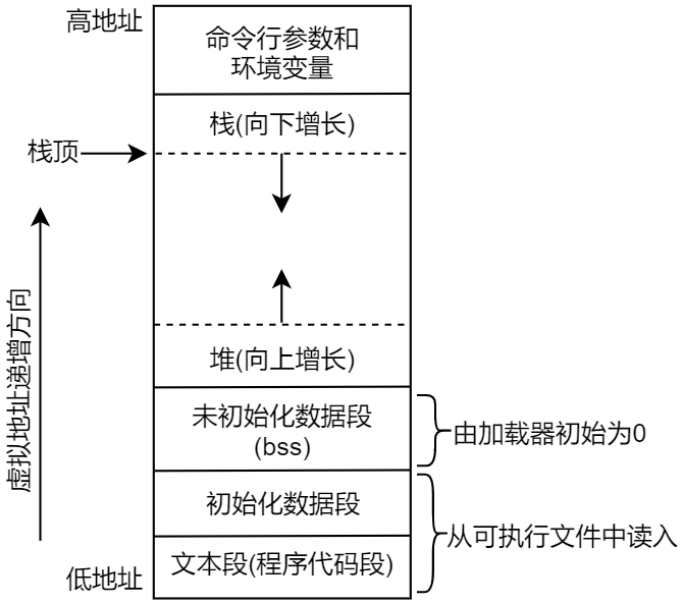


图 9.3.2 在 Linux/x86-32 体系中进程内存布局

9.4 进程的虚拟地址空间

上一小节我们讨论了 C 语言程序的构成以及运行时进程在内存中的布局方式，在 Linux 系统中，采用了虚拟内存管理技术，事实上大多数现在操作系统都是如此！在 Linux 系统中，每一个进程都在自己独立的地址空间中运行，在 32 位系统中，每个进程的逻辑地址空间均为 4GB，这 4GB 的内存空间按照 3:1 的比例进行分配，其中用户进程享有 3G 的空间，而内核独自享有剩下的 1G 空间，如下所示：

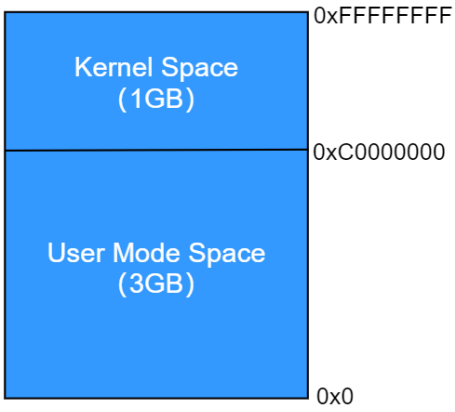


图 9.4.1 Linux 系统下逻辑地址空间划分

学习过驱动开发的读者对“虚拟地址”这个概念应该并不陌生，虚拟地址会通过硬件 MMU（内存管理单元）映射到实际的物理地址空间中，建立虚拟地址到物理地址的映射关系后，对虚拟地址的读写操作实际上就是对物理地址的读写操作，MMU 会将物理地址“翻译”为对应的物理地址，其关系如下所示：

图 9.4.2 虚拟地址到物理地址的映射关系

Linux 系统下，应用程序运行在一个虚拟地址空间中，所以程序中读写的内存地址对应也是虚拟地址，并不是真正的物理地址，譬如应用程序中读写 0x80800000 这个地址，实际上并不对应于硬件的 0x80800000 这个物理地址。

为什么需要引入虚拟地址呢？

计算机物理内存的大小是固定的，就是计算机的实际物理内存，试想一下，如果操作系统没有虚拟地址机制，所有的应用程序访问的内存地址就是实际的物理地址，所以要将所有应用程序加载到内存中，但是我们实际的物理内存只有 4G，所以就会出现一些问题：

- 当多个程序需要运行时，必须保证这些程序用到的内存总量要小于计算机实际的物理内存的大小。
- **内存使用效率低**。内存空间不足时，就需要将其它程序暂时拷贝到硬盘中，然后将新的程序装入内存。然而由于大量的数据装入装出，内存的使用效率就会非常低。
- **进程地址空间不隔离**。由于程序是直接访问物理内存的，所以每一个进程都可以修改其它进程的内存数据，甚至修改内核地址空间中的数据，所以有些恶意程序可以随意修改别的进程，就会造成一些破坏，系统不安全、不稳定。
- **无法确定程序的链接地址**。程序运行时，链接地址和运行地址必须一致，否则程序无法运行！因为程序代码加载到内存的地址是由系统随机分配的，是无法预知的，所以程序的运行地址在编译程序时是无法确认的。

针对以上的一些问题，就引入了虚拟地址机制，程序访问存储器所使用的逻辑地址就是虚拟地址，通过逻辑地址映射到真正的物理内存上。所有应用程序运行在自己的虚拟地址空间中，使得进程的虚拟地址空间和物理地址空间隔离开来，这样做带来了很多的优点：

- 进程与进程、进程与内核相互隔离。一个进程不能读取或修改另一个进程或内核的内存数据，这是因为每一个进程的虚拟地址空间映射到了不同的物理地址空间。提高了系统的安全性与稳定性。
- 在某些应用场合下，两个或者更多进程能够共享内存。因为每个进程都有自己的映射表，可以让不同进程的虚拟地址空间映射到相同的物理地址空间中。通常，共享内存可用于实现进程间通信。
- 便于实现内存保护机制。譬如在多个进程共享内存时，允许每个进程对内存采取不同的保护措施，例如，一个进程可能以只读方式访问内存，而另一进程则能够以可读可写的方式访问。
- 编译应用程序时，无需关心链接地址。前面提到了，当程序运行时，要求链接地址与运行地址一致，在引入了虚拟地址机制后，便无需关心这个问题。

关于本小节的内容就介绍这么多，理解本小节的内容可以帮助我们更好地理解后面小节中将要介绍的内容。

9.5 fork()创建子进程

一个现有的进程可以调用 fork()函数创建一个新的进程，调用 fork()函数的进程称为父进程，由 fork()函数创建出来的进程被称为子进程（child process），fork()函数原型如下所示（fork()为系统调用）：

```
#include <unistd.h>

pid_t fork(void);
```

使用该函数需要包含头文件<unistd.h>。

在诸多的应用中，创建多个进程是任务分解时行之有效的方法，譬如，某一网络服务器进程可在监听客户端请求的同时，为处理每一个请求事件而创建一个新的子进程，与此同时，服务器进程会继续监听更多的客户端连接请求。在一个大型的应用程序任务中，创建子进程通常会简化应用程序的设计，同时提高了系统的并发性（即同时能够处理更多的任务或请求，多个进程在宏观上实现同时运行）。

理解 fork()系统调用的关键在于，完成对其调用后将存在两个进程，一个是原进程（父进程）、另一个则是创建出来的子进程，并且每个进程都会从 fork()函数的返回处继续执行，会导致调用 fork()返回两次值，子进程返回一个值、父进程返回一个值。在程序代码中，可通过返回值来区分是子进程还是父进程。

fork()调用成功后，将会在父进程中返回子进程的 PID，而在子进程中返回值是 0；如果调用失败，父进程返回值-1，不创建子进程，并设置 errno。

fork()调用成功后，子进程和父进程会继续执行 fork()调用之后的指令，子进程、父进程各自在自己的进程空间中运行。事实上，子进程是父进程的一个副本，譬如子进程拷贝了父进程的数据段、堆、栈以及继承了父进程打开的文件描述符，父进程与子进程并不共享这些存储空间，这是子进程对父进程相应部分存储空间的完全复制，执行 fork()之后，每个进程均可修改各自的栈数据以及堆段中的变量，而并不影响另一个进程。

虽然子进程是父进程的一个副本，但是对于程序代码段（文本段）来说，两个进程执行相同的代码段，因为代码段是只读的，也就是说父子进程共享代码段，在内存中只存在一份代码段数据。

使用示例 1

使用 fork()创建子进程。

示例代码 9.5.1 fork()使用示例

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

int main(void)
{
    pid_t pid;

    pid = fork();
    switch (pid) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        printf("这是子进程打印信息<pid: %d, 父进程 pid: %d>\n",
                getpid(), getppid());
        _exit(0);    //子进程使用_exit()退出

    default:
        printf("这是父进程打印信息<pid: %d, 子进程 pid: %d>\n",
                getpid(), pid);
        exit(0);
    }
}
```

上述示例代码中，case 0 是子进程的分支，这里使用了_exit()结束进程而没有使用 exit()。

Tips：C 库函数 exit()建立在系统调用_exit()之上，这两个函数在 3.3 小节中向大家介绍过，这里我们强调，在调用了 fork()之后，父、子进程中一般只有一个会通过调用 exit()退出进程，而另一个则应使用_exit()退出，具体原因将会在后面章节内容中向大家做进一步说明！

直接测试运行查看打印结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
这是父进程打印信息<pid: 46802, 子进程pid: 46803>
这是子进程打印信息<pid: 46803, 父进程pid: 46802>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.5.1 测试结果

从打印结果可知，fork()之后的语句被执行了两次，所以 switch…case 语句被执行了两次，第一次进入到了"case 0"分支，通过上面的介绍可知，fork()返回值为 0 表示当前处于子进程；在子进程中我们通过 getpid()获取到子进程自己的 PID（46802），通过 getppid()获取到父进程的 PID（46803），将其打印出来。

第二次进入到了 default 分支，表示当前处于父进程，此时 fork()函数的返回值便是创建出来的子进程对应的 PID。

fork()函数调用完成之后，父进程、子进程会各自继续执行 fork()之后的指令，最终父进程会执行到 exit()结束进程，而子进程则会通过_exit()结束进程。

使用示例 2

示例代码 9.5.2 fork()函数使用示例 2

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

int main(void)
{
    pid_t pid;

    pid = fork();
    switch (pid) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        printf("这是子进程打印信息\n");
        printf("%d\n", pid);
        _exit(0);

    default:
        printf("这是父进程打印信息\n");
```

```
        printf("%d\n", pid);
        exit(0);
    }
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
这是父进程打印信息
46953
这是子进程打印信息
0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.5.2 测试结果

在 exit()函数之前添加了打印信息，而从上图中可以知道，打印的 pid 值并不相同，0 表示子进程打印出来的，46953 表示的是父进程打印出来的，所以从这里可以证实，fork()函数调用完成之后，父进程、子进程会各自继续执行 fork()之后的指令，它们共享代码段，但并不共享数据段、堆、栈等，而是子进程拥有父进程数据段、堆、栈等副本，所以对于同一个局部变量，它们打印出来的值是不相同的，因为 fork()调用返回值不同，在父、子进程中赋予了 pid 不同的值。

关于子进程

子进程被创建出来之后，便是一个独立的进程，拥有自己独立的进程空间，系统内唯一的进程号，拥有自己独立的 PCB（进程控制块），子进程会被内核同等调度执行，参与到系统的进程调度中。

子进程与父进程之间的这种关系被称为父子进程关系，父子进程关系相比于普通的进程间关系多多少少存在一些关联与“羁绊”，关于这些关联与“羁绊”我们将会在后面的课程中为大家介绍。

Tips：系统调度。Linux 系统是一个多任务、多进程、多线程的操作系统，一般来说系统启动之后会运行成百甚至上千个不同的进程，那么对于单核 CPU 计算机来说，在某一个时间它只能运行某一个进程的代码指令，那其它进程怎么办呢（多核处理器也是如此，同一时间每个核它只能运行某一个进程的代码）？这里就出现了调度的问题，系统是这样做的，每一个进程（或线程）执行一段固定的时间，时间到了之后切换执行下一个进程或线程，依次轮流执行，这就称为调度，由操作系统负责这件事情，当然系统调度的实现本身是一件非常复杂的事情，需要考虑的因素很多，这里只是让大家有个简单地认识，系统调度的基本单元是线程，关于线程，后面章节内容将会向大家介绍。

9.6 父、子进程间的文件共享

调用 fork()函数之后，子进程会获得父进程所有文件描述符的副本，这些副本的创建方式类似于 dup()，这也意味着父、子进程对应的文件描述符均指向相同的文件表，如下图所示：

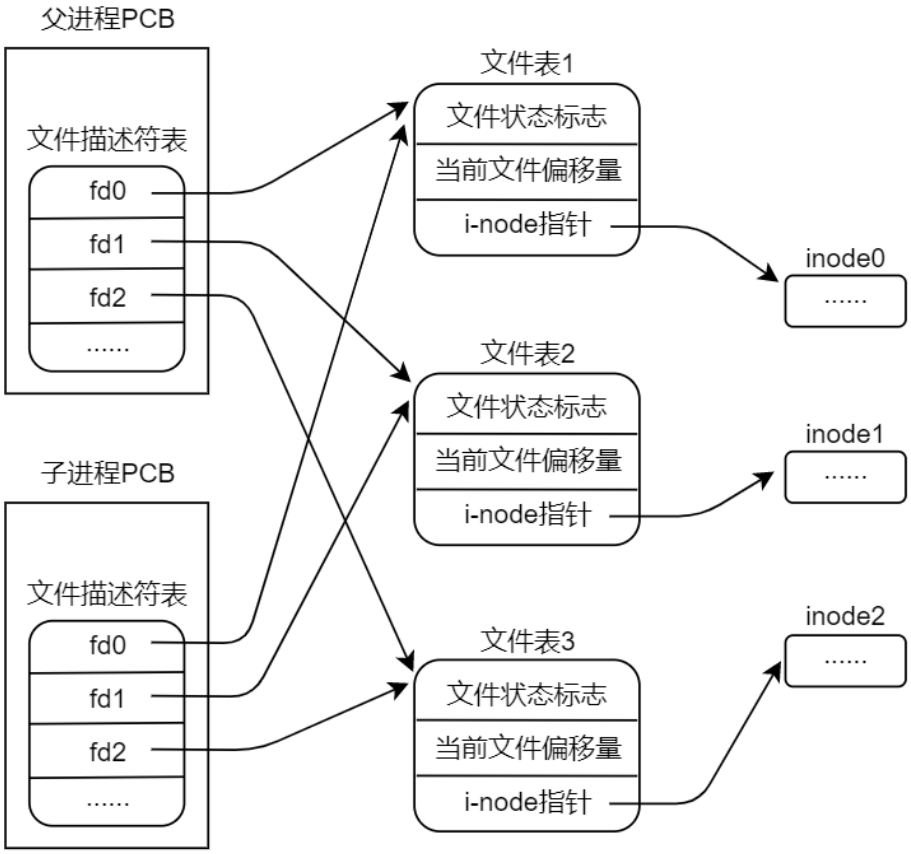


图 9.6.1 父、子进程间的文件共享

由此可知，子进程拷贝了父进程的文件描述符表，使得父、子进程中对应的文件描述符指向了相同的文件表，也意味着父、子进程中对应的文件描述符指向了磁盘中相同的文件，因而这些文件在父、子进程间实现了共享，譬如，如果子进程更新了文件偏移量，那么这个改变也会影响到父进程中相应文件描述符的位置偏移量。

接下来我们进行一个测试，父进程打开文件之后，然后 fork()创建子进程，此时子进程继承了父进程打开的文件描述符（父进程文件描述符的副本），然后父、子进程同时对文件进行写入操作，测试代码如下所示：

示例代码 9.6.1 子进程继承父进程文件描述符实现文件共享

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

int main(void)
{
    pid_t pid;
    int fd;
    int i;
```

```
    fd = open("./test.txt", O_RDWR | O_TRUNC);
    if (0 > fd) {
        perror("open error");
        exit(-1);
    }

    pid = fork();
    switch (pid) {
    case -1:
        perror("fork error");
        close(fd);
        exit(-1);

    case 0:
        /* 子进程 */
        for (i = 0; i < 4; i++) //循环写入 4 次
            write(fd, "1122", 4);
        close(fd);
        _exit(0);

    default:
        /* 父进程 */
        for (i = 0; i < 4; i++) //循环写入 4 次
            write(fd, "AABB", 4);
        close(fd);
        exit(0);
    }
}
```

上述代码中，父进程 open 打开文件之后，才调用 fork()创建了子进程，所以子进程了继承了父进程打开的文件描述符 fd，我们需要验证的便是两个进程对文件的写入操作是分别各自写入、还是每次都在文件末尾接续写入。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test.txt
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test.txt
AABBAABBAABBAABB1122112211221122dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.6.2 测试结果

有上述测试结果可知，此种情况下，父、子进程分别对同一个文件进行写入操作，结果是接续写，不管是父进程，还是子进程，在每次写入时都是从文件的末尾写入，很像使用了 O_APPEND 标志的效果。其原因也非常简单，图 9.6.1 中便给出了答案，子进程继承了父进程的文件描述符，两个文件描述符都指向了一

个相同的文件表，意味着它们的文件偏移量是同一个、绑定在了一起，相互影响，子进程改变了文件的位置偏移量就会作用到父进程，同理，父进程改变了文件的位置偏移量就会作用到子进程。

再来测试另外一种情况，父进程在调用 fork()之后，此时父进程和子进程都去打开同一个文件，然后再对文件进行写入操作，测试代码如下：

示例代码 9.6.2 父、子各自打开同一个文件实现文件共享

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

int main(void)
{
    pid_t pid;
    int fd;
    int i;

    pid = fork();
    switch (pid) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        fd = open("./test.txt", O_WRONLY);
        if (0 > fd) {
            perror("open error");
            _exit(-1);
        }

        for (i = 0; i < 4; i++) //循环写入 4 次
            write(fd, "1122", 4);
        close(fd);
        _exit(0);

    default:
        /* 父进程 */
        fd = open("./test.txt", O_WRONLY);
        if (0 > fd) {
            perror("open error");
            exit(-1);
```

```
        }

        for (i = 0; i < 4; i++) //循环写入 4 次
            write(fd, "AABB", 4);
        close(fd);
        exit(0);
    }
}
```

在上述示例中，父进程调用 fork()之后，然后在父、子进程中都去打开 test.txt 文件，然后在对其进行写入操作，子进程调用了 4 次 write、每次写入“1122”；而父进程调用了 4 次 write、每次写入“AABB”，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ touch test.txt
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat test.txt
1122AABBAABBAABBdt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.6.3 测试结果

从测试结果可知，这种文件共享方式实现的是一种两个进程分别各自对文件进行写入操作，因为父、子进程的这两个文件描述符分别指向的是不同的文件表，意味着它们有各自的文件偏移量，一个进程修改了文件偏移量并不会影响另一个进程的文件偏移量，所以写入的数据会出现覆盖的情况。

fork()函数使用场景

fork()函数有以下两种用法：

- 父进程希望子进程复制自己，使父进程和子进程同时执行不同的代码段。这在网络服务进程中是常见的，父进程等待客户端的服务请求，当接收到客户端发送的请求事件后，调用 fork()创建一个子进程，使子进程去处理此请求、而父进程可以继续等待下一个服务请求。
- 一个进程要执行不同的程序。譬如在程序 app1 中调用 fork()函数创建了子进程，此时子进程是要去执行另一个程序 app2，也就是子进程需要执行的代码是 app2 程序对应的代码，子进程将从 app2 程序的 main 函数开始运行。这种情况，通常在子进程从 fork()函数返回之后立即调用 exec 族函数来实现，关于 exec 函数将在后面内容向大家介绍。

9.7 系统调用 vfork()

除了 fork()系统调用之外，Linux 系统还提供了 vfork()系统调用用于创建子进程，vfork()与 fork()函数在功能上是相同的，并且返回值也相同，在一些细节上存在区别，vfork()函数原型如下所示：

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

pid_t vfork(void);
```

使用该函数需要包含头文件<sys/types.h>和<unistd.h>。

从前面的介绍可知，可以将 fork()认作对父进程的数据段、堆段、栈段以及其它一些数据结构创建拷贝，由此可以看出，使用 fork()系统调用的代价是很大的，它复制了父进程中的数据段和堆栈段中的绝大部分内容，这将会消耗比较多的时间，效率会有所降低，而且太浪费，原因有很多，其中之一在于，fork()函数之后子进程通常会调用 exec 函数，也就是 fork()第二种使用场景下，这使得子进程不再执行父程序中的代码段，而是执行新程序的代码段，从新程序的 main 函数开始执行、并为新程序重新初始化其数据段、堆段、栈段等；那么在这种情况下，子进程并不需要用到父进程的数据段、堆段、栈段（譬如父程序中定义的局部变量、全局变量等）中的数据，此时就会导致浪费时间、效率降低。

事实上，现代 Linux 系统采用了一些技术来避免这种浪费，其中很重要的一点就是内核采用了写时复制（**copy-on-write**）技术，关于这种技术的实现细节就不给大家介绍了，有兴趣读者可以自己搜索相应的文档了解。

出于这一原因，引入了 vfork()系统调用，虽然在一些细节上有所不同，但其效率要高于 fork()函数。类似于 fork()，vfork()可以为调用该函数的进程创建一个新的子进程，然而，vfork()是为子进程立即执行 exec()新的程序而专门设计的，也就是 fork()函数的第二个使用场景。

vfork()与 fork()函数主要有以下两个区别：

- vfork()与 fork()一样都创建了子进程，但 vfork()函数并不会将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 exec（或_exit），于是也就不会引用该地址空间的数据。不过在子进程调用 exec 或_exit 之前，它在父进程的空间中运行、子进程共享父进程的内存。这种优化工作方式的实现提高的效率；但如果子进程修改了父进程的数据（除了 vfork 返回值的变量）、进行了函数调用、或者没有调用 exec 或_exit 就返回将可能带来未知的结果。
- 另一个区别在于，vfork()保证子进程先运行，子进程调用 exec 之后父进程才可能被调度运行。

虽然 vfork()系统调用在效率上要优于 fork()，但是 vfork()可能会导致一些难以察觉的程序 bug，所以尽量避免使用 vfork()来创建子进程，虽然 fork()在效率上并没有 vfork()高，但是现代的 Linux 系统内核已经采用了写时复制技术来实现 fork()，其效率较之于早期的 fork()实现要高出许多，除非速度绝对重要的场合，我们的程序当中应舍弃 vfork()而使用 fork()。

使用示例

示例代码 9.7.1 vfork()函数使用示例

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

int main(void)
{
    pid_t pid;
    int num = 100;

    pid = vfork();
    switch (pid) {
    case -1:
        perror("vfork error");
        exit(-1);

    case 0:
```

```
        /* 子进程 */
        printf("子进程打印信息\n");
        printf("子进程打印 num: %d\n", num);
        _exit(0);

    default:
        /* 父进程 */
        printf("父进程打印信息\n");
        printf("父进程打印 num: %d\n", num);
        exit(0);
    }
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程打印信息
子进程打印num: 100
父进程打印信息
父进程打印num: 100
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.7.1 测试结果

在正式的使用场合下，一般应在子进程中立即调用 exec，如果 exec 调用失败，子进程则应调用_exit()退出（vfork 产生的子进程不应调用 exit 退出，因为这会导致对父进程 stdio 缓冲区的刷新和关闭）。上述示例代码只是一个简单地演示，并不是 vfork()的真正用法，后面学习到 exec 的时候还会再给大家进行介绍。

9.8 fork()之后的竞争条件

调用 fork()之后，子进程成为了一个独立的进程，可被系统调度运行，而父进程也继续被系统调度运行，这里出现了一个问题，调用 fork 之后，无法确定父、子两个进程谁将率先访问 CPU，也就是说无法确认谁先被系统调用运行（在多核处理器中，它们可能会同时各自访问一个 CPU），这将导致谁先运行、谁后运行这个顺序是不确定的，譬如有如下示例代码：

示例代码 9.8.1 fork()竞争条件测试代码

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

int main(void)
{
    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
```

```
        /* 子进程 */
        printf("子进程打印信息\n");
        _exit(0);

    default:
        /* 父进程 */
        printf("父进程打印信息\n");
        exit(0);
    }
}
```

示例代码中我们是无法确认"子进程打印信息"和"父进程打印信息"谁先会被打印出来，有时子进程先被执行，打印出"子进程打印信息"，而有时父进程会先被执行，打印出"子进程打印信息"，测试结果如下所示：

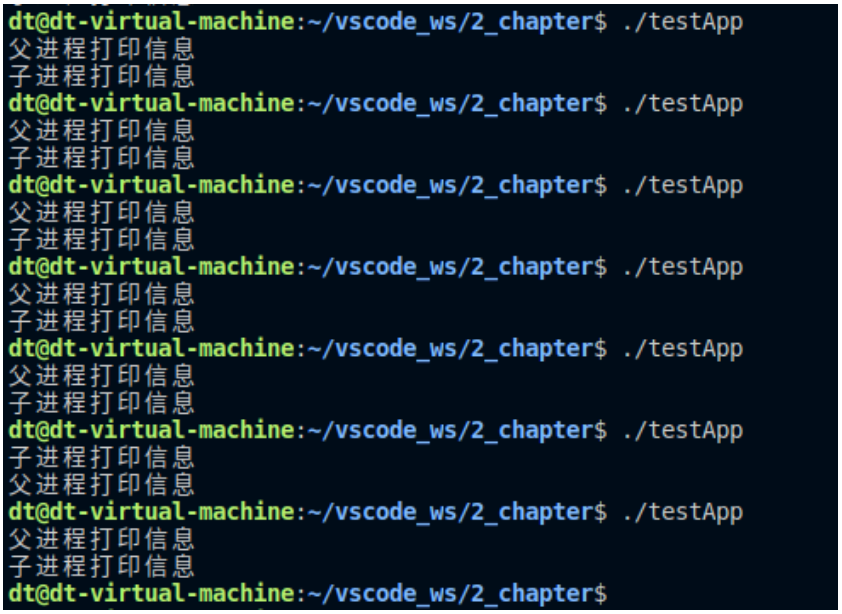


图 9.8.1 测试结果

从测试结果可知，虽然绝大部分情况下，父进程会先于子进程被执行，但是并不排除子进程先于父进程被执行的可能性。而对于有些特定的应用程序，它对于执行的顺序有一定要求的，譬如它必须要求父进程先运行，或者必须要求子进程先运行，程序产生正确的结果它依赖于特定的执行顺序，那么将可能因竞争条件而导致失败、无法得到正确的结果。

那如何明确保证某一特性执行顺序呢？这个时候可以通过采用采用某种同步技术来实现，譬如前面给大家介绍的信号，如果要让子进程先运行，则可使父进程被阻塞，等到子进程来唤醒它，示例代码如下所示：

示例代码 9.8.2 利用信号来调整进程间动作

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

static void sig_handler(int sig)
{
```

```
    printf("接收到信号\n");
}

int main(void)
{
    struct sigaction sig = {0};
    sigset_t wait_mask;

    /* 初始化信号集 */
    sigemptyset(&wait_mask);

    /* 设置信号处理方式 */
    sig.sa_handler = sig_handler;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGUSR1, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        printf("子进程开始执行\n");
        printf("子进程打印信息\n");
        printf("~~~~~~~~~~~~~~~\n");
        sleep(2);
        kill(getppid(), SIGUSR1);     //发送信号给父进程、唤醒它
        _exit(0);

    default:
        /* 父进程 */
        if (-1 != sigsuspend(&wait_mask))//挂起、阻塞
            exit(-1);

        printf("父进程开始执行\n");
        printf("父进程打印信息\n");
        exit(0);
    }
}
```

示例代码比较简单，这里我们希望子进程先运行打印相应信息，之后再执行父进程打印信息，在父进程分支中，直接调用了 sigsuspend()使父进程进入挂起状态，由子进程通过 kill 命令发送信号唤醒，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程开始执行
子进程打印信息
~~~~~~~~~~~~~~~~~
接收到信号
父进程开始执行
父进程打印信息
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.8.2 测试结果

9.9 进程的诞生与终止

9.9.1 进程的诞生

一个进程可以通过 fork()或 vfork()等系统调用创建一个子进程，一个新的进程就此诞生！事实上，Linux 系统下的所有进程都是由其父进程创建而来，譬如在 shell 终端通过命令的方式执行一个程序./app，那么 app 进程就是由 shell 终端进程创建出来的，shell 终端就是该进程的父进程。

既然所有进程都是由其父进程创建出来的，那么总有一个最原始的父进程吧，否则其它进程是怎么创建出来的呢？确实如此，在 Ubuntu 系统下使用"ps -aux"命令可以查看到系统下所有进程信息，如下：

```
dt@dt-virtual-machine:~$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1 185416  4664 ?        Ss   3月09   0:29 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    3月09   0:00 [kthreadd]
root           4  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/0:0H]
root           6  0.0  0.0      0     0 ?        I<   3月09   0:00 [mm_percpu_wq]
root           7  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/0]
root           8  0.0  0.0      0     0 ?        I    3月09  11:19 [rcu_sched]
root           9  0.0  0.0      0     0 ?        I    3月09   0:00 [rcu_bh]
root          10  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/0]
root          11  0.0  0.0      0     0 ?        S    3月09   0:05 [watchdog/0]
root          12  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/0]
root          13  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/1]
root          14  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/1]
root          15  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/1]
root          16  0.0  0.0      0     0 ?        S    3月09   0:00 [ksoftirqd/1]
root          18  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/1:0H]
root          19  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/2]
root          20  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/2]
root          21  0.0  0.0      0     0 ?        S    3月09   0:00 [migration/2]
root          22  0.0  0.0      0     0 ?        S    3月09   0:01 [ksoftirqd/2]
root          24  0.0  0.0      0     0 ?        I<   3月09   0:00 [kworker/2:0H]
root          25  0.0  0.0      0     0 ?        S    3月09   0:00 [cpuhp/3]
root          26  0.0  0.0      0     0 ?        S    3月09   0:04 [watchdog/3]
```

图 9.9.1 查看所有进程信息

上图中进程号为 1 的进程便是所有进程的父进程，通常称为 init 进程，它是 Linux 系统启动之后运行的第一个进程，它管理着系统上所有其它进程，init 进程是由内核启动，因此理论上说它没有父进程。

init 进程的 PID 总是为 1，它是所有子进程的父进程，一切从 1 开始、一切从 init 进程开始！

一个进程的生命周期便是从创建开始直至其终止。

9.9.2 进程的终止

通常，进程有两种终止方式：异常终止和正常终止，分别在 3.3 小节和 8.12 小节中给大家介绍过，如前所述，进程的正常终止有多种不同的方式，譬如在 main 函数中使用 return 返回、调用 exit()函数结束进程、调用_exit()或_Exit()函数结束进程等。

异常终止通常也有多种不同的方式，譬如在程序当中调用 abort()函数异常终止进程、当进程接收到某些信号导致异常终止等。

_exit()函数和 exit()函数的 status 参数定义了进程的终止状态（termination status），父进程可以调用 wait()函数以获取该状态。虽然参数 status 定义为 int 类型，但仅有低 8 位表示它的终止状态，一般来说，终止状态为 0 表示进程成功终止，而非 0 值则表示进程在执行过程中出现了一些错误而终止，譬如文件打开失败、读写失败等等，对非 0 返回值的解析并无定例。

在我们的程序当中，一般使用 exit()库函数而非_exit()系统调用，原因在于 exit()最终也会通过_exit()终止进程，但在此之前，它将会完成一些其它的工作，exit()函数会执行的动作如下：

- 如果程序中注册了进程终止处理函数，那么会调用终止处理函数。在 9.1.2 小节给大家介绍如何注册进程的终止处理函数；
- 刷新 stdio 流缓冲区。关于 stdio 流缓冲区的问题，稍后编写一个简单地测试程序进行说明；
- 执行_exit()系统调用。

所以，由此可知，exit()函数会比_exit()会多做一些事情，包括执行终止处理函数、刷新 stdio 流缓冲以及调用_exit()，在前面曾提到过，在我们的程序当中，父、子进程不应都使用 exit()终止，只能有一个进程使用 exit()、而另一个则使用_exit()退出，当然一般推荐的是子进程使用_exit()退出、而父进程则使用 exit()退出。其原因就在于调用 exit()函数终止进程时会刷新进程的 stdio 缓冲区。接下来我们便通过一个示例代码进行说明：

示例代码 9.9.1 exit()之 stdio 缓冲测试代码 1

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

int main(void)
{
    printf("Hello World!\n");

    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        exit(0);

    default:
        /* 父进程 */
        exit(0);
    }
```

```
}
```

在上述代码中，在 fork()创建子进程之前，我们通过 printf()打印了一行包括换行符\n 在内字符串，在 fork()创建子进程之后，都使用 exit()退出进程，正常的情况下程序就只会打印一行"Hello World!"，这是一个正常的情况，事实上也确实如此，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.9.2 测试结果

打印结果确实如我们所料，接下来将代码进行简单地修改，把 printf()打印的字符串最后面的换行符\n 去掉，如下所示：

示例代码 9.9.2 exit()之 stdio 缓冲测试代码 2

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

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

    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        exit(0);

    default:
        /* 父进程 */
        exit(0);
    }
}
```

printf 中将字符串后面的\n 换行符给去掉了，接下再进行测试，结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Hello World!Hello World!dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.9.3 测试结果

从打印结果可知，"Hello World!"被打印了两次，这是怎么回事呢？在程序当中明明只使用了 printf 打印了一次字符串。要解释这个问题，首先要知道，进程的用户空间内存中维护了 stdio 缓冲区，0 小节给大家介绍过，因此通过 fork()创建子进程时会复制这些缓冲区。标准输出设备默认使用的是行缓冲，当检测到换行符\n 时会立即显示函数 printf()输出的字符串，在示例代码 9.9.1 中 printf 输出的字符串中包含了换行符，所以会立即读走缓冲区中的数据并显示，读走之后此时缓冲区就空了，子进程虽然拷贝了父进程的缓冲区，但是空的，虽然父、子进程使用 exit()退出时会刷新各自的缓冲区，但对于空缓冲区自然无数据可读。

而对于示例代码 9.9.2 来说，printf()并没有添加换行符\n，当调用 printf()时并不会立即读取缓冲区中的数据进行显示，由此 fork()之后创建的子进程也自然拷贝了缓冲区的数据，当它们调用 exit()函数时，都会刷新各自的缓冲区、显示字符串，所以就会看到打印出了两次相同的字符串。

可以采用以下任一方法来避免重复的输出结果：

- 对于行缓冲设备，可以加上对应换行符，譬如 printf 打印输出字符串时在字符串后面添加\n 换行符，对于 puts()函数来说，本身会自动添加换行符；
- 在调用 fork()之前，使用函数 fflush()来刷新 stdio 缓冲区，当然，作为另一种选择，也可以使用 setvbuf()和 setbuf()来关闭 stdio 流的缓冲功能，这些内容在 3.11 中已经给大家介绍过；
- 子进程调用_exit()退出进程、而非使用 exit()，调用_exit()在退出时便不会刷新 stdio 缓冲区，这也解释前面为什么我们要在子进程中使用_exit()退出这样做的一个原因。将示例代码 9.9.2 中子进程的退出操作 exit()替换成_exit()进行测试，打印的结果便只会显示一次字符串，大家自己动手试一试！

关于本小节的内容，到这里就结束了，虽然笔者觉得自己已经介绍得很详细了，如果大家觉得还有不懂的地方，可以自己编写程序进行测试、验证，编程是一门动手实践性很强的工作，大家要善于从中发现一些问题，然后自己能够编写程序进行测试、验证，大家加油！

9.10 监视子进程

在很多应用程序的设计中，父进程需要知道子进程于何时被终止，并且需要知道子进程的终止状态信息，是正常终止、还是异常终止亦或者被信号终止等，意味着父进程会对子进程进行监视，本小节我们就来学习下如何通过系统调用 wait()以及其它变体来监视子进程的状态改变。

9.10.1 wait()函数

对于许多需要创建子进程的进程来说，有时设计需要监视子进程的终止时间以及终止时的一些状态信息，在某些设计需求下这是很有必要的。系统调用 wait()可以等待进程的任一子进程终止，同时获取子进程的终止状态信息，其函数原型如下所示：

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

pid_t wait(int *status);
```

使用该函数需要包含头文件<sys/types.h>和<sys/wait.h>。

函数参数和返回值含义如下：

status：参数 status 用于存放子进程终止时的状态信息，参数 status 可以为 NULL，表示不接收子进程终止时的状态信息。

返回值：若成功则返回终止的子进程对应的进程号；失败则返回-1。

系统调用 wait()将执行如下动作：

- 调用 wait()函数，如果其所有子进程都还在运行，则 wait()会一直阻塞等待，直到某一个子进程终止；

- 如果进程调用 wait()，但是该进程并没有子进程，也就意味着该进程并没有需要等待的子进程，那么 wait()将返回错误，也就是返回-1、并且会将 errno 设置为 ECHILD。
- 如果进程调用 wait()之前，它的子进程当中已经有一个或多个子进程已经终止了，那么调用 wait()也不会阻塞。wait()函数的作用除了获取子进程的终止状态信息之外，更重要的一点，就是回收子进程的一些资源，俗称为子进程“收尸”，关于这个问题后面再给大家进行介绍。所以在调用 wait()函数之前，已经有子进程终止了，意味着正等待着父进程为其“收尸”，所以调用 wait()将不会阻塞，而是会立即替该子进程“收尸”、处理它的“后事”，然后返回到正常的程序流程中，一次 wait()调用只能处理一次。

参数 status 不为 NULL 的情况下，则 wait()会将子进程的终止时的状态信息存储在它指向的 int 变量中，可以通过以下宏来检查 status 参数：

- **WIFEXITED(status)：**如果子进程正常终止，则返回 true；
- **WEXITSTATUS(status)：**返回子进程退出状态，是一个数值，其实就是子进程调用_exit()或 exit()时指定的退出状态；wait()获取得到的 status 参数并不是调用_exit()或 exit()时指定的状态，可通过 WEXITSTATUS 宏转换；
- **WIFSIGNALED(status)：**如果子进程被信号终止，则返回 true；
- **WTERMSIG(status)：**返回导致子进程终止的信号编号。如果子进程是被信号所终止，则可以通过此宏获取终止子进程的信号；
- **WCOREDUMP(status)：**如果子进程终止时产生了核心转储文件，则返回 true；

还有一些其它的宏定义，这里就不给一一介绍了，具体的请查看 man 手册。

使用示例

示例代码 9.10.1 wait()函数使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(void)
{
    int status;
    int ret;
    int i;

    /* 循环创建 3 个子进程 */
    for (i = 1; i <= 3; i++) {
        switch (fork()) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
            /* 子进程 */
```

```
                printf("子进程<%d>被创建\n", getpid());
                sleep(i);
                _exit(i);

            default:
                /* 父进程 */
                break;
            }
        }

        sleep(1);
        printf("~~~~~~~~~~~~~~~\n");
        for (i = 1; i <= 3; i++) {

            ret = wait(&status);
            if (-1 == ret) {
                if (ECHILD == errno) {
                    printf("没有需要等待回收的子进程\n");
                    exit(0);
                }
                else {
                    perror("wait error");
                    exit(-1);
                }
            }

            printf("回收子进程<%d>, 终止状态<%d>\n", ret,
                    WEXITSTATUS(status));
        }

        exit(0);
}
```

示例代码中，通过 for 循环创建了 3 个子进程，父进程中循环调用 wait()函数等待回收子进程，并将本次回收的子进程进程号以及终止状态打印出来，编译测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<109075>被创建
子进程<109076>被创建
子进程<109077>被创建
~~~~~~~~~~~~~~~
回收子进程<109075>, 终止状态<1>
回收子进程<109076>, 终止状态<2>
回收子进程<109077>, 终止状态<3>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.1 测试结果

9.10.2 waitpid()函数

使用 wait()系统调用存在着一些限制，这些限制包括如下：

- 如果父进程创建了多个子进程，使用 wait()将无法等待某个特定的子进程的完成，只能按照顺序等待下一个子进程的终止，一个一个来、谁先终止就先处理谁；
- 如果子进程没有终止，正在运行，那么 wait()总是保持阻塞，有时我们希望执行非阻塞等待，是否有子进程终止，通过判断即可得知；
- 使用 wait()只能发现那些被终止的子进程，对于子进程因某个信号（譬如 SIGSTOP 信号）而停止（注意，这里停止指的暂停运行），或是已停止的子进程收到 SIGCONT 信号后恢复执行的情况就无能为力了。

而设计 waitpid()则可以突破这些限制，waitpid()系统调用函数原型如下所示：

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

pid_t waitpid(pid_t pid, int *status, int options);
```

使用该函数需要包含头文件<sys/types.h>和<sys/wait.h>。

函数参数和返回值含义如下：

pid：参数 pid 用于表示需要等待的某个具体子进程，关于参数 pid 的取值范围如下：

- 如果 pid 大于 0，表示等待进程号为 pid 的子进程；
- 如果 pid 等于 0，则等待与调用进程（父进程）同一个进程组的所有子进程；
- 如果 pid 小于-1，则会等待进程组标识符与 pid 绝对值相等的所有子进程；
- 如果 pid 等于-1，则等待任意子进程。wait(&status)与 waitpid(-1, &status, 0)等价。

status：与 wait()函数的 status 参数意义相同。

options：稍后介绍。

返回值：返回值与 wait()函数的返回值意义基本相同，在参数 options 包含了 WNOHANG 标志的情况下，返回值会出现 0，稍后介绍。

参数 options 是一个位掩码，可以包括 0 个或多个如下标志：

- **WNOHANG：**如果子进程没有发生状态改变（终止、暂停），则立即返回，也就是执行非阻塞等待，可以实现轮训 poll，通过返回值可以判断是否有子进程发生状态改变，若返回值等于 0 表示没有发生改变。
- **WUNTRACED：**除了返回终止的子进程的状态信息外，还返回因信号而停止（暂停运行）的子进程状态信息；

- **WCONTINUED**：返回那些因收到 SIGCONT 信号而恢复运行的子进程的状态信息。

从以上的介绍可知，waitpid()在功能上要强于 wait()函数，它弥补了 wait()函数所带来的一些限制，具体在实际的编程使用当中，可根据自己的需求进行选择。

使用示例

使用 waitpid()替换 wait()，改写示例代码 9.10.1。

示例代码 9.10.2 waitpid()阻塞方式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(void)
{
    int status;
    int ret;
    int i;

    /* 循环创建 3 个子进程 */
    for (i = 1; i <= 3; i++) {
        switch (fork()) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
            /* 子进程 */
            printf("子进程<%d>被创建\n", getpid());
            sleep(i);
            _exit(i);

        default:
            /* 父进程 */
            break;
        }
    }

    sleep(1);
    printf("~~~~~~~~~~~~~~~~\n");
    for (i = 1; i <= 3; i++) {

        ret = waitpid(-1, &status, 0);
```

```
        if (-1 == ret) {
            if (ECHILD == errno) {
                printf("没有需要等待回收的子进程\n");
                exit(0);
            }
            else {
                perror("wait error");
                exit(-1);
            }
        }

        printf("回收子进程<%d>, 终止状态<%d>\n", ret,
                WEXITSTATUS(status));
    }

    exit(0);
}
```

将 wait(&status)替换成了 waitpid(-1, &status, 0)，通过上面的介绍可知，waitpid()函数的这种参数配置情况与 wait()函数是完全等价的，运行结果与示例代码 9.10.1 运行结果相同，这里不再演示！

将上述代码进行简单修改，将其修改成轮训方式，如下所示：

示例代码 9.10.3 waitpid()轮训方式

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(void)
{
    int status;
    int ret;
    int i;

    /* 循环创建 3 个子进程 */
    for (i = 1; i <= 3; i++) {
        switch (fork()) {
        case -1:
            perror("fork error");
            exit(-1);

        case 0:
            /* 子进程 */
```

```
            printf("子进程<%d>被创建\n", getpid());
            sleep(i);
            _exit(i);

        default:
            /* 父进程 */
            break;
        }
    }

    sleep(1);
    printf("~~~~~~~~~~~~~~~\n");
    for ( ; ; ) {

        ret = waitpid(-1, &status, WNOHANG);
        if (0 > ret) {
            if (ECHILD == errno)
                exit(0);
            else {
                perror("wait error");
                exit(-1);
            }
        }
        else if (0 == ret)
            continue;
        else
            printf("回收子进程<%d>, 终止状态<%d>\n", ret,
                    WEXITSTATUS(status));
    }

    exit(0);
}
```

将 waitpid()函数的 options 参数添加 WNOHANG 标志，将 waitpid()配置成非阻塞模式，使用轮训的方式依次回收各个子进程，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<111741>被创建
子进程<111742>被创建
子进程<111743>被创建
~~~~~~~~~~~~~~~~
回收子进程<111741>, 终止状态<1>
回收子进程<111742>, 终止状态<2>
回收子进程<111743>, 终止状态<3>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.2 测试结果

9.10.3 waitid()函数

除了以上给大家介绍的 wait()和 waitpid()系统调用之外，还有一个 waitid()系统调用，waitid()与 waitpid()类似，不过 waitid()提供了更多的扩展功能，具体的使用方法笔者便不再介绍，大家有兴趣可以自己通过 man 进行学习。

9.10.4 僵尸进程与孤儿进程

当一个进程创建子进程之后，它们俩就成为父子进程关系，父进程与子进程的生命周期往往是不相同的，这里就会出现两个问题：

- 父进程先于子进程结束。
- 子进程先于父进程结束。

本小节我们就来讨论下这两种不同的情况。

孤儿进程

父进程先于子进程结束，也就是意味着，此时子进程变成了一个“孤儿”，我们把这种进程就称为孤儿进程。在 Linux 系统当中，所有的孤儿进程都自动成为 init 进程（进程号为 1）的子进程，换言之，某一子进程的父进程结束后，该子进程调用 getppid()将返回 1，init 进程变成了孤儿进程的“养父”；这是判定某一子进程的“生父”是否还“在世”的方法之一，通过下面的代码进行测试：

示例代码 9.10.4 孤儿进程测试

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

int main(void)
{
    /* 创建子进程 */
    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
```

```
        /* 子进程 */
        printf("子进程<%d>被创建, 父进程<%d>\n", getpid(), getppid());
        sleep(3);    //休眠 3 秒钟等父进程结束
        printf("父进程<%d>\n", getppid());//再次获取父进程 pid
        _exit(0);

    default:
        /* 父进程 */
        break;
    }

    sleep(1);//休眠 1 秒
    printf("父进程结束!\n");
    exit(0);
}
```

在上述代码中，子进程休眠 3 秒钟，保证父进程先结束，而父进程休眠 1 秒钟，保证子进程能够打印出第一个 printf()，也就是在父进程结束前，打印子进程的父进程进程号；子进程 3 秒休眠时间过后，再次打印父进程的进程号，此时它的“生父”已经结束了。

我们来看看打印结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<112505>被创建, 父进程<112504>
父进程结束!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ 父进程<1911>

dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.3 测试结果

可以发现，打印结果并不是 1，意味着并不是 init 进程，而是 1911，这是怎么回事呢？通过"ps -axu"查询可知，进程号 1911 对应的是 upstart 进程，如下所示：

```
dt         1903  0.0  0.0  45320  1672 ?        Ss   3月10   0:01 /lib/systemd/systemd --user
dt         1904  0.0  0.0 210960   120 ?        S    3月10   0:00 (sd-pam)
dt         1909  0.0  0.0 207516  3664 ?        Sl   3月10   0:03 /usr/bin/gnome-keyring-daemon --daemonize --login
dt         1911  0.0  0.0  48372  3640 ?        Ss   3月10   0:06 /sbin/upstart --user
dt         1993  0.0  0.0  34668   464 ?        S    3月10   0:00 upstart-udev-bridge --daemon --user
dt         1996  0.0  0.0  43748  1848 ?        Ss   3月10   3:19 dbus-daemon --fork --session --address=unix:abstr
```

图 9.10.4 upstart 进程

事实上，/sbin/upstart 进程与 Ubuntu 系统图形化界面有关系，是图形化界面下的一个后台守护进程，可负责“收养”孤儿进程，所以图形化界面下，upstart 进程就自动成为了孤儿进程的父进程，这里笔者是在 Ubuntu 16.04 版本下进行的测试，可能不同的版本这里看到的结果会有不同。

既然在图形化界面下孤儿进程的父进程不是 init 进程，那么我们进入 Ubuntu 字符界面，按 Ctrl + Alt + F1 进入，如下所示：

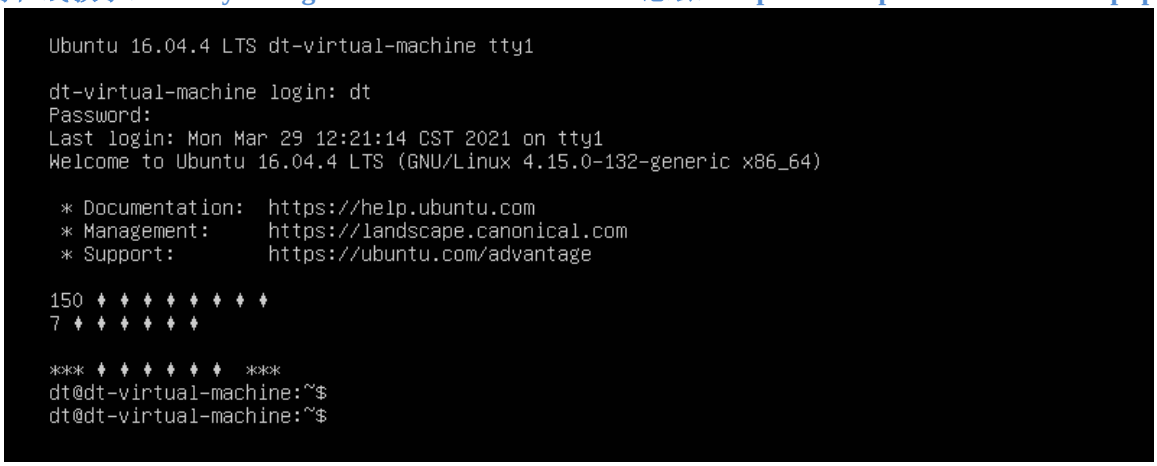


图 9.10.5 Ubuntu 字符界面

输入 Linux 用户名和密码登录，我们在运行一次：

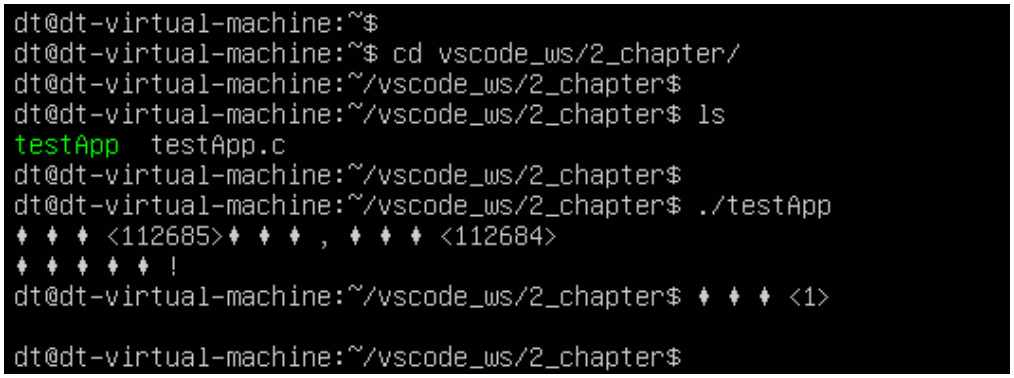


图 9.10.6 测试结果

字符界面模式下无法显示中文，所以出现了很多白色小方块，从打印结果可以发现，此时孤儿进程的父进程就成了 init 进程，大家可以自己测试下，按 Ctrl + Alt + F7 回到 Ubuntu 图形化界面。

僵尸进程

进程结束之后，通常需要其父进程为其“收尸”，回收子进程占用的一些内存资源，父进程通过调用 wait()（或其变体 waitpid()、waitid()等）函数回收子进程资源，归还给系统。

如果子进程先于父进程结束，此时父进程还未来得及给子进程“收尸”，那么此时子进程就变成了一个僵尸进程。子进程结束后其父进程并没有来得及立马给它“收尸”，子进程处于“曝尸荒野”的状态，在这么一个状态下，我们就将子进程成为僵尸进程；至于名字由来，肯定是对电影情节的一种效仿！

当父进程调用 wait()（或其变体，下文不再强调）为子进程“收尸”后，僵尸进程就会被内核彻底删除。另外一种情况，如果父进程并没有调用 wait()函数然后就退出了，那么此时 init 进程将会接管它的子进程并自动调用 wait()，故而从系统中移除僵尸进程。

如果父进程创建了某一子进程，子进程已经结束，而父进程还在正常运行，但父进程并未调用 wait()回收子进程，此时子进程变成一个僵尸进程。首先来说，这样的程序设计是有问题的，如果系统中存在大量的僵尸进程，它们势必会填满内核进程表，从而阻碍新进程的创建。需要注意的是，僵尸进程是无法通过信号将其杀死的，即使是“一击必杀”信号 SIGKILL 也无法将其杀死，那么这种情况下，只能杀死僵尸进程的父进程（或等待其父进程终止），这样 init 进程将会接管这些僵尸进程，从而将它们从系统中清理掉！所以，在我们的一个程序设计中，一定要监视子进程的状态变化，如果子进程终止了，要调用 wait()将其回收，避免僵尸进程。

示例代码

编写示例代码，产生一个僵尸进程。

示例代码 9.10.5 产生僵尸进程

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

int main(void)
{
    /* 创建子进程 */
    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        printf("子进程<%d>被创建\n", getpid());
        sleep(1);
        printf("子进程结束\n");
        _exit(0);

    default:
        /* 父进程 */
        break;
    }

    for ( ; ; )
        sleep(1);

    exit(0);
}
```

在上述代码中，子进程已经退出，但其父进程并没调用 wait()为其“收尸”，使得子进程成为一个僵尸进程，使用命令"ps -aux"可以查看到该僵尸进程，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[2] 113455
子进程<113456>被创建
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ 子进程结束
root      113241  0.0  0.0      0     0 ?        I    14:54   0:00 [kworker/u256:1]
root      113307  0.0  0.0      0     0 ?        I    15:09   0:00 [kworker/u256:0]
root      113356  0.0  0.0      0     0 ?        I    15:18   0:00 [kworker/u256:2]
root      113363  0.0  0.0      0     0 ?        I    15:19   0:00 [kworker/2:2]
root      113432  0.0  0.0      0     0 ?        I    15:29   0:00 [kworker/2:0]
dt        113455  0.0  0.0   4220   644 pts/19   S    15:31   0:00 ./testApp
dt        113456  0.0  0.0      0     0 pts/19   Z    15:31   0:00 [testApp] <defunct>
dt        113459  0.0  0.0  39104  3336 pts/19   R+   15:32   0:00 ps -aux
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.7 测试结果

通过命令可以查看到子进程 113456 依然存在，可以看到它的状态栏显示的是“Z”（zombie，僵尸），表示它是一个僵尸进程。僵尸进程无法被信号杀死，大家可以试试，要么等待其父进程终止、要么杀死其父进程，让 init 进程来处理，当我们杀死其父进程之后，僵尸进程也会被随之清理。

9.10.5 SIGCHLD 信号

SIGCHLD 信号在第八章中给大家介绍过，当发生以下两种情况时，父进程会收到该信号：

- 当父进程的某个子进程终止时，父进程会收到 SIGCHLD 信号；
- 当父进程的某个子进程因收到信号而停止（暂停运行）或恢复时，内核也可能向父进程发送该信号。

子进程的终止属于异步事件，父进程事先是无法预知的，如果父进程有自己需要做的事情，它不能一直 wait()阻塞等待子进程终止（或轮训），这样父进程将啥事也做不了，那么有什么办法来解决这样的尴尬情况，当然有办法，那就是通过 SIGCHLD 信号。

那既然子进程状态改变时（终止、暂停或恢复），父进程会收到 SIGCHLD 信号，SIGCHLD 信号的系统默认处理方式是将其忽略，所以我们要捕获它、绑定信号处理函数，在信号处理函数中调用 wait()收回子进程，回收完毕之后再回到父进程自己的工作流程中。

不过，使用这一方式时需要掌握一些窍门！

由 8.4.1 和 8.4.2 小节的介绍可知，当调用信号处理函数时，会暂时将引发调用的信号添加到进程的信号掩码中（除非 sigaction()指定了 SA_NODEFER 标志），这样一来，当 SIGCHLD 信号处理函数正在为一个终止的子进程“收尸”时，如果相继有两个子进程终止，即使产生了两次 SIGCHLD 信号，父进程也只能捕获到一次 SIGCHLD 信号，结果是，父进程的 SIGCHLD 信号处理函数每次只调用一次 wait()，那么就会导致有些僵尸进程成为“漏网之鱼”。

解决方案就是：在 SIGCHLD 信号处理函数中循环以非阻塞方式来调用 waitpid()，直至再无其它终止的子进程需要处理为止，所以，通常 SIGCHLD 信号处理函数内部代码如下所示：

```
while (waitpid(-1, NULL, WNOHANG) > 0)
    continue;
```

上述代码一直循环下去，直至 waitpid()返回 0，表明再无僵尸进程存在；或者返回-1，表明有错误发生。应在创建任何子进程之前，为 SIGCHLD 信号绑定处理函数。

使用示例

通过 SIGCHLD 信号实现异步方式监视子进程。

示例代码 9.10.6 异步方式监视 wait 回收子进程

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

static void wait_child(int sig)
{
    /* 替子进程收尸 */
    printf("父进程回收子进程\n");
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
}

int main(void)
{
    struct sigaction sig = {0};

    /* 为 SIGCHLD 信号绑定处理函数 */
    sigemptyset(sig.sa_mask);
    sig.sa_handler = wait_child;
    sig.sa_flags = 0;
    if (-1 == sigaction(SIGCHLD, &sig, NULL)) {
        perror("sigaction error");
        exit(-1);
    }

    /* 创建子进程 */
    switch (fork()) {
    case -1:
        perror("fork error");
        exit(-1);

    case 0:
        /* 子进程 */
        printf("子进程<%d>被创建\n", getpid());
        sleep(1);
        printf("子进程结束\n");
        _exit(0);
```

```
    default:
        /* 父进程 */
        break;
    }

    sleep(3);
    exit(0);
}
```

运行结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子进程<113989>被创建
子进程结束
父进程回收子进程
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.10.8 测试结果

9.11 执行新程序

在前面已经大家提到了 exec 函数，当子进程的工作不再是运行父进程的代码段，而是运行另一个新程序的代码，那么这个时候子进程可以通过 exec 函数来实现运行另一个新的程序。本小节我们就来学习下，如何在程序中运行一个新的程序，从新程序的 main()函数开始运行。

9.11.1 execve()函数

系统调用 execve()可以将新程序加载到某一进程的内存空间，通过调用 execve()函数将一个外部的可执行文件加载到进程的内存空间运行，使用新的程序替换旧的程序，而进程的栈、数据、以及堆数据会被新程序的相应部件所替换，然后从新程序的 main()函数开始执行。

execve()函数原型如下所示：

```
#include <unistd.h>

int execve(const char *filename, char *const argv[], char *const envp[]);
```

使用该函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

filename：参数 filename 指向需要载入当前进程空间的新程序的路径名，既可以是绝对路径、也可以是相对路径。

argv：参数 argv 则指定了传递给新程序的命令行参数。是一个字符串数组，该数组对应于 main(int argc, char *argv[])函数的第二个参数 argv，且格式也与之相同，是由字符串指针所组成的数组，以 NULL 结束。argv[0]对应的便是新程序自身路径名。

envp：参数 envp 也是一个字符串指针数组，指定了新程序的环境变量列表，参数 envp 其实对应于新程序的 environ 数组，同样也是以 NULL 结束，所指向的字符串格式为 name=value。

返回值：execve 调用成功将不会返回；失败将返回-1，并设置 errno。

对 execve()的成功调用将永不返回，而且也无需检查它的返回值，实际上，一旦该函数返回，就表明它发生了错误。

基于系统调用 execve()，还提供了一系列以 exec 为前缀命名的库函数，虽然函数参数各异，当其功能相同，通常将这些函数（包括系统调用 execve()）称为 exec 族函数，所以 exec 函数并不是指某一个函数、而是 exec 族函数，下一小节将会向大家介绍这些库函数。

通常将调用这些 exec 函数加载一个外部新程序的过程称为 exec 操作。

使用示例

编写一个简单地程序，在测试程序 testApp 当中通过 execve()函数运行另一个新程序 newApp。

示例代码 9.11.1 execve()函数使用示例

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

int main(int argc, char *argv[])
{
    char *arg_arr[5];
    char *env_arr[5] = {"NAME=app", "AGE=25",
                        "SEX=man", NULL};

    if (2 > argc)
        exit(-1);

    arg_arr[0] = argv[1];
    arg_arr[1] = "Hello";
    arg_arr[2] = "World";
    arg_arr[3] = NULL;
    execve(argv[1], arg_arr, env_arr);

    perror("execve error");
    exit(-1);
}
```

将上述程序编译成一个可执行文件 testApp。

接着编写新程序，在新程序当中打印出环境变量和传参，如下所示：

示例代码 9.11.2 新程序

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

extern char **environ;

int main(int argc, char *argv[])
{
    char **ep = NULL;
    int j;
```

```
    for (j = 0; j < argc; j++)
        printf("argv[%d]: %s\n", j, argv[j]);

    puts("env:");
    for (ep = environ; *ep != NULL; ep++)
        printf("    %s\n", *ep);

    exit(0);
}
```

将新程序编译成 newApp 可执行文件，两份程序编译好之后，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
newApp  testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.1 编译程序

接下来进行测试，运行 testApp 程序，传入一个参数，该参数便是新程序 newApp 的可执行文件路径：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
newApp  testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./newApp
argv[0]: ./newApp
argv[1]: Hello
argv[2]: World
env:
   NAME=app
   AGE=25
   SEX=man
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.2 测试结果

由上图打印结果可知，在我们的 testApp 程序中，成功通过 execve()运行了另一个新的程序 newApp，当 newApp 程序运行完成退出后，testApp 进程就结束了。

示例代码 9.11.1 中 execve()函数的使用并不是它真正的应用场景，通常由 fork()生成的子进程对 execve()的调用最为频繁，也就是子进程执行 exec 操作；示例代码 9.11.1 中的 execve 用法在实际的应用不常见，这里只是给大家进行演示说明。

说到这里，我们来分析一个问题，为什么需要在子进程中执行新程序？其实这个问题非常简单，虽然可以直接在子进程分支编写子进程需要运行的代码，但是不够灵活，扩展性不够好，直接将子进程需要运行的代码单独放在一个可执行文件中不是更好吗，所以就出现了 exec 操作。

9.11.2 exec 库函数

exec 族函数包括多个不同的函数，这些函数命名都以 exec 为前缀，上一小节给大家介绍的 execve()函数也属于 exec 族函数中的一员，但它属于系统调用；本小节我们介绍 exec 族函数中的库函数，这些库函数都是基于系统调用 execve()而实现的，虽然参数各异、但功能相同，包括：execl()、execlp()、execle()、execv()、execvp()、execvpe()，它们的函数原型如下所示：

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ... /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg, ... /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

使用这些函数需要包含头文件<unistd.h>。

接下来简单地介绍下它们之间的区别：

- execl()和 execv()都是基本的 exec 函数，都可用于执行一个新程序，它们之间的区别在于参数格式不同；参数 path 意义和格式都相同，与系统调用 execve()的 filename 参数相同，指向新程序的路径名，既可以是绝对路径、也可以是相对路径。execl()和 execv()不同的在于第二个参数，execv()的 argv 参数与 execve()的 argv 参数相同，也是字符串指针数组；而 execl()把参数列表依次排列，使用可变参数形式传递，本质上也是多个字符串，以 NULL 结尾，如下所示：

```
// execv 传参
char *arg_arr[5];

arg_arr[0] = "./newApp";
arg_arr[1] = "Hello";
arg_arr[2] = "World";
arg_arr[3] = NULL;
execv("./newApp", arg_arr);

// execl 传参
execl("./newApp", "./newApp", "Hello", "World", NULL);
```

- execlp()和 execvp()在 execl()和 execv()基础上加了一个 p，这个 p 其实表示的是 PATH；execl()和 execv()要求提供新程序的路径名，而 execlp()和 execvp()则允许只提供新程序文件名，系统会在由环境变量 PATH 所指定的目录列表中寻找相应的可执行文件，如果执行的新程序是一个 Linux 命令，这将很有用；当然，execlp()和 execvp()函数也兼容相对路径和绝对路径的方式。
- execle()和 execvpe()这两个函数在命名上加了一个 e，这个 e 其实表示的是 environment 环境变量，意味着这两个函数可以指定自定义的环境变量列表给新程序，参数 envp 与系统调用 execve()的 envp 参数相同，也是字符串指针数组，使用方式如下所示：

```
// execvpe 传参
char *env_arr[5] = {"NAME=app", "AGE=25",
        "SEX=man", NULL};
char *arg_arr[5];

arg_arr[0] = "./newApp";
arg_arr[1] = "Hello";
arg_arr[2] = "World";
```

```
arg_arr[3] = NULL;
execvpe("./newApp", arg_arr, env_arr);

// execle 传参
execle("./newApp", "./newApp", "Hello", "World", NULL, env_arr);
```

给大家介绍完这些 exec 函数之后，下面将进行实战。

9.11.3 exec 族函数使用示例

使用以上给大家介绍的 6 个 exec 库函数运行 ls 命令，并加入参数-a 和-l。

1、execl()函数运行 ls 命令。

示例代码 9.11.3 execl 执行 ls 命令

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

int main(void)
{
    execl("/bin/ls", "ls", "-a", "-l", NULL);
    perror("execl error");
    exit(-1);
}
```

2、execv()函数运行 ls 命令。

示例代码 9.11.4 execv()执行 ls 命令

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

int main(void)
{
    char *arg_arr[5];

    arg_arr[0] = "ls";
    arg_arr[1] = "-a";
    arg_arr[2] = "-l";
    arg_arr[3] = NULL;
    execv("/bin/ls", arg_arr);

    perror("execv error");
    exit(-1);
}
```

3、execlp()函数运行 ls 命令。

示例代码 9.11.5 execlp()执行 ls 命令

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

int main(void)
{
    execlp("ls", "ls", "-a", "-l", NULL);
    perror("execlp error");
    exit(-1);
}
```

4、execvp()函数运行 ls 命令。

示例代码 9.11.6 execvp()执行 ls 命令

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

int main(void)
{
    char *arg_arr[5];

    arg_arr[0] = "ls";
    arg_arr[1] = "-a";
    arg_arr[2] = "-l";
    arg_arr[3] = NULL;
    execvp("ls", arg_arr);

    perror("execvp error");
    exit(-1);
}
```

5、execle()函数运行 ls 命令。

示例代码 9.11.7 execle()执行 ls 命令

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

extern char **environ;

int main(void)
{
    execle("/bin/ls", "ls", "-a", "-l", NULL, environ);
    perror("execle error");
```

```
    exit(-1);
}
```

6、execvpe()函数运行 ls 命令。

示例代码 9.11.8 execvpe()执行 ls 命令

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

extern char **environ;

int main(void)
{
    char *arg_arr[5];

    arg_arr[0] = "ls";
    arg_arr[1] = "-a";
    arg_arr[2] = "-l";
    arg_arr[3] = NULL;
    execvpe("ls", arg_arr, environ);

    perror("execvpe error");
    exit(-1);
}
```

以上所有的这些示例代码，运行结果都是一样的，与"ls -al"命令效果相同，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
总用量 24
drwxr-xr-x 2 dt dt 4096 3月  30 12:58 .
drwxrwxr-x 5 dt dt 4096 3月  29 20:24 ..
-rwxrwxr-x 1 dt dt 8784 3月  30 12:58 testApp
-rw-rw-r-- 1 dt dt  275 3月  30 12:58 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.3 测试结果

9.11.4 system()函数

使用 system()函数可以很方便地在我们的程序当中执行任意 shell 命令，本小节来学习下 system()函数的用法，以及介绍 system()函数的实现方法。

首先来看看 system()函数原型，如下所示：

```
#include <stdlib.h>

int system(const char *command);
```

这是一个库函数，使用该函数需要包含头文件<stdlib.h>。

函数参数和返回值含义如下：

command：参数 command 指向需要执行的 shell 命令，以字符串的形式提供，譬如"ls -al"、"echo HelloWorld"等。

返回值：关于 system()函数的返回值有多种不同的情况，稍后给大家介绍。

system()函数其内部的是通过调用 fork()、execl()以及 waitpid()这三个函数来实现它的功能，首先 system()会调用 fork()创建一个子进程来运行 shell（可以把这个子进程成为 shell 进程），并通过 shell 执行参数 command 所指定的命令。譬如：

```
system("ls -la")
system("echo HelloWorld")
```

system()的返回值如下：

- 当参数 command 为 NULL，如果 shell 可用则返回一个非 0 值，若不可用则返回 0；针对一些非 UNIX 系统，该系统上可能是没有 shell 的，这样就会导致 shell 不可能；如果 command 参数不为 NULL，则返回值从以下的各种情况所决定。
- 如果无法创建子进程或无法获取子进程的终止状态，那么 system()返回-1；
- 如果子进程不能执行 shell，则 system()的返回值就好像是子进程通过调用_exit(127)终止了；
- 如果所有的系统调用都成功，system()函数会返回执行 command 的 shell 进程的终止状态。

system()的主要优点在于使用上方便简单，编程时无需自己处理对 fork()、exec 函数、waitpid()以及 exit()等调用细节，system()内部会代为处理；当然这些优点通常是以牺牲效率为代价的，使用 system()运行 shell 命令需要至少创建两个进程，一个进程用于运行 shell、另外一个或多个进程则用于运行参数 command 中解析出来的命令，每一个命令都会调用一次 exec 函数来执行；所以从这里可以看出，使用 system()函数其效率会大打折扣，如果我们的程序对效率或速度有所要求，那么建议大家不是直接使用 system()。

使用示例

以下示例代码演示了 system()函数的用法，执行测试程序时，将需要执行的命令通过参数传递给 main()函数，在 main 函数中调用 system()来执行该条命令。

示例代码 9.11.9 system()函数使用示例

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

int main(int argc, char *argv[])
{
    int ret;

    if (2 > argc)
        exit(-1);

    ret = system(argv[1]);
    if (-1 == ret)
        fputs("system error.\n", stderr);
    else {
        if (WIFEXITED(ret) && (127 == WEXITSTATUS(ret)))
            fputs("could not invoke shell.\n", stderr);
    }

    exit(0);
```

```
}
```

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp pwd
/home/dt/vscode_ws/2_chapter
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 'ls -al'
总用量 24
drwxr-xr-x 2 dt dt 4096 3月  30 17:13 .
drwxrwxr-x 5 dt dt 4096 3月  29 20:24 ..
-rwxrwxr-x 1 dt dt 8752 3月  30 17:13 testApp
-rw-rw-r-- 1 dt dt  312 3月  30 17:11 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.11.4 测试结果

9.12 进程状态与进程关系

本小节来聊一聊关于进程状态与进程关系相关的话题。

9.12.1 进程状态

Linux 系统下进程通常存在 6 种不同的状态，分为：就绪态、运行态、僵尸态、可中断睡眠状态（浅度睡眠）、不可中断睡眠状态（深度睡眠）以及暂停态。

- 就绪态（Ready）：指该进程满足被 CPU 调度的所有条件但此时并没有被调度执行，只要得到 CPU 就能够直接运行；意味着该进程已经准备好被 CPU 执行，当一个进程的时间片到达，操作系统调度程序会从就绪态链表中调度一个进程；
- 运行态：指该进程当前正在被 CPU 调度运行，处于就绪态的进程得到 CPU 调度就会进入运行态；
- 僵尸态：僵尸态进程其实指的就是僵尸进程，指该进程已经结束、但其父进程还未给它“收尸”；
- 可中断睡眠状态：可中断睡眠也称为浅度睡眠，表示睡的不够“死”，还可以被唤醒，一般来说可以通过信号来唤醒；
- 不可中断睡眠状态：不可中断睡眠称为深度睡眠，深度睡眠无法被信号唤醒，只能等待相应的条件成立才能结束睡眠状态。把浅度睡眠和深度睡眠统称为等待态（或者叫阻塞态），表示进程处于一种等待状态，等待某种条件成立之后便会进入到就绪态；所以，处于等待态的进程是无法参与进程系统调度的。
- 暂停态：暂停并不是进程的终止，表示进程暂停运行，一般可通过信号将进程暂停，譬如 SIGSTOP 信号；处于暂停态的进程是可以恢复进入到就绪态的，譬如收到 SIGCONT 信号。

一个新创建的进程会处于就绪态，只要得到 CPU 就能被执行。以下列出了进程各个状态之间的转换关系，如下所示：

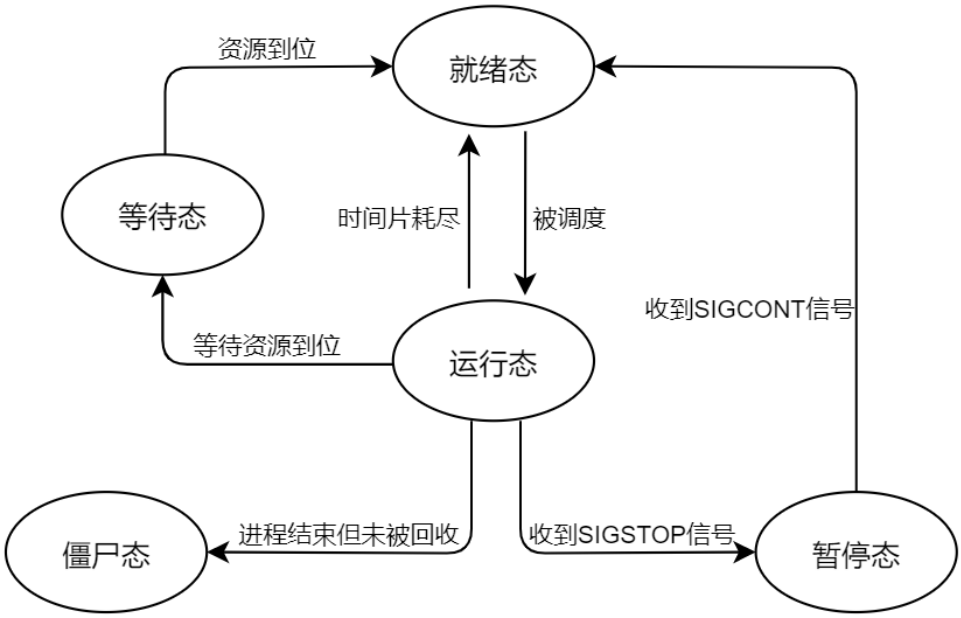


图 9.12.1 进程各状态之间的切换

9.12.2 进程关系

介绍完进程状态之后，接下来聊一聊进程关系，在 Linux 系统下，每个进程都有自己唯一的标识：进程号（进程 ID、PID），也有自己的生命周期，进程都有自己的父进程、而父进程也有父进程，这就形成了一个以 init 进程为根的进程家族树；当子进程终止时，父进程会得到通知并能取得子进程的退出状态。

除此之外，进程间还存在着其它一些层次关系，譬如进程组和会话；所以，由此可知，进程间存在着多种不同的关系，主要包括：无关系（相互独立）、父子进程关系、进程组以及会话。

1、无关系

两个进程间没有任何关系，相互独立。

2、父子进程关系

两个进程间构成父子进程关系，譬如一个进程 fork()创建出了另一个进程，那么这两个进程间就构成了父子进程关系，调用 fork()的进程称为父进程、而被 fork()创建出来的进程称为子进程；当然，如果“生父”先与子进程结束，那么 init 进程（“养父”）就会成为子进程的父进程，它们之间同样也是父子进程关系。

3、进程组

每个进程除了有一个进程 ID、父进程 ID 之外，还有一个进程组 ID，用于标识该进程属于哪一个进程组，进程组是一个或多个进程的集合，这些进程并不是孤立的，它们彼此之间或者存在父子、兄弟关系，或者在功能上有联系。

Linux 系统设计进程组实质上是为了方便对进程进行管理。假设为了完成一个任务，需要并发运行 100 个进程，但当处于某种场景时需要终止这 100 个进程，若没有进程组就需要一个一个去终止，这样非常麻烦且容易出现一些问题；有了进程组的概念之后，就可以将这 100 个进程设置为一个进程组，这些进程共享一个进程组 ID，这样一来，终止这 100 个进程只需要终止该进程组即可。

关于进程组需要注意以下以下内容：

- 每个进程必定属于某一个进程组、且只能属于一个进程组；
- 每一个进程组有一个组长进程，组长进程的 ID 就等于进程组 ID；
- 在组长进程的 ID 前面加上一个负号即是操作进程组；

- 组长进程不能再创建新的进程组；
- 只要进程组中还存在一个进程，则该进程组就存在，这与其组长进程是否终止无关；
- 一个进程组可以包含一个或多个进程，进程组的生命周期从被创建开始，到其内所有进程终止或离开该进程组；
- 默认情况下，新创建的进程会继承父进程的进程组 ID。

通过系统调用 getpgrp()或 getpgid()可以获取进程对应的进程组 ID，其函数原型如下所示：

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
pid_t getpgrp(void);
```

首先使用该函数需要包含头文件<unistd.h>。

这两个函数都用于获取进程组 ID，getpgrp()没有参数，返回值总是调用者进程对应的进程组 ID；而对于 getpgid()函数来说，可通过参数 pid 指定获取对应进程的进程组 ID，如果参数 pid 为 0 表示获取调用者进程的进程组 ID。

getpgid()函数成功将返回进程组 ID；失败将返回-1、并设置 errno。

所以由此可知，getpgrp()就等价于 getpgid(0)。

使用示例

示例代码 9.12.1 获取进程组 ID

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

int main(void)
{
    pid_t pid = getpid();

    printf("进程组 ID<%d>---getpgrp()\n", getpgrp());
    printf("进程组 ID<%d>---getpgid(0)\n", getpgid(0));
    printf("进程组 ID<%d>---getpgid(%d)\n", getpgid(pid), pid);
    exit(0);
}
```

测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
进程组ID<14779>---getpgrp()
进程组ID<14779>---getpgid(0)
进程组ID<14779>---getpgid(14779)
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.12.2 测试结果

从上面的结果可以发现，其新创建的进程对应的进程组 ID 等于该进程的 ID。

调用系统调用 setpgid()或 setpgrp()可以加入一个现有的进程组或创建一个新的进程组，其函数原型如下所示：

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
int setpgrp(void);
```

使用这些函数同样需要包含头文件<unistd.h>。

setpgid()函数将参数 pid 指定的进程的进程组 ID 设置为参数 gpid。如果这两个参数相等（pid==gpid），则由 pid 指定的进程变成为进程组的组长进程，创建了一个新的进程；如果参数 pid 等于 0，则使用调用者的进程 ID；另外，如果参数 gpid 等于 0，则创建一个新的进程组，由参数 pid 指定的进程作为进程组组长进程。

setpgrp()函数等价于 setpgid(0, 0)。

一个进程只能为它自己或它的子进程设置进程组 ID，在它的子进程调用 exec 函数后，它就不能更改该子进程的进程组 ID 了。

使用示例

示例代码 9.12.2 创建进程组或加入一个现有进程组

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

int main(void)
{
    printf("更改前进程组 ID<%d>\n", getpgrp());
    setpgrp();
    printf("更改后进程组 ID<%d>\n", getpgrp());
    exit(0);
}
```

4、会话

介绍完进程组之后，再来看下会话，会话是一个或多个进程组的集合，其与进程组、进程之间的关系如下图所示：

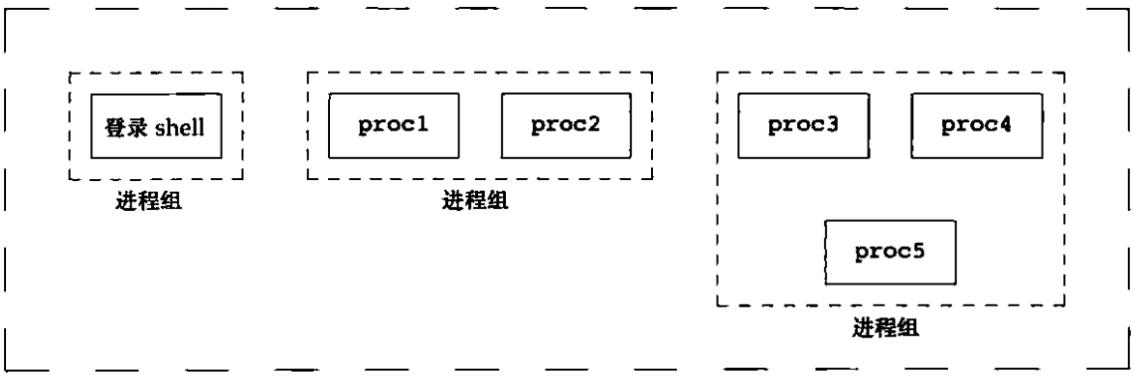


图 9.12.3 会话

一个会话可包含一个或多个进程组，但只能有一个前台进程组，其它的是后台进程组；每个会话都有一个会话首领（leader），即创建会话的进程。一个会话可以有控制终端、也可没有控制终端，在有控制终端

的情况下也只能连接一个控制终端，这通常是登录到其上的终端设备（在终端登录情况下）或伪终端设备（譬如通过 SSH 协议网络登录），一个会话中的进程组可被分为一个前台进程组以及一个或多个后台进程组。

会话的首领进程连接一个终端之后，该终端就成为会话的控制终端，与控制终端建立连接的会话首领进程被称为控制进程；产生在终端上的输入和信号将发送给会话的前台进程组中的所有进程，譬如 Ctrl + C（产生 SIGINT 信号）、Ctrl + Z（产生 SIGTSTP 信号）、Ctrl + \（产生 SIGQUIT 信号）等等这些由控制终端产生的信号。

当用户在某个终端登录时，一个新的会话就开始了；当我们在 Linux 系统下打开了多个终端窗口时，实际上就是创建了多个终端会话。

一个进程组由组长进程的 ID 标识，而对于会话来说，会话的首领进程的进程组 ID 将作为该会话的标识，也就是会话 ID（sid），在默认情况下，新创建的进程会继承父进程的会话 ID。通过系统调用 getsid()可以获取进程的会话 ID，其函数原型如下所示：

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

使用该函数需要包含头文件<unistd.h>，如果参数 pid 为 0，则返回调用者进程的会话 ID；如果参数 pid 不为 0，则返回参数 pid 指定的进程对应的会话 ID。成功情况下，该函数返回会话 ID，失败则返回-1、并设置 errno。

使用示例

示例代码 9.12.3 获取进程的会话 ID

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

int main(void)
{
    printf("会话 ID<%d>\n", getsid(0));
    exit(0);
}
```

打印结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
会话ID<3304>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.12.4 测试结果

使用系统调用 setsid()可以创建一个会话，其函数原型如下所示：

```
#include <unistd.h>

pid_t setsid(void);
```

如果调用者进程不是进程组的组长进程，调用 setsid()将创建一个新的会话，调用者进程是新会话的首领进程，同样也是一个新的进程组的组长进程，调用 setsid()创建的会话将没有控制终端。

setsid()调用成功将返回新会话的会话 ID；失败将返回-1，并设置 errno。

9.13 守护进程

本小节学习守护进程，将对守护进程的概念以及如何编写一个守护进程程序进行介绍。

9.13.1 何为守护进程

守护进程（Daemon）也称为精灵进程，是运行在后台的一种特殊进程，它独立于控制终端并且周期性地执行某种任务或等待处理某些事情的发生，主要表现为以下两个特点：

- **长期运行。**守护进程是一种生存期很长的一种进程，它们一般在系统启动时开始运行，除非强行终止，否则直到系统关机都会保持运行。与守护进程相比，普通进程都是在用户登录或运行程序时创建，在运行结束或用户注销时终止，但守护进程不受用户登录注销的影响，它们将会一直运行着、直到系统关机。
- **与控制终端脱离。**在 Linux 中，系统与用户交互的界面称为终端，每一个从终端开始运行的进程都会依附于这个终端，这是上一小节给大家介绍的控制终端，也就是会话的控制终端。当控制终端被关闭的时候，该会话就会退出，由控制终端运行的所有进程都会被终止，这使得普通进程都是和运行该进程的终端相绑定的；但守护进程能突破这种限制，它脱离终端并且在后台运行，脱离终端的目的是为了避免进程在运行的过程中的信息在终端显示并且进程也不会被任何终端所产生的信息所打断。

守护进程是一种很有用的进程。Linux 中大多数服务器就是用守护进程实现的，譬如，Internet 服务器 inetd、Web 服务器 httpd 等。同时，守护进程完成许多系统任务，譬如作业规划进程 crond 等。

守护进程 Daemon，通常简称为 d，一般进程名后面带有 d 就表示它是一个守护进程。守护进程与终端无任何关联，用户的登录与注销与守护进程无关、不受其影响，守护进程自成进程组、自成会话，即 pid=gid=sid。通过命令"ps -ajx"查看系统所有的进程，如下所示：

```
dt@dt-virtual-machine:~$ ps -ajx
  PPID    PID   PGID    SID TTY      TPGID STAT   UID   TIME COMMAND
     0      1      1      1 ?           -1 Ss       0   0:35 /sbin/init splash
     0      2      0      0 ?           -1 S        0   0:01 [kthreadd]
     2      4      0      0 ?           -1 I<       0   0:00 [kworker/0:0H]
     2      6      0      0 ?           -1 I<       0   0:00 [mm_percpu_wq]
     2      7      0      0 ?           -1 S        0   0:00 [ksoftirqd/0]
     2      8      0      0 ?           -1 I        0  14:05 [rcu_sched]
     2      9      0      0 ?           -1 I        0   0:00 [rcu_bh]
     2     10      0      0 ?           -1 S        0   0:00 [migration/0]
     2     11      0      0 ?           -1 S        0   0:06 [watchdog/0]
     2     12      0      0 ?           -1 S        0   0:00 [cpuhp/0]
     2     13      0      0 ?           -1 S        0   0:00 [cpuhp/1]
     2     14      0      0 ?           -1 S        0   0:05 [watchdog/1]
     2     15      0      0 ?           -1 S        0   0:00 [migration/1]
     2     16      0      0 ?           -1 S        0   0:00 [ksoftirqd/1]
     2     18      0      0 ?           -1 I<       0   0:00 [kworker/1:0H]
     2     19      0      0 ?           -1 S        0   0:00 [cpuhp/2]
     2     20      0      0 ?           -1 S        0   0:05 [watchdog/2]
     2     21      0      0 ?           -1 S        0   0:00 [migration/2]
     2     22      0      0 ?           -1 S        0   0:02 [ksoftirqd/2]
     2     24      0      0 ?           -1 I<       0   0:00 [kworker/2:0H]
     2     25      0      0 ?           -1 S        0   0:00 [cpuhp/3]
     2     26      0      0 ?           -1 S        0   0:05 [watchdog/3]
     2     27      0      0 ?           -1 S        0   0:00 [migration/3]
     2     28      0      0 ?           -1 S        0   0:00 [ksoftirqd/3]
     2     30      0      0 ?           -1 I<       0   0:00 [kworker/3:0H]
     2     31      0      0 ?           -1 S        0   0:00 [cpuhp/4]
     2     32      0      0 ?           -1 S        0   0:05 [watchdog/4]
     2     33      0      0 ?           -1 S        0   0:00 [migration/4]
     2     34      0      0 ?           -1 S        0   0:10 [ksoftirqd/4]
     2     36      0      0 ?           -1 I<       0   0:00 [kworker/4:0H]
```

图 9.13.1 查看系统中的所有进程

TTY 一栏是问号？表示该进程没有控制终端，也就是守护进程，其中 COMMAND 一栏使用中括号[]括起来的表示内核线程，这些线程是在内核里创建，没有用户空间代码，因此没有程序文件名和命令行，通常采用 k 开头的名字，表示 Kernel。

9.13.2 编写守护进程程序

如何将自己编写的程序运行之后变成一个守护进程呢？本小节就来学习如何编写守护进程程序，编写守护进程一般包含如下几个步骤：

1) 创建子进程、终止父进程

父进程调用 fork()创建子进程，然后父进程使用 exit()退出，这样做实现了下面几点。第一，如果该守护进程是作为一条简单地 shell 命令启动，那么父进程终止会让 shell 认为这条命令已经执行完毕。第二，虽然子进程继承了父进程的进程组 ID，但它有自己独立的进程 ID，这保证了子进程不是一个进程组的组长进程，这是下面将要调用 setsid 函数的先决条件！

2) 子进程调用 setsid 创建会话

这步是关键，在子进程中调用上一小节给大家介绍的 setsid()函数创建新的会话，由于之前子进程并不是进程组的组长进程，所以调用 setsid()会使得子进程创建一个新的会话，子进程成为新会话的首领进程，同样也创建了新的进程组、子进程成为组长进程，此时创建的会话将没有控制终端。所以这里调用 setsid 有三个作用：让子进程摆脱原会话的控制、让子进程摆脱原进程组的控制和让子进程摆脱原控制终端的控制。

在调用 fork 函数时，子进程继承了父进程的会话、进程组、控制终端等，虽然父进程退出了，但原先的会话期、进程组、控制终端等并没有改变，因此，那还不是真正意义上使两者独立开来。setsid 函数能够使子进程完全独立出来，从而脱离所有其他进程的控制。

3) 将工作目录更改为根目录

子进程是继承了父进程的当前工作目录，由于在进程运行中，当前目录所在的文件系统是不能卸载的，这对以后使用会造成很多的麻烦。因此通常的做法是让“/”作为守护进程的当前目录，当然也可以指定其它目录来作为守护进程的工作目录。

4) 重设文件权限掩码 umask

文件权限掩码 umask 用于对新建文件的权限位进行屏蔽，在 5.5.5 小节中有介绍。由于使用 fork 函数新建的子进程继承了父进程的文件权限掩码，这就给子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为 0，确保子进程有最大操作权限、这样可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是 umask，通常的使用方法为 umask(0)。

5) 关闭不再需要的文件描述符

子进程继承了父进程的所有文件描述符，这些被打开的文件可能永远不会被守护进程（此时守护进程指的就是子进程，父进程退出、子进程成为守护进程）读或写，但它们一样消耗系统资源，可能导致所在的文件系统无法卸载，所以必须关闭这些文件，这使得守护进程不再持有从其父进程继承过来的任何文件描述符。

6) 将文件描述符号为 0、1、2 定位到/dev/null

将守护进程的标准输入、标准输出以及标准错误重定向到/dev/null，这使得守护进程的输出无处显示、也无处从交互式用户那里接收输入。

7) 其它：忽略 SIGCHLD 信号

处理 SIGCHLD 信号不是必须的，但对于某些进程，特别是并发服务器进程往往是特别重要的，服务器进程在接收到客户端请求时会创建子进程去处理该请求，如果子进程结束之后，父进程没有去 wait 回收子进程，则子进程将成为僵尸进程；如果父进程 wait 等待子进程退出，将又会增加父进程的负担、也就是增加服务器的负担，影响服务器进程的并发性能，在 Linux 下，可以将 SIGCHLD 信号的处理方式设置为 SIG_IGN，也就是忽略该信号，可让内核将僵尸进程转交给 init 进程去处理，这样既不会产生僵尸进程、又省去了服务器进程回收子进程所占用的时间。

守护进程一般以单例模式运行，关于单例模式运行请看 9.13.3 小节内容。

接下来，我们根据上面的介绍的步骤，来编写一个守护进程程序，示例代码如下所示：

示例代码 9.13.1 守护进程示例代码

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

int main(void)
{
    pid_t pid;
    int i;

    /* 创建子进程 */
    pid = fork();
    if (0 > pid) {
        perror("fork error");
        exit(-1);
    }
    else if (0 < pid)//父进程
```

```
        exit(0);        //直接退出

    /*
     *子进程
     */

    /* 1.创建新的会话、脱离控制终端 */
    if (0 > setsid()) {
        perror("setsid error");
        exit(-1);
    }

    /* 2.设置当前工作目录为根目录 */
    if (0 > chdir("/")) {
        perror("chdir error");
        exit(-1);
    }

    /* 3.重设文件权限掩码 umask */
    umask(0);

    /* 4.关闭所有文件描述符 */
    for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
        close(i);

    /* 5.将文件描述符号为 0、1、2 定位到/dev/null */
    open("/dev/null", O_RDWR);
    dup(0);
    dup(0);

    /* 6.忽略 SIGCHLD 信号 */
    signal(SIGCHLD, SIG_IGN);

    /* 正式进入到守护进程 */
    for ( ; ; ) {
        sleep(1);
        puts("守护进程运行中.....");
    }

    exit(0);
}
```

整个代码的编写都是根据上面的介绍来完成的，这里就不再啰嗦，示例代码中使用到的函数在前面都已经学习过，其中第 4 步中调用 sysconf(_SC_OPEN_MAX)用于获取当前系统允许进程打开的最大文件数量。

我们在守护进程中添加了死循环，每隔 1 秒钟打印一行字符串信息，接下来编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.13.2 测试结果

运行之后，没有任何打印信息输出，原因在于守护进程已经脱离了控制终端，它的打印信息并不会输出显示到终端，在代码中已经将标准输入、输出以及错误重定位到了/dev/null，/dev/null 是一个黑洞文件，自然是看不到输出信息。

使用"ps -ajx"命令查看进程，如下所示：

```
    2  20555      0      0 ?           -1 I        0   0:00 [kworker/2:2]
    2  20568      0      0 ?           -1 I        0   0:00 [kworker/u256:2]
 1911  20634  20634  20634 ?           -1 Ss    1000   0:00 ./testApp
 3304  20662  20662   3304 pts/19    20662 R+    1000   0:00 ps -ajx
 3372  21681   2131   2131 ?           -1 Sl    1000   1:09 /home/dt/.vscode/extensions/ms-vscode
 3315  22627   2131   2131 ?           -1 Sl    1000   1:16 /usr/share/code/code /usr/share/code/
    2  37425      0      0 ?           -1 I<       0   0:00 [xfsalloc]
    2  37426      0      0 ?           -1 I<       0   0:00 [xfs_mru_cache]
    2  37431      0      0 ?           -1 S        0   0:00 [jfsIO]
    2  37432      0      0 ?           -1 S        0   0:00 [jfsCommit]
    2  37433      0      0 ?           -1 S        0   0:00 [jfsCommit]
    2  37434      0      0 ?           -1 S        0   0:00 [jfsCommit]
    2  37435      0      0 ?           -1 S        0   0:00 [jfsCommit]
    2  37436      0      0 ?           -1 S        0   0:00 [jfsCommit]
    2  37437      0      0 ?           -1 S        0   0:00 [jfsCommit]
    2  37438      0      0 ?           -1 S        0   0:00 [jfsSync]
 1911  79017   1996   1996 ?           -1 Sl    1000   0:00 /usr/lib/gvfs/gvfsd-http --spawner :1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.13.3 查看守护进程

从上图可知，testApp 进程成为了一个守护进程，与控制台脱离，当关闭当前控制终端时，testApp 进程并不会受到影响，依然会正常继续运行；而对于普通进程来说，终端关闭，那么由该终端运行的所有进程都会被强制关闭，因为它们处于同一个会话。关于这个问题，大家可以自己去测试下，对比测试普通进程与守护进程，当终端关闭之后是否还在继续运行。

守护进程可以通过终端命令行启动，但通常它们是由系统初始化脚本进行启动，譬如/etc/rc*或/etc/init.d/*等。

9.13.3 SIGHUP 信号

当用户准备退出会话时，系统向该会话发出 SIGHUP 信号，会话将 SIGHUP 信号发送给所有子进程，子进程接收到 SIGHUP 信号后，便会自动终止，当所有会话中的所有进程都退出时，会话也就终止了；因为程序当中一般不会对 SIGHUP 信号进行处理，所以对应的处理方式为系统默认方式，SIGHUP 信号的系统默认处理方式便是终止进程。

上面解释了，为什么子进程会随着会话的退出而退出，因为它收到了 SIGHUP 信号。不管是前台进程还是后台进程都会收到该信号。如果忽略该信号，将会出现什么样的结果？接下来我们编写一个简单地测试程序，示例代码如下所示：

示例代码 9.13.2 忽略 SIGHUP 示例代码

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

```
int main(void)
{
    signal(SIGHUP, SIG_IGN);

    for ( ; ; ) {
        sleep(1);
        puts("进程运行中.....");
    }
}
```

代码很简单，调用 signal()函数将 SIGHUP 信号的处理方式设置为忽略。测试运行

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
进程运行中.....
进程运行中.....
进程运行中.....
进程运行中.....
```

图 9.13.4 测试结果

成功运行之后，关闭终端

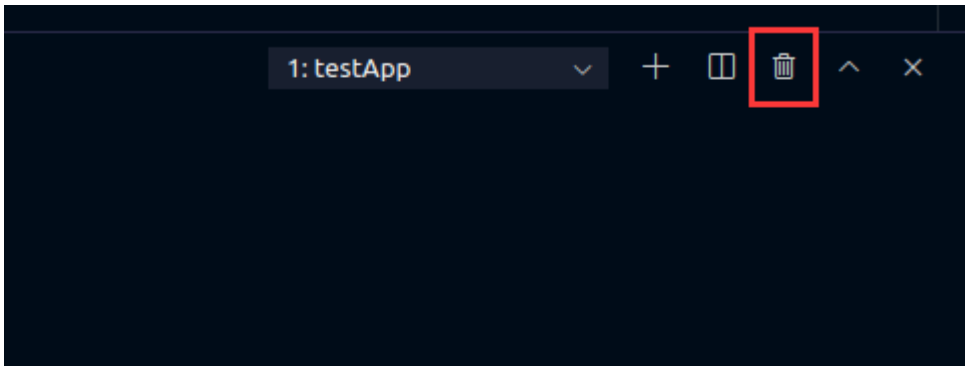


图 9.13.5 关闭终端

再重新打开终端，使用 ps 命令查看 testApp 进程是否存在，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws$
dt@dt-virtual-machine:~/vscode_ws$ ps -ajx | grep testApp | grep -v grep
 21992  23648  23648  23601 ?           -1 S     1000   0:00 ./testApp
dt@dt-virtual-machine:~/vscode_ws$
```

图 9.13.6 查看 testApp 进程

可以发现 testApp 进程依然还在运行，但此时它已经变成了守护进程，脱离了控制终端，所以由此可知，当程序当中忽略 SIGHUP 信号之后，进程不会随着终端退出而退出，事实上，控制终端只是会话中的一个进程，只有会话中的所有进程退出后，会话才会结束；很显然当程序中忽略了 SIGHUP 信号，导致该进程不会终止，所以会话也依然会存在，从上图可知，其会话 ID 等于 23601，但此时会话已经没有控制终端了。

9.14 单例模式运行

通常情况下，一个程序可以被多次执行，即程序在还没有结束的情况下，又再次执行该程序，也就是系统中同时存在多个该程序的实例化对象（进程），譬如大家所熟悉的聊天软件 QQ，我们可以在电脑上同时登陆多个 QQ 账号，譬如还有一些游戏也是如此，在一台电脑上同时登陆多个游戏账号，只要你电脑不卡机、随便你开几个号。

但对于有些程序设计来说，不允许出现这种情况，程序只能被执行一次，只要该程序没有结束，就无法再次运行，我们把这种情况称为单例模式运行。譬如系统中守护进程，这些守护进程一般都是服务器进程，服务器程序只需要运行一次即可，能够在系统整个的运行过程中提供相应的服务支持，多次同时运行并没有意义、甚至还会带来错误！

如果希望我们的程序具有单例模式运行的功能，应该如何去实现呢？

9.14.1 通过文件存在与否进行判断

首先这是一个非常简单且容易想到的方法：用一个文件的存在与否来做标志，在程序运行正式代码之前，先判断一个特定的文件是否存在，如果存在则表明进程已经运行，此时应该立马退出；如果不存在则表明进程没有运行，然后创建该文件，当程序结束时再删除该文件即可！

这种方法是大家比较容易想到的，通过一个特定文件的存在与否来做判断，当然这个特定的文件的命名要弄的特殊一点，避免在文件系统中不会真的存在该文件，接下来我们编写一个程序进行测试。

示例代码 9.14.1 简单方式实现单例模式运行

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

#define LOCK_FILE    "./testApp.lock"

static void delete_file(void)
{
    remove(LOCK_FILE);
}

int main(void)
{
    /* 打开文件 */
    int fd = open(LOCK_FILE, O_RDONLY | O_CREAT | O_EXCL, 0666);
    if (-1 == fd) {
        fputs("不能重复执行该程序!\n", stderr);
        exit(-1);
    }

    /* 注册进程终止处理函数 */
```

```
    if (atexit(delete_file))
        exit(-1);

    puts("程序运行中...");
    sleep(10);
    puts("程序结束");

    close(fd);              //关闭文件
    exit(0);
}
```

在上述示例代码中，通过当前目录下的 testApp.lock 文件作为特定文件进行判断该文件是否存在，当然这里只是举个例子，如果在实际应用编程中使用了这种方法，这个特定文件需要存放在一个特定的路径下。

代码中以 O_RDONLY|O_CREAT|O_EXCL 的方式打开文件，如果文件不存在则创建文件，如果文件存在则 open 会报错返回-1；使用 atexit 注册进程终止处理函数，当程序退出时，使用 remove()删除该文件。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[1] 460
程序运行中...
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序！
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序！
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序！
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序！
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ 程序结束

[1]+  已完成               ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
程序运行中...
程序结束
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.14.1 测试结果

在上面测试中，首先第一次以后台方式运行了 testApp 程序，之后再运行 testApp 程序，由于文件已经存在，所以 open()调用会失败，所以意味着进程正在运行中，所以会打印相应的字符串然后退出。直到第一次运行的程序结束时，才能执行 testApp 程序，这样就实现了一个简单地具有单例模式运行功能的程序。

虽然上面实现了一个简单地单例模式运行的程序，但是仔细一想其实有很大的问题，主要包括如下三个方面：

- 程序中使用_exit()退出，那么将无法执行 delete_file()函数，意味着无法删除这个特定的文件；
- 程序异常退出。程序异常同样无法执行到进程终止处理函数 delete_file()，同样将导致无法删除这个特定的文件；
- 计算机掉电关机。这种情况就更加直接了，计算机可能在程序运行到任意位置时发生掉电关机的情况，这是无法预料的；如果文件没有删除就发生了这种情况，计算机重启之后文件依然存在，导致程序无法执行。

针对第一种情况，我们使用 exit()代替_exit()可以很好的解决这种问题；但是对于第二种情况来说，异常退出，譬如进程接收到信号导致异常终止，有一种解决办法便是设置信号处理方式为忽略信号，这样当进程接收到信号时就会被忽略，或者是针对某些信号注册信号处理函数，譬如 SIGTERM、SIGINT 等，在信号处理函数中删除文件然后再退出进程；但依然有个问题，并不是所有信号都可被忽略或捕获的，譬如 SIGKILL 和 SIGSTOP，这两个信号是无法被忽略和捕获的，故而这种也不靠谱。

针对第三种情况的解决办法便是，使得该特定文件会随着系统的重启而销毁，这个怎么做呢？其实这个非常简单，将文件放置到系统/tmp 目录下，/tmp 是一个临时文件系统，当系统重启之后/tmp 目录下的文件就会被销毁，所以该目录下的文件的生命周期便是系统运行周期。

由此可知，虽然针对第一种情况和第三种情况都有相应的解决办法，但对于第二种情况来说，其解决办法并不靠谱，所以使用这种方法实现单例模式运行并不靠谱。

9.14.2 使用文件锁

介绍完上面第一种比较容易想到的方法外，接下来介绍一种靠谱的方法，使用文件锁来实现，事实上这种方式才是实现单例模式运行靠谱的方法。

同样也需要通过一个特定的文件来实现，当程序启动之后，首先打开该文件，调用 open 时一般使用 O_WRONLY | O_CREAT 标志，当文件不存在则创建该文件，然后尝试去获取文件锁，若是成功，则将程序的进程号（PID）写入到该文件中，写入后不要关闭文件或解锁（释放文件锁），保证进程一直持有该文件锁；若是程序获取锁失败，代表程序已经被运行、则退出本次启动。

Tips：*当程序退出或文件关闭之后，文件锁会自动解锁！*

文件锁属于本书高级 I/O 章节内容，在 13.6 小节对此做了详细介绍，这里就不再说明，通过系统调用 flock()、fcntl()或库函数 lockf()均可实现对文件进行上锁，本小节我们以系统调用 flock()为例，系统调用 flock()产生的是咨询锁（建议性锁）、并不能产生强制性锁。

接下来编写一个示例代码，使用 flock()函数对文件上锁，实现程序以单例模式运行，如下所示：

示例代码 9.14.2 使用文件锁实现单利模式运行

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>

#define LOCK_FILE    "./testApp.pid"

int main(void)
{
    char str[20] = {0};
    int fd;

    /* 打开 lock 文件，如果文件不存在则创建 */
    fd = open(LOCK_FILE, O_WRONLY | O_CREAT, 0666);
    if (-1 == fd) {
        perror("open error");
```

```
        exit(-1);
    }

    /* 以非阻塞方式获取文件锁 */
    if (-1 == flock(fd, LOCK_EX | LOCK_NB)) {
        fputs("不能重复执行该程序!\n", stderr);
        close(fd);
        exit(-1);
    }

    puts("程序运行中...");

    ftruncate(fd, 0);     //将文件长度截断为 0
    sprintf(str, "%d\n", getpid());
    write(fd, str, strlen(str));//写入 pid

    for ( ; ; )
        sleep(1);

    exit(0);
}
```

程序启动首先打开一个特定的文件，这里只是举例，以当前目录下的 testApp.pid 文件作为特定文件，以 O_WRONLY | O_CREAT 方式打开，如果文件不存在则创建该文件；打开文件之后使用 flock 尝试获取文件锁，调用 flock()时指定了互斥锁标志 LOCK_NB，意味着同时只能有一个进程拥有该锁，如果获取锁失败，表示该程序已经启动了，无需再次执行，然后退出；如果获取锁成功，将进程的 PID 写入到该文件中，当程序退出时，会自动解锁、关闭文件。

运行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp &
[1] 13676
程序运行中...
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
不能重复执行该程序!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 9.14.2 测试结果

这种机制在一些程序尤其是服务器程序中很常见，服务器程序使用这种方法来保证程序的单例模式运行；在 Linux 系统中/var/run/目录下有很多以.pid 为后缀结尾的文件，这个实际上是为了保证程序以单例模式运行而设计的，作为程序实现单例模式运行所需的特定文件，如下所示：

```
dt@dt-virtual-machine:/var/run$ pwd
/var/run
dt@dt-virtual-machine:/var/run$
dt@dt-virtual-machine:/var/run$ ls
acpid.pid                lightdm                snapd.socket
acpid.socket             lightdm.pid            sshd
agetty.reload            lock                   sshd.pid
alsa                     log                    sudo
apport.lock              mlocate.daily.lock     systemd
avahi-daemon             motd.dynamic           thermald
blkid                    mount                  tmpfiles.d
crond.pid                network                udev
crond.reboot             NetworkManager         udisks2
cups                     plymouth               unattended-upgrades.lock
dbus                     pppconfig              unattended-upgrades.progress
dhclient-ens33.pid       reboot-required        user
dnsmasq                  reboot-required.pkgs   utmp
do-not-hibernate         resolvconf             uuidd
firefox-restart-required rsyslogd.pid           vmblock-fuse
initctl                  sendsigs.omit.d        vmtoolsd.pid
initramfs                shm                    vmware
irqbalance.pid           snapd-snap.socket
dt@dt-virtual-machine:/var/run$
```

图 9.14.3 /var/run 目录下的文件

这些以.pid 为后缀的文件，命名方式通常是程序名+.pid，譬如 acpid.pid 对应的程序便是 acpid、lightdm.pid 对应的程序便是 lightdm 等等。如果我们要去实现一个以单例模式运行的程序，譬如一个守护进程，那么也应该将这个特定文件放置于 Linux 系统/var/run/目录下，并且文件的命名方式为 name.pid（name 表示进程名）。

关于实现单例模式运行相关内容就给大家介绍这么多，最常用的还是使用文件锁，第一种方法通过文件存在否与进行判断事实上并不靠谱；除此之外，还有其它一些方法也可用于实现单例模式运行，譬如在程序启动时通过 ps 判断进程是否存在等，关于更多的方法，欢迎大家留言！

第十章 进程间通信简介

前面学习了进程相关的内容，介绍了如何通过 fork()或 vfork()创建子进程，以及在子进程中通过 exec()函数执行一个新的程序。本章向大家介绍一个新的内容---进程间通信。

所谓进程间通信指的是系统中两个进程之间的通信，不同的进程都在各自的地址空间中、相互独立、隔离，所以它们是处在于不同的地址空间中，因此相互通信比较难，Linux 内核提供了多种进程间通信的机制。本章就来聊一聊这些进程间通信的手段，让大家对此有一个基本的认识！

10.1 进程间通信简介

进程间通信（interprocess communication，简称 IPC）指两个进程之间的通信。系统中的每一个进程都有各自的地址空间，并且相互独立、隔离，每个进程都处于自己的地址空间中。所以同一个进程的不同模块（譬如不同的函数）之间进行通信都是很简单的，譬如使用全局变量等。

但是，两个不同的进程之间要进行通信通常是比较难的，因为这两个进程处于不同的地址空间中；通常情况下，大部分的程序是不要考虑进程间通信的，因为大家所接触绝大部分程序都是单进程程序（可以有多个线程），对于一些复杂、大型的应用程序，则会根据实际需要将其设计成多进程程序，譬如 GUI、服务区应用程序等。

在一些中小型应用程序中通常不会将应用程序设计成多进程程序，自然而然便不需要考虑进程间通信的问题，所以，本章内容以了解为主、了解进程间通信以及内核提供的进程间通信机制，并不详解介绍进程间通信，如果大家在今后的工作当中参与开发的应用程序是一个多进程程序、需要考虑进程间通信的问题，此时再去深入学习这方面的知识！

10.2 进程间通信的机制有哪些？

Linux 内核提供了多种 IPC 机制，基本是从 UNIX 系统继承而来，而对 UNIX 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间通信方面的侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行了系统的改进和扩充，形成了“System V IPC”，通信进程局限在单个计算机内；后者则跳过了该限制，形成了基于套接字（Socket，也就是网络）的进程间通信机制。Linux 则把两者继承了下来，如下如所示：

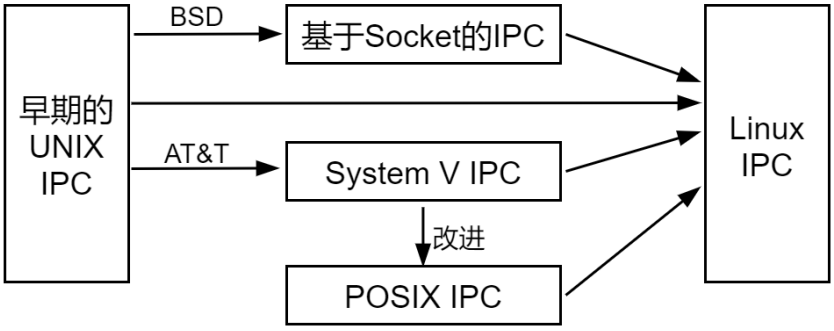


图 10.2.1 Linux 锁继承的进程间通信手段

其中，早期的 UNIX IPC 包括：管道、FIFO、信号；System V IPC 包括：System V 信号量、System V 消息队列、System V 共享内存；上图中还出现了 POSIX IPC，事实上，较早的 System V IPC 存在着一些不足之处，而 POSIX IPC 则是在 System V IPC 的基础上进行改进所形成的，弥补了 System V IPC 的一些不足之处。POSIX IPC 包括：POSIX 信号量、POSIX 消息队列、POSIX 共享内存。

总结如下：

- UNIX IPC：管道、FIFO、信号；
- System V IPC：信号量、消息队列、共享内存；
- POSIX IPC：信号量、消息队列、共享内存；
- Socket IPC：基于 Socket 进程间通信。

10.3 管道和 FIFO

管道是 UNIX 系统上最古老的 IPC 方法，它在 20 世纪 70 年代早期 UNIX 的第三个版本上就出现了。把一个进程连接到另一个进程的数据流称为管道，管道被抽象成一个文件，5.1 小节曾提及过管道文件（pipe）这样一种文件类型。

管道包括三种：

- 普通管道 pipe：通常有两种限制，一是单工，数据只能单向传输；二是只能在父子或者兄弟进程间使用；
- 流管道 s_pipe：去除了普通管道的第一种限制，为半双工，可以双向传输；只能在父子或兄弟进程间使用；
- 有名管道 name_pipe（FIFO）：去除了普通管道的第二种限制，并且允许在不相关（不是父子或兄弟关系）的进程间进行通讯。

普通管道可用于具有亲缘关系的进程间通信，并且数据只能单向传输，如果要实现双向传输，则必须要使用两个管道；而流管道去除了普通管道的第一种限制，可以半双工的方式实现双向传输，但也只能在具有亲缘关系的进程间通信；而有名管道（FIFO）则同时突破了普通管道的两种限制，即可实现双向传输、又能在非亲缘关系的进程间通信。

10.4 信号

关于信号相关的内容在本书第八章中给大家介绍过，用于通知接收信号的进程有某种事件发生，所以可用于进程间通信；除了用于进程间通信之外，进程还可以发送信号给进程本身。

10.5 消息队列

消息队列是消息的链表，存放在内核中并由消息队列标识符标识，消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺陷。消息队列包括 POSIX 消息队列和 System V 消息队列。

消息队列是 UNIX 下不同进程之间实现共享资源的一种机制，UNIX 允许不同进程将格式化的数据流以消息队列形式发送给任意进程，有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。

10.6 信号量

信号量是一个计数器，与其它进程间通信方式不大相同，它主要用于控制多个进程间或一个进程内的多个线程间对共享资源的访问，相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，同时，进程也可以修改该标志，除了用于共享资源的访问控制外，还可用于进程同步。

它常作为一种锁机制，防止某进程在访问资源时其它进程也访问该资源，因此，主要作为进程间以及同一个进程内不同线程之间的同步手段。Linux 提供了一组精心设计的信号量接口来对信号量进行操作，它们声明在头文件 sys/sem.h 中。

10.7 共享内存

共享内存就是映射一段能被其它进程所访问的内存，这段共享内存由一个进程创建，但其它的多个进程都可以访问，使得多个进程可以访问同一块内存空间。共享内存是最快的 IPC 方式，它是针对其它进程间通信方式运行效率低而专门设计的，它往往与其它通信机制，譬如结合信号量来使用，以实现进程间的同步和通信。

10.8 套接字（Socket）

Socket 是一种 IPC 方法，是基于网络的 IPC 方法，允许位于同一主机（计算机）或使用网络连接起来的不同主机上的应用程序之间交换数据，说白了就是网络通信。

在一个典型的客户端/服务器场景中，应用程序使用 socket 进行通信的方式如下：

- 各个应用程序创建一个 socket。socket 是一个允许通信的“设备”，两个应用程序都需要用到它。
- 服务器将自己的 socket 绑定到一个众所周知的地址（名称）上使得客户端能够定位到它的位置。

第十一章 线程

上一章，学习了进程相关的知识内容，对进程有了一个比较全面的认识和理解；本章开始，将学习 Linux 应用编程中非常重要的编程技巧---线程（Thread）；与进程类似，线程是允许应用程序并发执行多个任务的一种机制，线程参与系统调度，事实上，系统调度的最小单元是线程、而并非进程。虽然线程的概念比较简单，但是其所涉及到的内容比较多，所以本章篇幅会相对比较长，大家加油！

本章将会讨论如下主题内容。

- 线程的基本概念，线程 VS 进程；
- 线程标识；
- 线程创建与回收；
- 线程取消；
- 线程终止；
- 线程分离；
- 线程同步技术；
- 线程安全。

11.1 线程概述

11.1.1 线程概念

什么是线程？

线程是参与系统调度的最小单位。它被包含在进程之中，是进程中的实际运行单位。一个线程指的是进程中一个单一顺序的控制流（或者说是执行路线、执行流），一个进程中可以创建多个线程，多个线程实现并发运行，每个线程执行不同的任务。譬如某应用程序设计了两个需要并发运行的任务 task1 和 task2，可将两个不同的任务分别放置在两个线程中。

线程是如何创建起来的？

当一个程序启动时，就有一个进程被操作系统（OS）创建，与此同时一个线程也立刻运行，该线程通常叫做程序的主线程（Main Thread），因为它是程序一开始时就运行的线程。应用程序都是以 main()做为入口开始运行的，所以 main()函数就是主线程的入口函数，main()函数所执行的任务就是主线程需要执行的任务。

所以由此可知，任何一个进程都包含一个主线程，只有主线程的进程称为单线程进程，譬如前面章节内容中所编写的所有应用程序都是单线程程序，它们只有主线程；既然有单线程进程，那自然就存在多线程进程，所谓多线程指的是除了主线程以外，还包含其它的线程，其它线程通常由主线程来创建（调用 pthread_create 创建一个新的线程），那么创建的新线程就是主线程的子线程。

主线程的重要性体现在两方面：

- 其它新的线程（也就是子线程）是由主线程创建的；
- 主线程通常会在最后结束运行，执行各种清理工作，譬如回收各个子线程。

线程的特点？

线程是程序最基本的运行单位，而进程不能运行，真正运行的是进程中的线程。当启动应用程序后，系统就创建了一个进程，可以认为进程仅仅是一个容器，它包含了线程运行所需的数据结构、环境变量等信息。

同一进程中的多个线程将共享该进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈（call stack，我们称为线程栈），自己的寄存器环境（register context）、自己的线程本地存储（thread-local storage）。

在多线程应用程序中，通常一个进程中包括了多个线程，每个线程都可以参与系统调度、被 CPU 执行，线程具有以下一些特点：

- 线程不单独存在、而是包含在进程中；
- 线程是参与系统调度的基本单位；
- 可并发执行。同一进程的多个线程之间可并发执行，在宏观上实现同时运行的效果；
- 共享进程资源。同一进程中的各个线程，可以共享该进程所拥有的资源，这首先表现在：所有线程都具有相同的地址空间（进程的地址空间），这意味着，线程可以访问该地址空间的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量等等。

线程与进程？

进程创建多个子进程可以实现并发处理多任务（本质上便是多个单线程进程），多线程同样也可以实现（一个多线程进程）并发处理多任务的需求，那我们究竟选择哪种处理方式呢？首先我们就需要来分析下多进程和多线程两种编程模型的优势和劣势。

多进程编程的劣势：

- 进程间切换开销大。多个进程同时运行（指宏观上同时运行，无特别说明，均指宏观上），微观上依然是轮流切换运行，进程间切换开销远大于同一进程的多个线程间切换的开销，通常对于一些中小型应用程序来说不划算。
- 进程间通信较为麻烦。每个进程都在各自的地址空间中、相互独立、隔离，处在于不同的地址空间中，因此相互通信较为麻烦，在上一章节给大家有所介绍。

解决方案便是使用多线程编程，多线程能够弥补上面的问题：

- 同一进程的多个线程间切换开销比较小。
- 同一进程的多个线程间通信容易。它们共享了进程的地址空间，所以它们都是在同一个地址空间中，通信容易。
- 线程创建的速度远大于进程创建的速度。
- 多线程在多核处理器上更有优势！

终上所述，多线程编程相比于多进程编程的优势是比较明显的，在实际的应用当中多线程远比多进程应用更为广泛。那既然如此，为何还存在多进程编程模型呢？难道多线程编程就不存在缺点吗？当然不是，多线程也有它的缺点、劣势，譬如多线程编程难度高，对程序员的编程功底要求比较高，因为在多线程环境下需要考虑很多的问题，例如线程安全问题、信号处理的问题等，编写与调试一个多线程程序比单线程程序困难得多。

当然除此之外，还有一些其它的缺点，这里就不再一一列举了。多进程编程通常会用在一些大型应用程序项目中，譬如网络服务器应用程序，在中小型应用程序中用的比较少。

11.1.2 并发和并行

在前面的内容中，曾多次提到了并发这个概念，与此相类似的概念还有并行、串行，这里和大家聊一聊这些概念含义的区别。

对于串行比较容易理解，它指的是一种顺序执行，譬如先完成 task1，接着做 task2、直到完成 task2，然后做 task3、直到完成 task3……依次按照顺序完成每一件事情，必须要完成上一件事才能去做下一件事，只有一个执行单元，这就是串行运行。

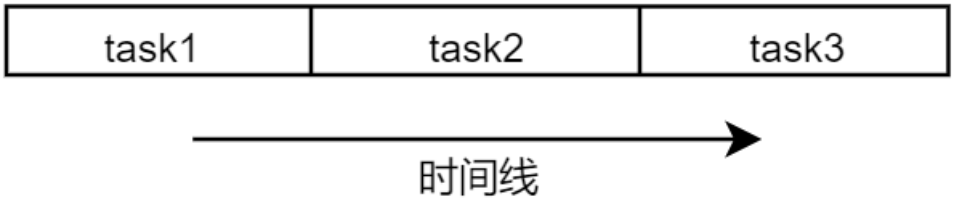


图 11.1.1 串行运行示意图

并行与串行则截然不同，并行指的是可以并排/并列执行多个任务，这样的系统，它通常有多个执行单元，所以可以实现并行运行，譬如并行运行 task1、task2、task3。

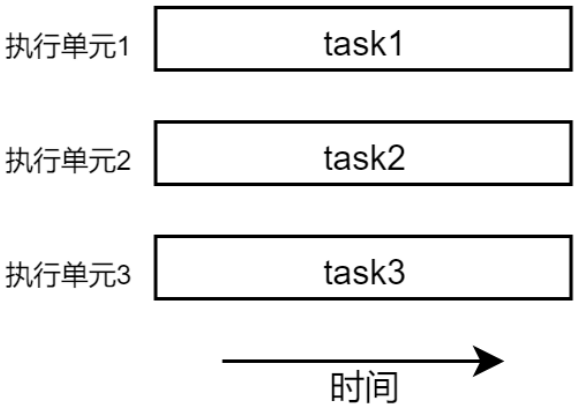


图 11.1.2 并行运行示意图 1

并行运行并不一定要同时开始运行、同时结束运行，只需满足在某一个时间段上存在多个任务被多个执行单元同时在运行着，譬如：

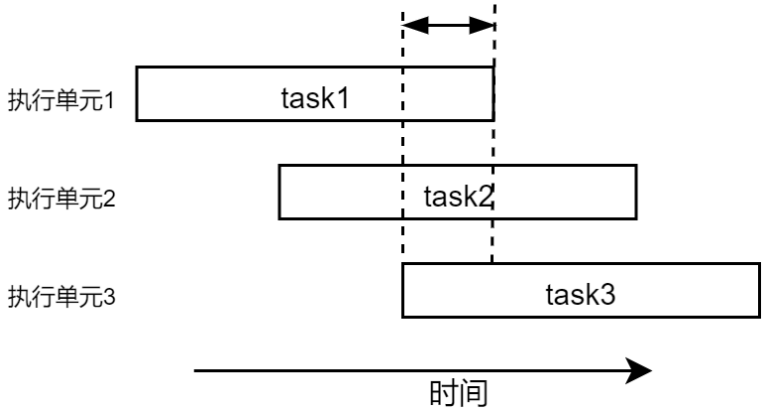


图 11.1.3 并行运行示意图 2

相比于串行和并行，并发强调的是一种时分复用，与串行的区别在于，它不必等待上一个任务完成之后在做下一个任务，可以打断当前执行的任务切换执行下一个任何，这就是时分复用。在同一个执行单元上，将时间分解成不同的片段（时间片），每个任务执行一段时间，时间一到则切换执行下一个任务，依次这样轮训（交叉/交替执行），这就是并发运行。如下图所示：

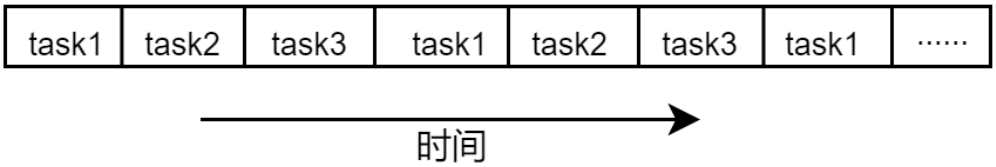


图 11.1.4 并发运行示例图

笔者在网络上看到了很多比较有意思、形象生动的比喻，用来说明串行、并行以及并发这三个概念的区别，这里笔者截取其中的一个：

- 你吃饭吃到一半，电话来了，你一直到吃完了以后才去接电话，这就说明你不支持并发也不支持并行，仅仅只是串行。
- 你吃饭吃到一半，电话来了，你停下吃饭去接了电话，电话接完后继续吃饭，这说明你支持并发。
- 你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

这里再次进行总结：

- 串行：一件事、一件事接着做
- 并发：交替做不同的事；
- 并行：同时做不同的事。

需要注意的是，并行运行情况下的多个执行单元，每一个执行单元同样也可以以并发方式运行。

从通用角度上介绍完这三个概念之后，类比到计算机系统中，首先我们需要知道两个前提条件：

- 多核处理器和单核处理器：对于单核处理器来说，只有一个执行单元，同时只能执行一条指令；而对于多核处理起来说，有多个执行单元，可以并行执行多条指令，譬如 8 核处理器，那么可以并行执行 8 条不同的指令。
- 计算机操作系统中，通常同时运行着几十上百个不同的线程，在单核或多核处理系统中都是如此！

对于单核处理器系统来说，它只有一个执行单元（譬如 I.MX6U 硬件平台，单核 Cortex-A7 SoC），只能采用并发运行系统中的线程，而肯定不可能是串行，而事实上确实如此。内核实现了调度算法，用于控制系统中所有线程的调度，简单点来说，系统中所有参与调度的线程会加入到系统的调度队列中，它们由内核控制，每一个线程执行一段时间后，由系统调度切换执行调度队列中下一个线程，依次进行。在前面章节内容中也给大家有简单地提到过系统调用的问题，关于更加详细的内容，这里便不再介绍了，我们只需有个大概的认识、了解即可！

对于多核处理器系统来说，它拥有多个执行单元，在操作系统中，多个执行单元以并行方式运行多个线程，同时每一个执行单元以并发方式运行系统中的多个线程。

同时运行

计算机处理器运行速度是非常快的，在单个处理核心虽然以并发方式运行着系统中的线程（微观上交替/交叉方式运行不同的线程），但在宏观上所表现出来的效果是同时运行着系统中的所有线程，因为处理器的运算速度太快了，交替轮训一次所花费的时间在宏观上几乎是可以忽略不计的，所以表示出来的效果就是同时运行着所有线程。

这就好比现实生活中所看到的一些事情，它所给带来的视角效果，譬如一辆车在高速上行驶，有时你会感觉到车的轮毂没有转动，一种视角暂留现象，因为车轮转动速度太快了，人眼是看不清的，会感觉车轮好像是静止的，事实上，车轮肯定是在转动着。

本小节的内容到这里就结束了，理解了本小节的内容，对于后面内容的将会有很大的帮助、也可以帮助大家快速理解后面的内容，大家加油！

11.2 线程 ID

就像每个进程都有一个进程 ID 一样，每个线程也有其对应的标识，称为线程 ID。进程 ID 在整个系统中是唯一的，但线程 ID 不同，线程 ID 只有在它所属的进程上下文中才有意义。

进程 ID 使用 pid_t 数据类型来表示，它是一个非负整数。而线程 ID 使用 pthread_t 数据类型来表示，一个线程可通过库函数 pthread_self()来获取自己的线程 ID，其函数原型如下所示：

```
#include <pthread.h>

pthread_t pthread_self(void);
```

使用该函数需要包含头文件<pthread.h>。

该函数调用总是成功，返回当前线程的线程 ID。

可以使用 pthread_equal()函数来检查两个线程 ID 是否相等，其函数原型如下所示：

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

如果两个线程 ID t1 和 t2 相等，则 pthread_equal()返回一个非零值；否则返回 0。在 Linux 系统中，使用无符号长整型（unsigned long int）来表示 pthread_t 数据类型，但是在其它系统当中，则不一定是无符号

长整型，所以我们必须将 pthread_t 作为一种不透明的数据类型加以对待，所以 pthread_equal()函数用于比较两个线程 ID 是否相等是有用的。

线程 ID 在应用程序中非常有用，原因如下：

- 很多线程相关函数，譬如后面将要学习的 pthread_cancel()、pthread_detach()、pthread_join()等，它们都是利用线程 ID 来标识要操作的目标线程；
- 在一些应用程序中，以特定线程的线程 ID 作为动态数据结构的标签，这某些应用场合颇为有用，既可以用来标识整个数据结构的创建者或属主线程，又可以确定随后对该数据结构执行操作的具体线程。

11.3 创建线程

启动程序时，创建的进程只是一个单线程的进程，称之为初始线程或主线程，本小节我们讨论如何创建一个新的线程。

主线程可以使用库函数 pthread_create()负责创建一个新的线程，创建出来的新线程被称为主线程的子线程，其函数原型如下所示：

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

使用该函数需要包含头文件<pthread.h>。

函数参数和返回值含义如下：

thread：pthread_t 类型指针，当 pthread_create()成功返回时，新创建的线程的线程 ID 会保存在参数 thread 所指向的内存中，后续的线程相关函数会使用该标识来引用此线程。

attr：pthread_attr_t 类型指针，指向 pthread_attr_t 类型的缓冲区，pthread_attr_t 数据类型定义了线程的各种属性，关于线程属性将会在 11.8 小节介绍。如果将参数 attr 设置为 NULL，那么表示将线程的所有属性设置为默认值，以此创建新线程。

start_routine：参数 start_routine 是一个函数指针，指向一个函数，新创建的线程从 start_routine()函数开始运行，该函数返回值类型为 void *，并且该函数的参数只有一个 void *，其实这个参数就是 pthread_create()函数的第四个参数 arg。如果需要向 start_routine()传递的参数有一个以上，那么需要把这些参数放到一个结构体中，然后把这个结构体对象的地址作为 arg 参数传入。

arg：传递给 start_routine()函数的参数。一般情况下，需要将 arg 指向一个全局或堆变量，意思就是说在线程的生命周期中，该 arg 指向的对象必须存在，否则如果线程中访问了该对象将会出现错误。当然也可将参数 arg 设置为 NULL，表示不需要传入参数给 start_routine()函数。

返回值：成功返回 0；失败时将返回一个错误号，并且参数 thread 指向的内容是不确定的。

注意 pthread_create()在调用失败时通常会返回错误码，它并不像其它库函数或系统调用一样设置 errno，每个线程都提供了全局变量 errno 的副本，这只是为了与使用 errno 到的函数进行兼容，在线程中，从函数中返回错误码更为清晰整洁，不需要依赖那些随着函数执行不断变化的全局变量，这样可以把错误的范围限制在引起出错的函数中。

线程创建成功，新线程就会加入到系统调度队列中，获取到 CPU 之后就会立马从 start_routine()函数开始运行该线程的任务；调用 pthread_create()函数后，通常我们无法确定系统接着会调度哪一个线程来使用 CPU 资源，先调度主线程还是新创建的线程呢（而在多核 CPU 或多 CPU 系统中，多核线程可能会在不同的核心上同时执行）？如果程序对执行顺序有强制要求，那么就必须采用一些同步技术来实现。这与前面学习父、子进程时也出现了这个问题，无法确定父进程、子进程谁先被系统调度。

使用示例

使用 pthread_create()函数创建一个除主线程之外的新线程，示例代码如下所示：

示例代码 11.3.1 pthread_create()创建线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程: 进程 ID<%d>  线程 ID<%lu>\n", getpid(), pthread_self());
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    int ret;

    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "Error: %s\n", strerror(ret));
        exit(-1);
    }

    printf("主线程: 进程 ID<%d>  线程 ID<%lu>\n", getpid(), pthread_self());
    sleep(1);
    exit(0);
}
```

应该将 pthread_t 作为一种不透明的数据类型加以对待，但是在示例代码中需要打印线程 ID，所以要明确其数据类型，示例代码中使用了 printf()函数打印线程 ID 时，将其作为 unsigned long int 数据类型，在 Linux 系统下，确实是使用 unsigned long int 来表示 pthread_t，所以这样做没有问题！

主线程休眠了 1 秒钟，原因在于，如果主线程不进行休眠，它就可能会立马退出，这样可能会导致新创建的线程还没有机会运行，整个进程就结束了。

在主线程和新线程中，分别通过 getpid()和 pthread_self()来获取进程 ID 和线程 ID，将结果打印出来，运行结果如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
/tmp/cc8nwjLD.o：在函数‘main’中：
testApp.c:(.text+0x69)：对‘pthread_create’未定义的引用
collect2: error: ld returned 1 exit status
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.3.1 编译报错

编译时出现了错误，提示“对‘pthread_create’未定义的引用”，示例代码确实已经包含了<pthread.h>头文件，但为什么会出现这样的报错，仔细看，这个报错是出现在程序代码链接时、而并非是编译过程，所以可知这是链接库的文件，如何解决呢？

```
gcc -o testApp testApp.c -lpthread
```

使用-l 选项指定链接库 pthread，原因在于 pthread 不在 gcc 的默认链接库中，所以需要手动指定。再次编译便不会有问题了，如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
主线程：进程ID<30793>  线程ID<140458989827840>
新线程：进程ID<30793>  线程ID<140458981488384>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.3.2 测试结果

从打印信息可知，正如前面所介绍那样，两个线程的进程 ID 相同，说明新创建的线程与主线程本来就属于同一个进程，但是它们的线程 ID 不同。从打印结果可知，Linux 系统下线程 ID 数值非常大，看起来像是一个指针。

11.4 终止线程

在示例代码 11.3.1 中，我们在新线程的启动函数（线程 start 函数）new_thread_start()通过 return 返回之后，意味着该线程已经终止了，除了在线程 start 函数中执行 return 语句终止线程外，终止线程的方式还有多种，可以通过如下方式终止线程的运行：

- 线程的 start 函数执行 return 语句并返回指定值，返回值就是线程的退出码；
- 线程调用 pthread_exit()函数；
- 调用 pthread_cancel()取消线程（将在 11.6 小节介绍）；

如果进程中的任意线程调用 exit()、_exit()或者_Exit()，那么将会导致整个进程终止，这里需要注意！

pthread_exit()函数将终止调用它的线程，其函数原型如下所示：

```
#include <pthread.h>

void pthread_exit(void *retval);
```

使用该函数需要包含头文件<pthread.h>。

参数 retval 的数据类型为 void *，指定了线程的返回值、也就是线程的退出码，该返回值可由另一个线程通过调用 pthread_join()来获取；同理，如果线程是在 start 函数中执行 return 语句终止，那么 return 的返回值也是可以通过 pthread_join()来获取的。

参数 retval 所指向的内容不应分配于线程栈中，因为线程终止后，将无法确定线程栈的内容是否有效；出于同样的理由，也不应在线程栈中分配线程 start 函数的返回值。

调用 pthread_exit()相当于在线程的 start 函数中执行 return 语句，不同之处在于，可在线程 start 函数所调用的任意函数中调用 pthread_exit()来终止线程。如果主线程调用了 pthread_exit()，那么主线程也会终止，但其它线程依然正常运行，直到进程中的所有线程终止才会使得进程终止。

使用示例

示例代码 11.4.1 pthread_exit()终止线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程 start\n");
    sleep(1);
    printf("新线程 end\n");
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid;
    int ret;

    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "Error: %s\n", strerror(ret));
        exit(-1);
    }

    printf("主线程 end\n");
    pthread_exit(NULL);
    exit(0);
}
```

新线程中调用 sleep()休眠，保证主线程先调用 pthread_exit()终止，休眠结束之后新线程也调用 pthread_exit()终止，编译测试看看打印结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
主线程end
新线程start
新线程end
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.4.1 测试结果

正如上面介绍到，主线程调用 pthread_exit()终止之后，整个进程并没有结束，而新线程还在继续运行。

11.5 回收线程

在父、子进程当中，父进程可通过 wait()函数（或其变体 waitpid()）阻塞等待子进程退出并获取其终止状态，回收子进程资源；而在线程当中，也需要如此，通过调用 pthread_join()函数来阻塞等待线程的终止，并获取线程的退出码，回收线程资源；pthread_join()函数原型如下所示：

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

使用该函数需要包含头文件<pthread.h>。

函数参数和返回值含义如下：

thread：pthread_join()等待指定线程的终止，通过参数 thread（线程 ID）指定需要等待的线程；

retval：如果参数 retval 不为 NULL，则 pthread_join()将目标线程的退出状态（即目标线程通过 pthread_exit()退出时指定的返回值或者在线程 start 函数中执行 return 语句对应的返回值）复制到*retval 所指向的内存区域；如果目标线程被 pthread_cancel()取消，则将 PTHREAD_CANCELED 放在*retval 中。如果对目标线程的终止状态不感兴趣，则可将参数 retval 设置为 NULL。

返回值：成功返回 0；失败将返回错误码。

调用 pthread_join()函数将会以阻塞的形式等待指定的线程终止，如果该线程已经终止，则 pthread_join()立刻返回。如果多个线程同时尝试调用 pthread_join()等待指定线程的终止，那么结果将是不确定的。

若线程并未分离（detached，将在 11.6.1 小节介绍），则必须使用 pthread_join()来等待线程终止，回收线程资源；如果线程终止后，其它线程没有调用 pthread_join()函数来回收该线程，那么该线程将变成僵尸线程，与僵尸进程的概念相类似；同样，僵尸线程除了浪费系统资源外，若僵尸线程积累过多，那么会导致应用程序无法创建新的线程。

当然，如果进程中存在着僵尸线程并未得到回收，当进程终止之后，进程会被其父进程回收，所以僵尸线程同样也会被回收。

所以，通过上面的介绍可知，pthread_join()执行的功能类似于针对进程的 waitpid()调用，不过二者之间存在一些显著差别：

- 线程之间关系是对等的。进程中的任意线程均可调用 pthread_join()函数来等待另一个线程的终止。譬如，如果线程 A 创建了线程 B，线程 B 再创建线程 C，那么线程 A 可以调用 pthread_join()等待线程 C 的终止，线程 C 也可以调用 pthread_join()等待线程 A 的终止；这与进程间层次关系不同，父进程如果使用 fork()创建了子进程，那么它也是唯一能够对子进程调用 wait()的进程，线程之间不存在这样的关系。
- 不能以非阻塞的方式调用 pthread_join()。对于进程，调用 waitpid()既可以实现阻塞方式等待、也可以实现非阻塞方式等待。

使用示例

示例代码 11.5.1 pthread_join()等待线程终止

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程 start\n");
    sleep(2);
    printf("新线程 end\n");
    pthread_exit((void *)10);
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

主线程调用 pthread_create()创建新线程之后，新线程执行 new_thread_start()函数，而在主线程中调用 pthread_join()阻塞等待新线程终止，新线程终止后，pthread_join()返回，将目标线程的退出码保存在*tret 所指向的内存中。测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程start
新线程end
新线程终止，code=10
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.5.1 测试结果

11.6 取消线程

在通常情况下，进程中的多个线程会并发执行，每个线程各司其职，直到线程的任务完成之后，该线程中会调用 pthread_exit()退出，或在线程 start 函数执行 return 语句退出。

有时候，在程序设计需求当中，需要向一个线程发送一个请求，要求它立刻退出，我们把这种操作称为取消线程，也就是向指定的线程发送一个请求，要求其立刻终止、退出。譬如，一组线程正在执行一个运算，一旦某个线程检测到错误发生，需要其它线程退出，取消线程这项功能就派上用场了。

本小节就来讨论 Linux 系统下的线程取消机制。

11.6.1 取消一个线程

通过调用 pthread_cancel()库函数向一个指定的线程发送取消请求，其函数原型如下所示：

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

使用该函数需要包含头文件<pthread.h>，参数 thread 指定需要取消的目标线程；成功返回 0，失败将返回错误码。

发出取消请求之后，函数 pthread_cancel()立即返回，不会等待目标线程的退出。默认情况下，目标线程也会立刻退出，其行为表现为如同调用了参数为 PTHREAD_CANCELED（其实就是(void *)-1）的 pthread_exit()函数，但是，线程可以设置自己不被取消或者控制如何被取消（11.6.2 小节介绍），所以 pthread_cancel()并不会等待线程终止，仅仅只是提出请求。

使用示例

示例代码 11.6.1 pthread_cancel()取消线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程--running\n");
```

```
    for ( ; ; )
        sleep(1);
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 向新线程发送取消请求 */
    ret = pthread_cancel(tid);
    if (ret) {
        fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

主线程创建新线程，新线程 new_thread_start()函数直接运行 for 死循环；主线程休眠一段时间后，调用 pthread_cancel()向新线程发送取消请求，接着再调用 pthread_join()等待新线程终止、获取其终止状态，将线程退出码打印出来。测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--running
新线程终止，code=-1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.6.1 测试结果

由打印结果可知，当主线程发送取消请求之后，新线程便退出了，而且退出码为-1，也就是PTHREAD_CANCELED。

11.6.2 取消状态以及类型

默认情况下，线程是响应其它线程发送过来的取消请求的，响应请求然后退出线程。当然，线程可以选择不被取消或者控制如何被取消，通过 pthread_setcancelstate()和 pthread_setcanceltype()来设置线程的取消性状态和类型。

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

使用这些函数需要包含头文件<pthread.h>，pthread_setcancelstate()函数会将调用线程的取消性状态设置为参数 state 中给定的值，并将线程之前的取消性状态保存在参数 oldstate 指向的缓冲区中，如果对之前的状态不感兴趣，Linux 允许将参数 oldstate 设置为 NULL；pthread_setcancelstate()调用成功将返回 0，失败返回非 0 值的错误码。

pthread_setcancelstate()函数执行的设置取消性状态和获取旧状态操作，这两步是一个原子操作。

参数 state 必须是以下值之一：

- **PTHREAD_CANCEL_ENABLE**：线程可以取消，这是新创建的线程取消性状态的默认值，所以新建线程以及主线程默认都是可以取消的。
- **PTHREAD_CANCEL_DISABLE**：线程不可被取消，如果此类线程接收到取消请求，则会将请求挂起，直至线程的取消性状态变为 PTHREAD_CANCEL_ENABLE。

使用示例

修改示例代码 11.6.1，在新线程的 new_thread_start()函数中调用 pthread_setcancelstate()函数将线程的取消性状态设置为 PTHREAD_CANCEL_DISABLE，我们来试试，此时主线程还能不能取消新线程，示例代码如下所示：

示例代码 11.6.2 pthread_setcancelstate()使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
```

```
{
    /* 设置为不可被取消 */
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    for ( ; ; ) {
        printf("新线程--running\n");
        sleep(2);
    }
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 向新线程发送取消请求 */
    ret = pthread_cancel(tid);
    if (ret) {
        fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

新线程 new_thread_start()函数中调用 pthread_setcancelstate()将自己设置为不可被取消，主线程延时 1 秒钟之后调用 pthread_cancel()向新线程发送取消请求，那么此时新线程是不会终止的，pthread_cancel()立刻返回之后进入到 pthread_join()函数，那么此时会被阻塞等待新线程终止，接下来运行测试看看，结果会不会是这样：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--running
新线程--running
新线程--running
新线程--running
新线程--running
新线程--running
```

图 11.6.2 测试结果

测试结果确实如此，将一直重复打印"新线程--running"，因为新线程是一个死循环（测试完成按 Ctrl+C 退出）。

pthread_setcanceltype()函数

如果线程的取消性状态为 PTHREAD_CANCEL_ENABLE，那么对取消请求的处理则取决于线程的取消性类型，该类型可以通过调用 pthread_setcanceltype()函数来设置，它的参数 type 指定了需要设置的类型，而线程之前的取消性类型则会保存在参数 oldtype 所指向的缓冲区中，如果对之前的类型不敢兴趣，Linux 下允许将参数 oldtype 设置为 NULL。同样 pthread_setcanceltype()函数调用成功将返回 0，失败返回非 0 值的错误码。

pthread_setcanceltype()函数执行的设置取消性类型和获取旧类型操作，这两步是一个原子操作。

参数 type 必须是以下值之一：

- **PTHREAD_CANCEL_DEFERRED**：取消请求到来时，线程还是继续运行，取消请求被挂起，直到线程到达某个取消点（cancellation point，将在 11.6.3 小节介绍）为止，这是所有新建线程包括主线程默认的取消性类型。
- **PTHREAD_CANCEL_ASYNCHRONOUS**：可能会在任何时间点（也许是立即取消，但不一定）取消线程，这种取消性类型应用场景很少，不再介绍！

当某个线程调用 fork()创建子进程时，子进程会继承调用线程的取消性状态和取消性类型，而当某线程调用 exec 函数时，会将新程序主线程的取消性状态和类型重置为默认值，也就是 PTHREAD_CANCEL_ENABLE 和 PTHREAD_CANCEL_DEFERRED。

11.6.3 取消点

若将线程的取消性类型设置为 PTHREAD_CANCEL_DEFERRED 时（线程可以取消状态下），收到其它线程发送过来的取消请求时，仅当线程抵达某个取消点时，取消请求才会起作用。

那什么是取消点呢？所谓取消点其实就是一系列函数，当执行到这些函数的时候，才会真正响应取消请求，这些函数就是取消点；在没有出现取消点时，取消请求是无法得到处理的，究其原因在于系统认为，但没有到达取消点时，线程此时正在执行的工作是不能被停止的，正在执行关键代码，此时终止线程将可能会导致出现意想不到的异常发生。

取消点函数包括哪些呢？下表给大家简单地列出了一些：

表 11.6.1 可作为取消点的函数

accept()	mq_timedsend()	pthread_join()	sendto()
aio_suspend()	msgrcv()	pthread_testcancel()	sigsuspend()
clock_nanosleep()	msgsnd()	pwrite()	sigtimedwait()
close()	msync()	read()	sigwait()
connect()	nanosleep()	readv()	sigwaitinfo()
creat()	open()	recv()	sleep()
fcntl()	openat()	recvfrom()	system()
fdatasync()	pause()	recvmsg()	tcdrain()
fsync()	poll()	select()	wait()
lockf()	pread()	sem_timedwait()	waitid()
mq_receive()	pselect()	sem_wait()	waitpid()
mq_send()	pthread_cond_timedwait()	send()	write()
mq_timedreceive()	pthread_cond_wait()	sendmsg()	writev()

除了表 11.6.1 所列函数之外，还有大量的函数，系统实现可以将其作为取消点，这里便不再一一列举出来了，大家也可以通过 man 手册进行查询，命令为"man 7 pthreads"，如下所示：

```
Cancellation points
    POSIX.1 specifies that certain functions must, and certain other functions may, be cancellation points.  If a thread is cancelable,
    a cancellation request is pending for the thread, then the thread is canceled when it calls a function that is a cancellation point.

    The following functions are required to be cancellation points by POSIX.1-2001 and/or POSIX.1-2008:

        accept()
        aio_suspend()
        clock_nanosleep()
        close()
        connect()
        creat()
        fcntl() F_SETLKW
        fdatasync()
        fsync()
        getmsg()
        getpmsg()
        lockf() F_LOCK
        mq_receive()
        mq_send()
        mq_timedreceive()
        mq_timedsend()
        msgrcv()
        msgsnd()
        msync()
        nanosleep()
        open()
        openat() [Added in POSIX.1-2008]
        pause()
        poll()
        pread()
        pselect()
        pthread_cond_timedwait()
        pthread_cond_wait()
        pthread_join()
        pthread_testcancel()
```

图 11.6.3 查看可作为取消点的函数

线程在调用这些函数时，如果收到了取消请求，那么线程便会遭到取消；除了这些作为取消点的函数之外，不得将任何其它函数视为取消点（亦即，调用这些函数不会招致取消）。

示例代码 11.6.1 中，新线程处于 for 循环之中，调用 sleep()休眠，由表 11.6.1 可知，sleep()函数可以作为取消点（printf 可能也是），当新线程接收到取消请求之后，便会立马退出，当如果将其修改为如下：

```
static void *new_thread_start(void *arg)
{
    printf("新线程--running\n");
    for ( ; ; ) {
    }
    return (void *)0;
```

```
}
```

那么线程将永远无法被取消，因为这里不存在取消点。大家可以将代码进行修改测试，看结果是不是如此！

11.6.4 线程可取消性的检测

假设线程执行的是一个不含取消点的循环（譬如 for 循环、while 循环），那么这时线程永远也不会响应取消请求，也就意味着除了线程自己主动退出，其它线程将无法通过向它发送取消请求而终止它，就如上小节最后给大家列举的例子。

在实际应用程序当中，确实会遇到这种情况，线程最终运行在一个循环当中，该循环体内执行的函数不存在任何一个取消点，但实际项目需求是：该线程必须可以被其它线程通过发送取消请求的方式终止，那这个时候怎么办？此时可以使用 pthread_testcancel()，该函数目的很简单，就是产生一个取消点，线程如果已有处于挂起状态的取消请求，那么只要调用该函数，线程就会随之终止。其函数原型如下所示：

```
#include <pthread.h>

void pthread_testcancel(void);
```

功能测试

接下来进行一个测试，主线程创建一个新的进程，新进程的取消性状态和类型置为默认，新进程最终执行的是一个不含取消点的循环；主线程向新线程发送取消请求，示例代码如下所示：

示例代码 11.6.3 不含取消点的循环

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    printf("新线程--start run\n");
    for ( ; ; ) {

    }
    return (void *)0;
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;
```

```
    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 向新线程发送取消请求 */
    ret = pthread_cancel(tid);
    if (ret) {
        fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

新线程的 new_thread_start()函数中是一个 for 死循环，没有执行任何函数，所以是一个没有取消点的循环体，主线程调用 pthread_cancel()是无法将其终止的，接下来测试下结果是否如此：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--start run
```

图 11.6.4 测试结果

执行完之后，程序一直会没有退出，说明主线程确实无法终止新线程。接下来再做一个测试，在 new_thread_start 函数的 for 循环体中执行 pthread_testcancel()函数，如下所示：

示例代码 11.6.4 使用 pthread_testcancel()产生取消点

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

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

static void *new_thread_start(void *arg)
{
	printf("新线程--start run\n");
	for ( ; ; ) {
		pthread_testcancel();
	}
	return (void *)0;
}

int main(void)
{
	pthread_t tid;
	void *tret;
	int ret;

	/* 创建新线程 */
	ret = pthread_create(&tid, NULL, new_thread_start, NULL);
	if (ret) {
		fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
		exit(-1);
	}

	sleep(1);

	/* 向新线程发送取消请求 */
	ret = pthread_cancel(tid);
	if (ret) {
		fprintf(stderr, "pthread_cancel error: %s\n", strerror(ret));
		exit(-1);
	}

	/* 等待新线程终止 */
	ret = pthread_join(tid, &tret);
	if (ret) {
		fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
		exit(-1);
	}
	printf("新线程终止, code=%ld\n", (long)tret);
```

```
    exit(0);
}
```

如果 pthread_testcancel()可以产生取消点，那么主线程便可以终止新线程，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--start run
新线程终止，code=-1
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.6.5 测试结果

从打印结果可知，确实如上面介绍那样，pthread_testcancel()函数就是取消点。

11.7 分离线程

默认情况下，当线程终止时，其它线程可以通过调用 pthread_join()获取其返回状态、回收线程资源，有时，程序员并不关系线程的返回状态，只是希望系统在线程终止时能够自动回收线程资源并将其移除。在这种情况下，可以调用 pthread_detach()将指定线程进行分离，也就是分离线程，pthread_detach()函数原型如下所示：

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

使用该函数需要包含头文件<pthread.h>，参数 thread 指定需要分离的线程，函数 pthread_detach()调用成功将返回 0；失败将返回一个错误码。

一个线程既可以将另一个线程分离，同时也可以将自己分离，譬如：

```
pthread_detach(pthread_self());
```

一旦线程处于分离状态，就不能再使用 pthread_join()来获取其终止状态，此过程是不可逆的，一旦处于分离状态之后便不能再恢复到之前的状态。处于分离状态的线程，当其终止后，能够自动回收线程资源。

使用示例

示例代码 11.7.1 pthread_detach()分离线程使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void *new_thread_start(void *arg)
{
    int ret;

    /* 自行分离 */
    ret = pthread_detach(pthread_self());
    if (ret) {
```

```
        fprintf(stderr, "pthread_detach error: %s\n", strerror(ret));
        return NULL;
    }

    printf("新线程 start\n");
    sleep(2);     //休眠 2 秒钟
    printf("新线程 end\n");
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);     //休眠 1 秒钟

    /* 等待新线程终止 */
    ret = pthread_join(tid, NULL);
    if (ret)
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));

    pthread_exit(NULL);
}
```

示例代码中，主线程创建新的线程之后，休眠 1 秒钟，调用 pthread_join()等待新线程终止；新线程调用 pthread_detach(pthread_self())将自己分离，休眠 2 秒钟之后 pthread_exit()退出线程；主线程休眠 1 秒钟是能够确保调用 pthread_join()函数时新线程已经将自己分离了，所以按照上面的介绍可知，此时主线程调用 pthread_join()必然会失败，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程start
pthread_join error: Invalid argument
新线程end
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.7.1 测试结果

打印结果正如我们所料，主线程调用 pthread_join()确实会出错，错误提示为“Invalid argument”。

11.8 注册线程清理处理函数

9.1.2 小节学习了 atexit()函数，使用 atexit()函数注册进程终止处理函数，当进程调用 exit()退出时就会执行进程终止处理函数；其实，当线程退出时也可以这样做，当线程终止退出时，去执行这样的处理函数，我们把这个称为线程清理函数（thread cleanup handler）。

与进程不同，一个线程可以注册多个清理函数，这些清理函数记录在栈中，每个线程都可以拥有一个清理函数栈，栈是一种先进后出的数据结构，也就是说它们的执行顺序与注册（添加）顺序相反，当执行完所有清理函数后，线程终止。

线程通过函数 pthread_cleanup_push()和 pthread_cleanup_pop()分别负责向调用线程的清理函数栈中添加和移除清理函数，函数原型如下所示：

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

使用这些函数需要包含头文件<pthread.h>。

调用 pthread_cleanup_push()向清理函数栈中添加一个清理函数，第一个参数 routine 是一个函数指针，指向一个需要添加的清理函数，routine()函数无返回值，只有一个 void *类型参数；第二个参数 arg，当调用清理函数 routine()时，将 arg 作为 routine()函数的参数。

既然有添加，自然就会伴随着删除，就好比对应入栈和出栈，调用函数 pthread_cleanup_pop()可以将清理函数栈中最顶层（也就是最后添加的函数，最后入栈）的函数移除。

当线程执行以下动作时，清理函数栈中的清理函数才会被执行：

- 线程调用 pthread_exit()退出时；
- 线程响应取消请求时；
- 用非 0 参数调用 pthread_cleanup_pop()

除了以上三种情况之外，其它方式终止线程将不会执行线程清理函数，譬如在线程 start 函数中执行 return 语句退出时不会执行清理函数。

函数 pthread_cleanup_pop()的 execute 参数，可以取值为 0，也可以为非 0；如果为 0，清理函数不会被调用，只是将清理函数栈中最顶层的函数移除；如果参数 execute 为非 0，则除了将清理函数栈中最顶层的函数移除之外，还会该清理函数。

尽管上面我们将 pthread_cleanup_push()和 pthread_cleanup_pop()称之为函数，但它们是通过宏来实现，可展开为分别由{和}所包裹的语句序列，所以必须在与线程相同的作用域中以匹配对的形式使用，必须一一对应着来使用，譬如：

```
pthread_cleanup_push(cleanup, NULL);
```

```
pthread_cleanup_push(cleanup, NULL);
pthread_cleanup_push(cleanup, NULL);
......
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
pthread_cleanup_pop(0);
```

否则会编译报错，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
testApp.c: In function 'new_thread_start':
testApp.c:24:1: error: expected 'while' before 'int'
 int main(void)
 ^
testApp.c:46:1: error: expected declaration or statement at end of input
 }
 ^
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
testApp.c:46:1: error: expected declaration or statement at end of input
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.8.1 编译报错

使用示例

示例代码 11.8.1 给出了一个使用线程清理函数的例子，虽然例子并没有什么实际作用，当它描述了其中所涉及到的清理机制。

示例代码 11.8.1 pthread_cleanup_push()注册线程清理函数

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

static void *new_thread_start(void *arg)
{
    printf("新线程--start run\n");
    pthread_cleanup_push(cleanup, "第 1 次调用");
    pthread_cleanup_push(cleanup, "第 2 次调用");
    pthread_cleanup_push(cleanup, "第 3 次调用");

    sleep(2);
```

```
    pthread_exit((void *)0);        //线程终止

    /* 为了与 pthread_cleanup_push 配对,不添加程序编译会通不过 */
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

主线程创建新线程之后，调用 pthread_join()等待新线程终止；新线程调用 pthread_cleanup_push()函数添加线程清理函数，调用了三次，但每次添加的都是同一个函数，只是传入的参数不同；清理函数添加完成，休眠一段时间之后，调用 pthread_exit()退出。之后还调用了 3 次 pthread_cleanup_pop()，在这里的目的仅仅只是为了与 pthread_cleanup_push()配对使用，否则编译不通过。接下来编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--start run
cleanup: 第3次调用
cleanup: 第2次调用
cleanup: 第1次调用
新线程终止，code=0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.8.2 测试结果

从打印结果可知，先添加到线程清理函数栈中的函数会后被执行，添加顺序与执行顺序相反。

将新线程中调用的 pthread_exit()替换为 return，在进行测试，发现并不会执行清理函数。

有时在线程功能设计中，线程清理函数并不一定需要在线程退出时才执行，譬如当完成某一个步骤之后，就需要执行线程清理函数，此时我们可以调用 pthread_cleanup_pop()并传入非 0 参数，来手动执行线程清理函数，示例代码如下所示：

示例代码 11.8.2 手动执行线程清理函数

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

static void cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

static void *new_thread_start(void *arg)
{
    printf("新线程--start run\n");
    pthread_cleanup_push(cleanup, "第 1 次调用");
    pthread_cleanup_push(cleanup, "第 2 次调用");
    pthread_cleanup_push(cleanup, "第 3 次调用");

    pthread_cleanup_pop(1);       //执行最顶层的清理函数
    printf("~~~~~~~~~~~~~~~~~~\n");
    sleep(2);
    pthread_exit((void *)0);      //线程终止

    /* 为了与 pthread_cleanup_push 配对 */
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
```

```
}

int main(void)
{
    pthread_t tid;
    void *tret;
    int ret;

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, &tret);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }
    printf("新线程终止, code=%ld\n", (long)tret);

    exit(0);
}
```

上述代码中，在新线程调用 pthread_exit()之前，先调用 pthread_cleanup_pop(1)手动运行了最顶层的清理函数，并将其从栈中移除，测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
新线程--start run
cleanup: 第3次调用
~~~~~~~~~~~~~~~~~
cleanup: 第2次调用
cleanup: 第1次调用
新线程终止, code=0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.8.3 测试结果

从打印结果可知，调用 pthread_cleanup_pop(1)执行了最后一次注册的清理函数，调用 pthread_exit()退出线程时执行了 2 次清理函数，因为前面调用 pthread_cleanup_pop()已经将顶层的清理函数移除栈中了，自然在退出时就不会再执行了。

11.9 线程属性

如前所述，调用 pthread_create()创建线程，可对新建线程的各种属性进行设置。在 Linux 下，使用 pthread_attr_t 数据类型定义线程的所有属性，本书并不打算详细讨论这些属性，以介绍为主，简单地了解下线程属性。

调用 pthread_create()创建线程时，参数 attr 设置为 NULL，表示使用属性的默认值创建线程。如果不使用默认值，参数 attr 必须要指向一个 pthread_attr_t 对象，而不能使用 NULL。当定义 pthread_attr_t 对象之后，需要使用 pthread_attr_init()函数对该对象进行初始化操作，当对象不再使用时，需要使用 pthread_attr_destroy()函数将其销毁，函数原型如下所示：

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

使用这些函数需要包含头文件<pthread.h>，参数 attr 指向一个 pthread_attr_t 对象，即需要进行初始化的线程属性对象。在调用成功时返回 0，失败将返回一个非 0 值的错误码。

调用 pthread_attr_init()函数会将指定的 pthread_attr_t 对象中定义的各种线程属性初始化为它们各自对应的默认值。

pthread_attr_t 数据结构中包含的属性比较多，本小节并不会一一点出，可能比较关注属性包括：线程栈的位置和大小、线程调度策略和优先级，以及线程的分离状态属性等。Linux 为 pthread_attr_t 对象的每种属性提供了设置属性的接口以及获取属性的接口。

11.9.1 线程栈属性

每个线程都有自己的栈空间，pthread_attr_t 数据结构中定义了栈的起始地址以及栈大小，调用函数 pthread_attr_getstack()可以获取这些信息，函数 pthread_attr_setstack()对栈起始地址和栈大小进行设置，其函数原型如下所示：

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *attr, void **stackaddr, size_t *stacksize);
```

使用这些函数需要包含头文件<pthread.h>，函数 pthread_attr_getstack()，参数和返回值含义如下：

attr：参数 attr 指向线程属性对象。

stackaddr：调用 pthread_attr_getstack()可获取栈起始地址，并将起始地址信息保存在*stackaddr 中；

stacksize：调用 pthread_attr_getstack()可获取栈大小，并将栈大小信息保存在参数 stacksize 所指向的内存中；

返回值：成功返回 0，失败将返回一个非 0 值的错误码。

函数 pthread_attr_setstack()，参数和返回值含义如下：

attr：参数 attr 指向线程属性对象。

stackaddr：设置栈起始地址为指定值。

stacksize：设置栈大小为指定值；

返回值：成功返回 0，失败将返回一个非 0 值的错误码。

如果想单独获取或设置栈大小、栈起始地址，可以使用下面这些函数：

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

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
```

使用示例

创建新的线程，将线程的栈大小设置为 4Kbyte。

示例代码 11.9.1 设置线程栈大小 pthread_attr_getstack()

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

static void *new_thread_start(void *arg)
{
    puts("Hello World!");
    return (void *)0;
}

int main(int argc, char *argv[])
{
    pthread_attr_t attr;
    size_t stacksize;
    pthread_t tid;
    int ret;

    /* 对 attr 对象进行初始化 */
    pthread_attr_init(&attr);

    /* 设置栈大小为 4K */
    pthread_attr_setstacksize(&attr, 4096);

    /* 创建新线程 */
    ret = pthread_create(&tid, &attr, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待新线程终止 */
    ret = pthread_join(tid, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
```

```
    }

    /* 销毁 attr 对象 */
    pthread_attr_destroy(&attr);
    exit(0);
}
```

11.9.2 分离状态属性

前面介绍了线程分离的概念，如果对现已创建的某个线程的终止状态不感兴趣，可以使用pthread_detach()函数将其分离，那么该线程在退出时，操作系统会自动回收它所占用的资源。

如果我们在创建线程时就确定要将该线程分离，可以修改 pthread_attr_t 结构中的 detachstate 线程属性，让线程一开始运行就处于分离状态。调用函数 pthread_attr_setdetachstate()设置 detachstate 线程属性，调用 pthread_attr_getdetachstate()获取 detachstate 线程属性，其函数原型如下所示：

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

需要包含头文件<pthread.h>，参数 attr 指向 pthread_attr_t 对象；调用 pthread_attr_setdetachstate()函数将 detachstate 线程属性设置为参数 detachstate 所指定的值，参数 detachstate 取值如下：

- **PTHREAD_CREATE_DETACHED**：新建线程一开始运行便处于分离状态，以分离状态启动线程，无法被其它线程调用 pthread_join()回收，线程结束后由操作系统收回其所占用的资源；
- **PTHREAD_CREATE_JOINABLE**：这是 detachstate 线程属性的默认值，正常启动线程，可以被其它线程获取终止状态信息。

函数 pthread_attr_getdetachstate()用于获取 detachstate 线程属性，将 detachstate 线程属性保存在参数 detachstate 所指定的内存中。

使用示例

示例代码 11.9.2 给出了以分离状态启动线程的示例。

示例代码 11.9.2 以分离状态启动线程

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

static void *new_thread_start(void *arg)
{
    puts("Hello World!");
    return (void *)0;
}

int main(int argc, char *argv[])
{
    pthread_attr_t attr;
```

```
    pthread_t tid;
    int ret;

    /* 对 attr 对象进行初始化 */
    pthread_attr_init(&attr);

    /* 设置以分离状态启动线程 */
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    /* 创建新线程 */
    ret = pthread_create(&tid, &attr, new_thread_start, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    sleep(1);

    /* 销毁 attr 对象 */
    pthread_attr_destroy(&attr);
    exit(0);
}
```

11.10 线程安全

当我们编写的程序是一个多线程应用程序时，就不得不考虑到线程安全的问题，确保我们编写的程序是一个线程安全（thread-safe）的多线程应用程序，什么是线程安全以及如何保证线程安全？带着这些问题，本小节将讨论线程安全相关的话题。

Tips：在阅读本小节内容之前，建议先阅读第十二章内容，这章内容原本计划是放在本小节内容之前的，但由于排版问题，不得不将其单独列为一章。

11.10.1 线程栈

进程中创建的每个线程都有自己的栈地址空间，将其称为线程栈。譬如主线程调用 pthread_create()创建了一个新的线程，那么这个新的线程有它自己独立的栈地址空间、而主线程也有它自己独立的栈地址空间。通过 11.9.1 小节可知，在创建一个新的线程时，可以配置线程栈的大小以及起始地址，当然在大部分情况下，保持默认即可！

既然每个线程都有自己的栈地址空间，那么每个线程运行过程中所定义的自动变量（局部变量）都是分配在自己的线程栈中的，它们不会相互干扰。在示例代码 11.10.1 中，主线程创建了 5 个新的线程，这 5 个线程使用同一个 start 函数 new_thread，该函数中定义了局部变量 number 和 tid 以及 arg 参数，意味着这 5 个线程的线程栈中都各自为这些变量分配了内存空间，任何一个线程修改了 number 或 tid 都不会影响其它线程。

示例代码 11.10.1 线程栈示例

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

```
#include <pthread.h>

static void *new_thread(void *arg)
{
    int number = *((int *)arg);
    unsigned long int tid = pthread_self();
    printf("当前为<%d>号线程，线程 ID<%lu>\n", number, tid);
    return (void *)0;
}

static int nums[5] = {0, 1, 2, 3, 4};

int main(int argc, char *argv[])
{
    pthread_t tid[5];
    int j;

    /* 创建 5 个线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j], NULL, new_thread, &nums[j]);

    /* 等待线程结束 */
    for (j = 0; j < 5; j++)
        pthread_join(tid[j], NULL);//回收线程

    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
当前为<1>号线程，线程ID<139672255092480>
当前为<2>号线程，线程ID<139672246699776>
当前为<4>号线程，线程ID<139672229914368>
当前为<0>号线程，线程ID<139672221521664>
当前为<3>号线程，线程ID<139672238307072>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.1 测试结果

11.10.2 可重入函数

要解释可重入（Reentrant）函数为何物，首先需要区分单线程程序和多线程程序。本章开头部分已向各位读者进行了详细介绍，单线程程序只有一条执行流（一个线程就是一条执行流），贯穿程序始终；而对于多线程程序而言，同一进程却存在多条独立、并发的执行流。

进程中执行流的数量除了与线程有关之外，与信号处理也有关联。因为信号是异步的，进程可能会在其运行过程中的任何时间点收到信号，进而跳转、执行信号处理函数，从而在一个单线程进程（包含信号处理）中形成了两条（即主程序和信号处理函数）独立的执行流。

接下来再来介绍什么是可重入函数，如果一个函数被同一进程的多个不同的执行流同时调用，每次函数调用总是能产生正确的结果（或者叫产生预期的结果），把这样的函数就称为可重入函数。

Tips：上面所说的同时指的是宏观上同时调用，实质上也就是该函数被多个执行流并发/并行调用，无特别说明，本章内容所提到的同时均指宏观上的概念。

重入指的是同一个函数被不同执行流调用，前一个执行流还没有执行完该函数、另一个执行流又开始调用该函数了，其实就是同一个函数被多个执行流并发/并行调用，在宏观角度上理解指的就是被多个执行流同时调用。

看到这里大家可能会有点不解，我们使用示例进行讲解。示例代码 11.10.2 是一个单线程与信号处理关联的程序。main()函数中调用 signal()函数为 SIGINT 信号注册了一个信号处理函数 sig_handler，信号处理函数 sig_handler 会调用 func 函数；main()函数最终会进入到一个循环中，循环调用 func()。

示例代码 11.10.2 信号与可重入问题

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

static void func(void)
{
    /*..... */
}

static void sig_handler(int sig)
{
    func();
}

int main(int argc, char *argv[])
{
    sig_t ret = NULL;

    ret = signal(SIGINT, (sig_t)sig_handler);
    if (SIG_ERR == ret) {
        perror("signal error");
        exit(-1);
    }

    /* 死循环 */
    for ( ; ; )
        func();

    exit(0);
```

```
}
```

当 main()函数正在执行 func()函数代码，此时进程收到了 SIGINT 信号，便会打断当前正常执行流程、跳转到 sig_handler()函数执行，进而调用 func、执行 func()函数代码；这里就出现了主程序与信号处理函数并发调用 func()的情况，示意图如下所示：

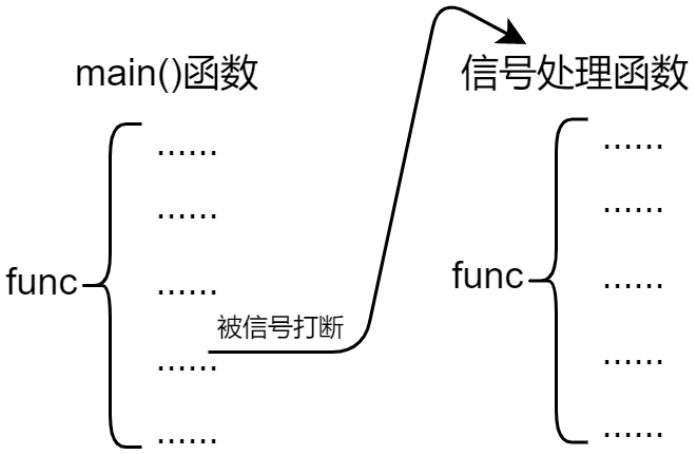


图 11.10.2 被信号打断

在信号处理函数中，执行完 func()之后，信号处理函数退出、返回到主程序流程，也就是被信号打断的位置处继续运行。如果每次出现这种情况执行 func()函数都能产生正确的结果，那么 func()函数就是一个可重入函数。

接着再来看看在多线程环境下，示例代码 11.10.1 是一个多线程程序，主线程调用 pthread_create()函数创建了 5 个新的线程，这 5 个线程使用同一个入口函数 new_thread；所以它们执行的代码是一样的，除了参数 arg 不同之外；在这种情况下，这 5 个线程中的多个线程就可能会出现并发调用 pthread_self()函数的情况。

以上举例说明了函数被多个执行流同时调用的两种情况：

- 在一个含有信号处理的程序当中，主程序正执行函数 func()，此时进程接收到信号，主程序被打断，跳转到信号处理函数中执行，信号处理函数中也调用了 func()。
- 在多线程环境下，多个线程并发调用同一个函数。

所以由此可知，在多线程环境以及信号处理有关应用程序中，需要注意不可重入函数的问题，如果多条执行流同时调用一个不可重入函数则可能会得不到预期的结果、甚至有可能导致程序崩溃！不止是在应用程序中，在一个包含了中断处理的裸机应用程序中亦是如此！所以不可重入函数通常存在着一定的安全隐患。

可重入函数的分类

笔者认为可重入函数可以分为两类：

- **绝对的可重入函数：**所谓绝对，指的是该函数不管如何调用，都刚断言它是可重入的，都能得到预期的结果。
- **带条件的可重入函数：**指的是在满足某个/某些条件的情况下，可以断言该函数是可重入的，不管怎么调用都能得到预期的结果。

绝对可重入函数

笔者查阅过很多的书籍以及网络文章，并未发现有提出过这种分类，所以这完全是笔者个人对此的一个理解，首先来看一下绝对可重入函数的一个例子，如下所示：

函数 func()就是一个标准的绝对可重入函数：

```
static int func(int a)
```

```
{
    int local;
    int j;

    for (local = 0, j = 0; j < 5; j++) {
        local += a * a;
        a += 2;
    }

    return local;
}
```

该函数内操作的变量均是函数内部定义的自动变量（局部变量），每次调用函数，都会在栈内存空间为局部变量分配内存，当函数调用结束返回时、再由系统回收这些变量占用的栈内存，所以局部变量生命周期只限于函数执行期间。

除此之外，该函数的参数和返回值均是值类型、而并非是引用类型（就是指针）。

如果多条执行流同时调用函数 func()，那必然会在栈空间中存在多份局部变量，每条执行流操作各自的局部变量，相互不影响，所以即使函数同时被调用，依然每次都能得到正确的结果。所以上面列举的函数 func()就是一个非常标准的绝对可重入函数，函数内部仅操作了函数内定义的局部变量，除了使用栈上的变量以外不依赖于任何环境变量，这样的函数就是 purecode（纯代码）可重入，可以允许该函数的多个副本同时在运行，由于它们使用的是分离的栈，所以不会相互干扰！

总结下绝对可重入函数的特点：

- 函数内所使用到的变量均为局部变量，换句话说，该函数内的操作的内存地址均为本地栈地址；
- 函数参数和返回值均是值类型；
- 函数内调用的其它函数也均是绝对可重入函数。

带条件的可重入函数

带条件的可重入函数通常需要满足一定的条件时才是可重入函数，我们来看一个不可重入函数的例子，如下所示：

```
static int glob = 0;

static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        local = glob;
        local++;
        glob = local;
    }
}
```

当多个执行流同时调用该函数，全局变量 glob 的最终值将不得而知，最终可能会得不到正确的结果，因为全局变量 glob 将成为多个线程间的共享数据，它们都会对 glob 变量进行读写操作、会导致数据不一致

的问题，关于这个问题在 12.1 小节中给大家做了详细说明。这个函数就是典型的不可重入函数，函数运行需要读取、修改全局变量 glob，该变量并非在函数自己的栈上，意味着该函数运行依赖于外部环境变量。

但如果对上面的函数进行修改，函数 func()内仅读取全局变量 glob 的值，而不更改它的值：

```
static int glob = 0;

static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        local = glob;
        local++;
        printf("local=%d\n", local);
    }
}
```

修改完之后，函数 func()内仅读取了变量 glob，而并未更改 glob 的值，那么此时函数 func()就是一个可重入函数了；但是这里需要注意，它需要满足一个条件，这个条件就是：当多个执行流同时调用函数 func()时，全局变量 glob 的值绝对不会在其它某个地方被更改；譬如线程 1 和线程 2 同时调用了函数 func()，但是另一个线程 3 在线程 1 和线程 2 同时调用了函数 func()的时候，可能会发生更改变量 glob 值的情况，如果是这样，那么函数 func()依然是不可重入函数。这就是有条件的可重入函数的概念，这通常需要程序员本身去规避这类问题，标准 C 语言函数库中也存在很多这类带条件的可重入函数，后面给大家看一下。

再来看一个例子：

```
static void func(int *arg)
{
    int local = *arg;
    int j;

    for (j = 0; j < 10; j++)
        local++;

    *arg = local;
}
```

这是一个参数为引用类型的函数，传入了一个指针，并在函数内部读写该指针所指向的内存地址，该函数是一个可重入函数，但同样需要满足一定的条件；如果多个执行流同时调用该函数时，所传入的指针是共享变量的地址，那么在这种情况，最终可能得不到预期的结果；因为在这种情况下，函数 func()所读写的便是多个执行流的共享数据，会出现数据不一致的情况，所以是不安全的。

但如果每个执行流所传入的指针是其本地变量（局部变量）对应的地址，那就是没有问题的，所以呢，这个函数就是一个带条件的可重入函数。

总结

相信笔者列举了这么多例子，大家应该明白了什么是可重入函数以及绝对可重入函数和带条件的可重入函数的区别，还有很多的例子这里就不再一一列举了，相信通过笔者的介绍大家应该知道如何去判断它们了。

很多的 C 库函数有两个版本：可重入版本和不可重入版本，可重入版本函数其名称后面加上了“_r”，用于表明该函数是一个可重入函数；而不可重入版本函数其名称后面没有“_r”，前面章节内容中也已经遇到过很多次了，譬如 asctime()/asctime_r()、ctime()/ctime_r()、localtime()/localtime_r()等。

通过 man 手册可以查询到它们“ATTRIBUTES”信息，譬如执行"man 3 ctime"，在帮助页面上往下翻便可以找到，如下所示：

ATTRIBUTES
For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
asctime()	Thread safety	MT-Unsafe race:asctime locale
asctime_r()	Thread safety	MT-Safe locale
ctime()	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale
ctime_r(), gmtime_r(), localtime_r(), mktime()	Thread safety	MT-Safe env locale
gmtime(), localtime()	Thread safety	MT-Unsafe race:tmbuf env locale

图 11.10.3 asctime()/asctime_r()函数的 ATTRIBUTES 信息

可以看到上图中有些函数 Value 这栏会显示 MT-Unsafe、而有些函数显示的却是 MT-Safe。MT 指的是 multithreaded（多线程），所以 MT-Unsafe 就是多线程不安全、MT-Safe 指的是多线程安全，通常习惯上将 MT-Safe 和 MT-Unsafe 称为线程安全或线程不安全。

Value 值为 MT-Safe 修饰的函数表示该函数是一个线程安全函数，使用 MT-Unsafe 修饰的函数表示它是一个线程不安全函数，下一小节会给大家介绍什么是线程安全函数。从上图可以看出，asctime_r()/ctime_r()/gmtime_r()/localtime_r()这些可重入函数都是线程安全函数，但这些函数都是带条件的可重入函数，可以发现在 MT-Safe 标签后面会携带诸如 env 或 locale 之类的标签，这其实就表示该函数需要在满足 env 或 locale 条件的情况下才是可重入函数；如果是绝对可重入函数，MT-Safe 标签后面不会携带任何标签，譬如数学库函数 sqrt：

ATTRIBUTES
For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
sqrt(), sqrtf(), sqrtl()	Thread safety	MT-Safe

图 11.10.4 sqrt 函数的 ATTRIBUTES 信息

诸如 env 或 locale 等标签，可以通过 man 手册进行查询，命令为"man 7 attributes"，这文档里边的内容反正笔者是没太看懂，不知所云；但是经过我的对比 env 或 locale 这两个标签还是很容易理解的。这两个标签在 man 测试里边出现的频率相对于其它的标签要大，这里笔者就简单地提一下：

- env：这个标签指的是该函数内部会读取进程的某个/某些环境变量，譬如 getenv()函数，前面也给大家介绍过，进程的环境变量其实就是程序的一个全局变量，前面也讲了，对于这类读取（但没更改）了全局变量的可重入函数应该要满足的条件，这里就不再重述了；
- local：local 指的是本地，很容易理解，通常该类函数传入了指针，前面也提到了传入了指针的可重入函数应该要满足什么样的条件才是可重入的，这里也不再重述！

本小节内容写得有点多了，笔者觉得讲的是比较清楚了，下小节给大家介绍线程安全函数。

11.10.3 线程安全函数

了解了可重入函数之后，再来看看线程安全函数。

一个函数被多个线程（其实也是多个执行流，但是不包括由信号处理函数所产生的执行流）同时调用时，它总会一直产生正确的结果，把这样的函数称为线程安全函数。线程安全函数包括可重入函数，可重入函数是线程安全函数的一个真子集，也就是说可重入函数一定是线程安全函数，但线程安全函数不一定是可重入函数，它们之间的关系如下：

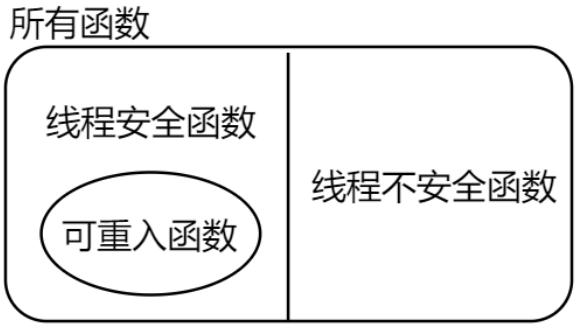


图 11.10.5 线程安全函数与可重入函数

譬如下面这个函数是一个不可重入函数，同样也是一个线程不安全函数（上小节的最后一个例子）：

```
static int glob = 0;

static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        local = glob;
        local++;
        glob = local;
    }
}
```

如果对该函数进行修改，使用线程同步技术（譬如互斥锁）对共享变量 glob 的访问进行保护，在读写该变量之前先上锁、读写完成之后在解锁。这样，该函数就变成了一个线程安全函数，但是它依然不是可重入函数，因为该函数更改了外部全局变量的值。

可重入函数只是单纯从语言语法角度分析它的可重入性质，不涉及到一些具体的实现机制，譬如线程同步技术，这是判断可重入函数和线程安全函数的区别，因为你单从概念上去分析的话，其实可以发现可重入函数和线程安全函数好像说的是同一个东西，“一个函数被多个线程同时调用时，它总会一直产生正确的结果，把这样的函数称为线程安全函数”，多个线程指的就是多个执行流（不包括信号处理函数执行流），所以从这里看跟可重入函数的概念是很相似的。

判断一个函数是否为线程安全函数的方法是，该函数被多个线程同时调用是否总能产生正确的结果，如果每次都能产生预期的结果则表示该函数是一个线程安全函数。判读一个函数是否为可重入函数的方法是，从语言语法角度分析，该函数被多个执行流同时调用是否总能产生正确的结果，如果每次都能产生预期的结果则表示该函数是一个可重入函数。

POSIX.1-2001 和 POSIX.1-2008 标准中规定的所有函数都必须是线程安全函数，但以下函数除外：

asctime()	basename()	catgets()	crypt()
ctermid()	ctime()	dbm_clearerr()	dbm_close()
dbm_delete()	dbm_error()	dbm_fetch()	dbm_firstkey()
dbm_nextkey()	dbm_open()	dbm_store()	dirname()
dlerror()	drand48()	ecvt()	encrypt()
endgrent()	endpwent()	endutxent()	fcvt()
ftw()	gcvt()	getc_unlocked()	getchar_unlocked()
getdate()	getenv()	getgrent()	getgrgid()
getgrnam()	gethostbyaddr()	gethostbyname()	gethostent()
getlogin()	getnetbyaddr()	getnetbyname()	getnetent()
getopt()	getprotobyname()	getprotobynumber()	getprotoent()
getpwent()	getpwnam()	getpwuid()	getservbyname()
getservbyport()	getservent()	getutxent()	getutxid()
getutxline()	gmtime()	hcreate()	hdestroy()
hsearch()	inet_ntoa()	l64a()	lgamma()
lgammaf()	lgammal()	localeconv()	localtime()
lrand48()	mrand48()	nftw()	nl_langinfo()
ptsname()	putc_unlocked()	putchar_unlocked()	putenv()
pututxline()	rand()	readdir()	setenv()
setgrent()	setkey()	setpwent()	setutxent()
strerror()	strsignal()	strtok()	system()
tmpnam()	ttyname()	unsetenv()	wcrtomb()
wcsrtombs()	wcstombs()	wctomb()	

表 11.10.1 POSIX.1-2001 和 POSIX.1-2008 中列出的线程不安全函数

以上所列举出的这些函数被认为是线程不安全函数，大家也可以通过 man 手册查询到这些函数，"man 7 pthreads"，如下所示：

图 11.10.6 通过 man 手册查询到线程不安全函数

如果想确认某个函数是不是线程安全函数可以

上小节给大家提到过，man 手册可以查看库函数的 ATTRIBUTES 信息，如果函数被标记为 MT-Safe，则表示该函数是一个线程安全函数，如果被标记为 MT-Unsafe，则意味着该函数是一个非线程安全函数，对

于非线程安全函数，在多线程编程环境下尤其要注意，如果某函数可能会被多个线程同时调用时，该函数不能是非线程安全函数，一定要是线程安全函数，否则将会出现意想不到的结果、甚至使得整个程序崩溃！

对于一个中大型的多线程应用程序项目来说，能够保证整个程序的安全性，这是非常重要的，程序员必须要正确对待线程安全以及信号处理等这类在多线程环境下敏感的问题，这通常对程序员提出了更高的要求。

11.10.4 一次性初始化

在多线程编程环境下，有些代码段只需要执行一次，譬如一些初始化相关的代码段，通常比较容易想到的就是将其放在 main()主函数进行初始化，这样也就是意味着该段代码只在主线程中被调用，只执行过一次。大家想一下这样的问题：当你写了一个 C 函数 func()，该函数可能会被多个线程调用，并且该函数中有一段初始化代码，该段代码只能被执行一次（无论哪个线程执行都可以）、如果执行多次会出现问题，如下所示：

```
static void func(void)
{
    /* 只能执行一次的代码段 */
    init_once();
    /************************/


    .....
    .....
}
```

大家可能会问，怎么会有这样的需求呢？当然有，譬如下小节将要介绍的线程特有数据就需要有这样的需求，那我们如何去保证这段代码只能被执行一次呢（被进程中的任一线程执行都可以）？本小节向大家介绍 pthread_once()函数，该函数原型如下所示：

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

在多线程编程环境下，尽管 pthread_once()调用会出现在多个线程中，但该函数会保证 init_routine()函数仅执行一次，究竟在哪个线程中执行是不定的，是由内核调度来决定。函数参数和返回值含义如下：

once_control：这是一个 pthread_once_t 类型指针，在调用 pthread_once()函数之前，我们需要定义了一个 pthread_once_t 类型的静态变量，调用 pthread_once()时参数 once_control 指向该变量。通常在定义变量时会使用 PTHREAD_ONCE_INIT 宏对其进行初始化，譬如：

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

init_routine：一个函数指针，参数 init_routine 所指向的函数就是要求只能被执行一次的代码段，pthread_once()函数内部会调用 init_routine()，即使 pthread_once()函数会被多次执行，但它能保证 init_routine()仅被执行一次。

返回值：调用成功返回 0；失败则返回错误编码以指示错误原因。

如果参数 once_control 指向的 pthread_once_t 类型变量，其初值不是 PTHREAD_ONCE_INIT，pthread_once()的行为将是不正常的；PTHREAD_ONCE_INIT 宏在<pthread.h>头文件中定义。

如果在一个线程调用 pthread_once()时，另外一个线程也调用了 pthread_once，则该线程将会被阻塞等待，直到第一个完成初始化后返回。换言之，当调用 pthread_once 成功返回时，调用总是能够肯定所有的状态已经初始化完成了。

使用示例

接下来我们测试下，当 pthread_once()被多次调用时，init_routine()函数是不是只会被执行一次，示例代码如下所示：

示例代码 11.10.3 pthread_once()函数使用示例

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

static pthread_once_t once = PTHREAD_ONCE_INIT;

static void initialize_once(void)
{
    printf("initialize_once 被执行: 线程 ID<%lu>\n", pthread_self());
}

static void func(void)
{
    pthread_once(&once, initialize_once);//执行一次性初始化函数
    printf("函数 func 执行完毕.\n");
}

static void *thread_start(void *arg)
{
    printf("线程%d 被创建: 线程 ID<%lu>\n", *((int *)arg), pthread_self());
    func();        //调用函数 func
    pthread_exit(NULL); //线程终止
}

static int nums[5] = {0, 1, 2, 3, 4};

int main(void)
{
    pthread_t tid[5];
    int j;

    /* 创建 5 个线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j], NULL, thread_start, &nums[j]);


    /* 等待线程结束 */
    for (j = 0; j < 5; j++)
        pthread_join(tid[j], NULL);//回收线程
```

```
    exit(0);
}
```

程序中调用 pthread_create()创建了 5 个子线程，新线程的入口函数均为 thread_start()，thread_start()函数会调用 func()，并在 func()函数调用 pthread_once()，需要执行的一次性初始化函数为 initialize_once()，换言之，pthread_once()函数会被执行 5 次，每个子线程各自执行一次。

编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
线程1被创建：线程ID<140621723465472>
initialize once被执行：线程ID<140621723465472>
线程3被创建：线程ID<140621706680064>
函数func执行完毕.
线程0被创建：线程ID<140621698287360>
函数func执行完毕.
线程2被创建：线程ID<140621715072768>
函数func执行完毕.
线程0被创建：线程ID<140621689894656>
函数func执行完毕.
函数func执行完毕.
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.7 测试结果

从打印信息可知，initialize_once()函数确实只被执行了一次，也就是被编号为 1 的线程所执行，其它线程均未执行该函数。

11.10.5 线程特有数据

线程特有数据也称为线程私有数据，简单点说，就是为每个调用线程分别维护一份变量的副本（copy），每个线程通过特有数据键（key）访问时，这个特有数据键都会获取到本线程绑定的变量副本。这样就可以避免变量成为多个线程间的共享数据。

C 库中有很多函数都是非线程安全函数，非线程安全函数在多线程环境下，被多个线程同时调用时将会发生意想不到的结果，得不到预期的结果。譬如很多库函数都会返回一个字符串指针，譬如 asctime()、ctime()、localtime()等，返回出来的字符串可以被调用线程直接使用，但该字符串缓冲区通常是这些函数内部所维护的静态数组或者是某个全局数组（这里笔者只是猜测，具体是哪一种我也不清楚，没有翻看这些函数内部的实现）。

既然如此，多次调用这些函数返回的字符串其实指向的是同一个缓冲区，每次调用都会刷新缓冲区中的数据。这些函数是非线程安全的，譬如当 ctime()被多个线程同时调用时，返回的字符串中的数据可能是混乱的，因为某一线程调用它时，缓冲区中的数据可能被另一个调用线程修改了。针对这些非线程安全函数，可以使用线程特有数据将其变为线程安全函数，线程特有数据通常会在编写一些库函数的时使用到，后面我们会演示如何使用线程特有数据。

线程特有数据的核心思想其实非常简单，就是为每一个调用线程（调用某函数的线程，该函数就是我们要通过线程特有数据将其实现为线程安全的函数）分配属于该线程的私有数据区，为每个调用线程分别维护一份变量的副本。

线程特有数据主要涉及到 3 个函数：pthread_key_create()、pthread_setspecific()以及 pthread_getspecific()，接下来一一向大家进行介绍。

pthread_key_create()函数

在为线程分配私有数据区之前，需要调用 pthread_key_create()函数创建一个特有数据键（key），并且只需要在首个调用的线程中创建一次即可，所以通常会使用到上小节所学习的 pthread_once()函数。pthread_key_create()函数原型如下所示：

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

使用该函数需要包含头文件<pthread.h>。

函数参数和返回值含义如下：

key：调用该函数会创建一个特有数据键，并通过参数 key 所指向的缓冲区返回给调用者，参数 key 是一个 pthread_key_t 类型的指针，可以把 pthread_key_t 称为 key 类型。调用 pthread_key_create()之前，需要定义一个 pthread_key_t 类型变量，调用 pthread_key_create()时参数 key 指向 pthread_key_t 类型变量。

destructor：参数 destructor 是一个函数指针，指向一个自定义的函数，其格式如下：

```
void destructor(void *value)
{
    /* code */
}
```

调用 pthread_key_create()函数允许调用者指定一个自定义的解构函数（类似于 C++中的析构函数），使用参数 destructor 指向该函数；该函数通常用于释放与特有数据键关联的线程私有数据区占用的内存空间，当使用线程特有数据的线程终止时，destructor()函数会被自动调用。

返回值：成功返回 0；失败将返回一个错误编号以指示错误原因，返回的错误编号其实就是全局变量 errno，可以使用诸如 strerror()函数查看其错误字符串信息。

pthread_setspecific()函数

调用 pthread_key_create()函数创建特有数据键（key）后通常需要为调用线程分配私有数据缓冲区，譬如通过 malloc()（或类似函数）申请堆内存，每个调用线程分配一次，且只会在线程初次调用此函数时分配。为线程分配私有数据缓冲区之后，通常需要调用 pthread_setspecific()函数，pthread_setspecific()函数其实完成了这样的操作：首先保存指向线程私有数据缓冲区的指针，并将其与特有数据键以及当前调用线程关联起来；其函数原型如下所示：

```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
```

函数参数和返回值含义如下：

key：pthread_key_t 类型变量，参数 key 应赋值为调用 pthread_key_create()函数时创建的特有数据键，也就是 pthread_key_create()函数的参数 key 所指向的 pthread_key_t 变量。

value：参数 value 是一个 void 类型的指针，指向由调用者分配的一块内存，作为线程的私有数据缓冲区，当线程终止时，会自动调用参数 key 指定的特有数据键对应的解构函数来释放这一块动态申请的内存空间。

返回值：调用成功返回 0；失败将返回一个错误编码，可以使用诸如 strerror()函数查看其错误字符串信息。

pthread_getspecific()函数

调用 pthread_setspecific()函数将线程私有数据缓冲区与调用线程以及特有数据键关联之后，便可以使用 pthread_getspecific()函数来获取调用线程的私有数据区了。其函数原型如下所示：

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

```
void *pthread_getspecific(pthread_key_t key);
```

参数 key 应赋值为调用 pthread_key_create()函数时创建的特有数据键，也就是 pthread_key_create()函数的参数 key 指向的 pthread_key_t 变量。

pthread_getspecific()函数应返回当前调用线程关联到特有数据键的私有数据缓冲区，返回值是一个指针，指向该缓冲区。如果当前调用线程并没有设置线程私有数据缓冲区与特有数据键进行关联，则返回值应为 NULL，函数中可以利用这一点来判断当前调用线程是否为初次调用该函数，如果是初次调用，则必须为该线程分配私有数据缓冲区。

pthread_key_delete()函数

除了以上介绍的三个函数外，如果需要删除一个特有数据键（key）可以使用函数 pthread_key_delete()，pthread_key_delete()函数删除先前由 pthread_key_create()创建的键。其函数原型如下所示：

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

参数 key 为要删除的键。函数调用成功返回 0，失败将返回一个错误编号。

调用 pthread_key_delete()函数将释放参数 key 指定的特有数据键，可以供下一次调用 pthread_key_create()时使用；调用 pthread_key_delete()时，它并不将查当前是否有线程正在使用该键所关联的线程私有数据缓冲区，所以它并不会触发键的解构函数，也就不会释放键关联的线程私有数据区占用的内存资源，并且调用 pthread_key_delete()后，当线程终止时也不再执行键的解构函数。所以，通常在调用 pthread_key_delete()之前，必须确保以下条件：

- 所有线程已经释放了私有数据区（显式调用解构函数或线程终止）。
- 参数 key 指定的特有数据键将不再使用。

任何在调用 pthread_key_delete()之后使用键的操作都会导致未定义的行为，譬如调用 pthread_setspecific()或 pthread_getspecific()将会以错误形式返回。

使用示例

接下来编写一个使用线程特有数据的例子，很多书籍上都会使用 strerror()函数作为例子，这个函数曾在 3.2 小节向大家介绍过，通过 man 手册查询到 strerror()函数是一个非线程安全函数，其实它有对应的可重入版本 strerror_r()，可重入版本 strerror_r()函数则是一个线程安全函数。

这里暂且不管 strerror_r()函数，我们来聊一聊 strerror()函数，函数内部的实现方式，这里简单地提一下：调用 strerror()函数，需要传入一个错误编号，错误编号赋值给参数 errnum，在 Linux 系统中，每一个错误编号都会对应一个字符串，用于描述该错误，strerror()函数会根据传入的 errnum 找到对应的字符串，返回指向该字符串的指针。

事实上，在 Linux 的实现中，标准 C 语言函数库（glibc）提供的 strerror()函数是线程安全的，但在 man 手册中记录它是一个非线程安全函数，笔者猜测可能在某些操作系统的 C 语言函数库实现中，该函数是非线程安全函数的；但在 glibc 库中，它确实是线程安全函数，为此笔者还特意去查看了 glibc 库中 strerror 函数的源码，证实了这一点，这里大家一定要注意。

以下是 strerror()函数以非线程安全方式实现的一种写法（具体的写法不止这一种，这里只是以此为例）：

示例代码 11.10.4 strerror()函数以非线程安全方式实现的一种写法

```
#define _GNU_SOURCE

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

```
#define MAX_ERROR_LEN 256
static char buf[MAX_ERROR_LEN];

static char *strerror(int errnum)
{
    if (errnum < 0 || errnum >= _sys_nerr || NULL == _sys_errlist[errnum])
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errnum);
    else {
        strncpy(buf, _sys_errlist[errnum], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';//终止字符
    }

    return buf;
}
```

再次说明，glibc 库中 strerror()是线程安全函数，本文为了向大家介绍/使用线程特有数据，以非线程安全方式实现了 strerror()函数。

首先在源码中需要定义_GNU_SOURCE 宏，_GNU_SOURCE 宏在前面章节已有介绍，这里不再重述！源码中需要定义_GNU_SOURCE 宏，不然编译源码将会提示_sys_nerr 和_sys_errlist 找不到。该函数利用了 glibc 定义的一对全局变量：_sys_errlist 是一个指针数组，其中的每一个元素指向一个与 errno 错误编号相匹配的描述性字符串；_sys_nerr 表示_sys_errlist 数组中元素的个数。

可以看到该函数返回的字符串指针，其实是一个静态数组，当多个线程同时调用该函数时，那么 buf 缓冲区中的数据将会出现混乱，因为前一个调用线程拷贝到 buf 中的数据可能会被后一个调用线程重写覆盖等情况。

对此，我们可以对示例代码 11.10.4 进行测试，让多个线程都调用它，看看测试结果，测试代码如下：

示例代码 11.10.5 非线程安全版 strerror 测试

```
#define _GNU_SOURCE

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

#define MAX_ERROR_LEN 256
static char buf[MAX_ERROR_LEN];

/**********************************
 * 为了避免与库函数 strerror 重名
 * 这里将其改成 my_strerror
 **********************************/
static char *my_strerror(int errnum)
{
    if (errnum < 0 || errnum >= _sys_nerr || NULL == _sys_errlist[errnum])
```

```
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errnum);
    else {
        strncpy(buf, _sys_errlist[errnum], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';//终止字符
    }

    return buf;
}

static void *thread_start(void *arg)
{
    char *str = my_strerror(2); //获取错误编号为 2 的错误描述信息
    printf("子线程: str (%p) = %s\n", str, str);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    char *str = NULL;
    int ret;

    str = my_strerror(1);     //获取错误编号为 1 的错误描述信息

    /* 创建子线程 */
    if (ret = pthread_create(&tid, NULL, thread_start, NULL)) {
        fprintf(stderr, "pthread_create error: %d\n", ret);
        exit(-1);
    }

    /* 等待回收子线程 */
    if (ret = pthread_join(tid, NULL)) {
        fprintf(stderr, "pthread_join error: %d\n", ret);
        exit(-1);
    }

    printf("主线程: str (%p) = %s\n", str, str);
    exit(0);
}
```

主线程首先调用 my_strerror()获取到了编号为 1 的错误描述信息，接着创建了一个子线程，在子线程中调用 my_strerror()获取编号为 2 的错误描述信息，并将其打印出来，包括字符串的地址值；子线程结束后，主线程也打印了之前获取到的错误描述信息。我们想看到的结果是，主线程和子线程打印的错误描述信息是不一样的，因为错误编号不同，但上面的测试结果证实它们打印的结果是相同的：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子线程: str (0x601500) = No such file or directory
主线程: str (0x601500) = No such file or directory
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.8 测试结果

从以上测试结果可知，子线程和主线程锁获取到的错误描述信息是相同的，字符串指针指向的是同一个缓冲区；原因就在于，my_strerror()函数是一个非线程安全函数，函数内部修改了全局静态变量、并返回了它的指针，每一次调用访问的都是同一个静态变量，所以后一次调用会覆盖掉前一次调用的结果。

接下来我们使用本小节所介绍的线程特有数据技术对示例代码 11.10.4 中 strerror()函数进行修改，如下所示：

示例代码 11.10.6 使用线程特有数据实现线程安全的 strerror()函数

```
#define _GNU_SOURCE

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

#define MAX_ERROR_LEN 256

static pthread_once_t once = PTHREAD_ONCE_INIT;
static pthread_key_t strerror_key;

static void destructor(void *buf)
{
    free(buf);   //释放内存
}

static void create_key(void)
{
    /* 创建一个键(key)，并且绑定键的解构函数 */
    if (pthread_key_create(&strerror_key, destructor))
        pthread_exit(NULL);
}

/*******************************
 * 对 strerror 函数重写
 * 使其变成为一个线程安全函数
 *******************************/
static char *strerror(int errnum)
```

```
{
    char *buf;

    /* 创建一个键(只执行一次 create_key) */
    if (pthread_once(&once, create_key))
        pthread_exit(NULL);

    /* 获取 */
    buf = pthread_getspecific(strerror_key);
    if (NULL == buf) {   //首次调用 my_strerror 函数，则需给调用线程分配线程私有数据
        buf = malloc(MAX_ERROR_LEN);//分配内存
        if (NULL == buf)
            pthread_exit(NULL);

        /* 保存缓冲区地址,与键、线程关联起来 */
        if (pthread_setspecific(strerror_key, buf))
            pthread_exit(NULL);
    }

    if (errnum < 0 || errnum >= _sys_nerr || NULL == _sys_errlist[errnum])
        snprintf(buf, MAX_ERROR_LEN, "Unknown error %d", errnum);
    else {
        strncpy(buf, _sys_errlist[errnum], MAX_ERROR_LEN - 1);
        buf[MAX_ERROR_LEN - 1] = '\0';//终止字符
    }

    return buf;
}
```

改进版的 strerror()所做的第一步是调用 pthread_once()，以确保只会执行一次 create_key()函数，而在 create_key()函数中便是调用 pthread_key_create()创建了一个键、并绑定了相应的解构函数 destructor()，解构函数用于释放与键关联的所有线程私有数据所占的内存空间。

接着，函数 strerror()调用 pthread_getspecific()以获取该调用线程与键相关联的私有数据缓冲区地址，如果返回为 NULL，则表明该线程是首次调用 strerror()函数，因为函数会调用 malloc()为其分配一个新的私有数据缓冲区，并调用 pthread_setspecific()来保存缓冲区地址、并与键以及该调用线程建立关联。如果 pthread_getspecific()函数的返回值并不等于 NULL，那么该值将指向以存在的私有数据缓冲区，此缓冲区由之前对 strerror()的调用所分配。

剩余部分代码与示例代码 11.10.4 非线程安全版的 strerror()实现类似，唯一的区别在于，buf 是线程特有数据的缓冲区地址，而非全局的静态变量。

改进版的 strerror 就是一个线程安全函数，编写一个线程安全函数当然要保证该函数中调用的其它函数也必须是线程安全的，那如何确认自己调用的函数是线程安全函数呢？其实非常简单，前面也给大家介绍过，譬如通过 man 手册查看函数的 ATTRIBUTES 描述信息，或者查看 man 手册中记录的非线程安全函数列表（执行"man 7 pthreads"命令查看）、进行对比。

Tips：有时会发现 ATTRIBUTES 描述信息与非线程安全函数列表不一致，譬如 ATTRIBUTES 描述信息中显示该函数是 MT-Unsafe（非线程安全函数）标识的，但是却没记录在非线程安全函数列表中，此时我们应该以列表为准！默认该函数是线程安全的。

大家可以去测试下改进版的 strerror，这里笔者便不再给大家演示了，需要注意的是，在测试代码中定义的 strerror 函数其名字需要改成其它的名称，避免与库函数 strerror 重名。

11.10.6 线程局部存储

通常情况下，程序中定义的全局变量是进程中所有线程共享的，所有线程都可以访问这些全局变量；而线程局部存储在定义全局或静态变量时，使用__thread 修饰符修饰变量，此时，每个线程都会拥有一份对该变量的拷贝。线程局部存储中的变量将一直存在，直至线程终止，届时会自动释放这一存储。

线程局部存储的主要优点在于，比线程特有数据的使用要简单。要创建线程局部变量，只需简单地在全局或静态变量的声明中包含__thread 修饰符即可！譬如：

```
static __thread char buf[512];
```

但凡带有这种修饰符的变量，每个线程都拥有一份对变量的拷贝，意味着每个线程访问的都是该变量在本线程的副本，从而避免了全局变量成为多个线程的共享数据。

关于线程局部变量的声明和使用，需要注意以下几点：

- 如果变量声明中使用了关键字 static 或 extern，那么关键字__thread 必须紧随其后。
- 与一般的全局或静态变量申明一眼，线程局部变量在申明时可设置一个初始值。
- 可以使用 C 语言取值操作符（&）来获取线程局部变量的地址。

Tips：线程局部存储需要内核、Pthreads 以及 GCC 编译器的支持。

使用示例

我们编写一个简单的程序来测试线程局部存储，示例代码如下所示：

示例代码 11.10.7 线程局部存储测试

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

static __thread char buf[100];

static void *thread_start(void *arg)
{
    strcpy(buf, "Child Thread\n");
    printf("子线程: buf (%p) = %s", buf, buf);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    strcpy(buf, "Main Thread\n");
```

```
    /* 创建子线程 */
    if (ret = pthread_create(&tid, NULL, thread_start, NULL)) {
        fprintf(stderr, "pthread_create error: %d\n", ret);
        exit(-1);
    }

    /* 等待回收子线程 */
    if (ret = pthread_join(tid, NULL)) {
        fprintf(stderr, "pthread_join error: %d\n", ret);
        exit(-1);
    }

    printf("主线程: buf (%p) = %s", buf, buf);
    exit(0);
}
```

程序中定义了一个全局变量 buf，使用__thread 修饰，使其变为线程局部变量；主线程中首先调用 strcpy 拷贝了字符串到 buf 缓冲区中，随后创建了一个子线程，子线程也调用了 strcpy()向 buf 缓冲区拷贝了数据；并调用 printf 打印 buf 缓冲区存储的字符串以及 buf 缓冲区的指针值。

子线程终止后，主线程也打印 buf 缓冲区中存储的字符串以及 buf 缓冲区的指针值，运行结果如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
子线程: buf (0x7f87b6db369c) = Child Thread
主线程: buf (0x7f87b75a671c) = Main Thread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 11.10.9 测试结果

从地址便可以看出来，主线程和子线程中使用的 buf 绝不是同一个变量，这就是线程局部存储，使得每个线程都拥有一份对变量的拷贝，各个线程操作各自的变量不会影响其它线程。

大家可以使用线程局部存储方式对示例代码 11.10.4 strerror 函数进行修改，使其成为一个线程安全函数。

11.11 更多细节问题

本小节将对线程各方面的细节做深入讨论，其主要包括线程与信号之间牵扯的问题、线程与进程控制（fork()、exec()、exit()等）之间的交互。之所以出现了这些问题，其原因在于线程技术的问世晚于信号、进程控制等，然而线程的出现必须要能够兼容现有的这些技术，不能出现冲突，这就使得线程与它们之间的结合使用将会变得比较复杂！当中所涉及到的细节问题也会比较多。

11.11.1 线程与信号

Linux 信号模型是基于进程模型而设计的，信号的问世远早于线程；自然而然，线程与信号之间就会存在一些冲突，其主要原因在于：信号既要能够在传统的单线程进程中保持它原有的功能、特性，与此同时，又需要设计出能够适用于多线程环境的新特性！

信号与多线程模型之间结合使用，将会变得比较复杂，需要考虑的问题将会更多，在实际应用开发当中，如果能够避免我们应尽量避免此类事情的发生；但尽管如此，事实上，信号与多线程模型确实存在于实际的应用开发项目中。本小节我们就来讨论信号与线程之间牵扯的问题。

⑴、信号如何映射到线程

信号模型在一些方面是属于进程层面（由进程中的所有线程线程共享）的，而在另一些方面是属于单个线程层面的，以下对其进行汇总：

- 信号的系统默认行为是属于进程层面。8.3 小节介绍到，每一个信号都有其对应的系统默认动作，当进程中的任一线程收到任何一个未经处理（忽略或捕获）的信号时，会执行该信号的默认操作，信号的默认操作通常是停止或终止进程。
- 信号处理函数属于进程层面。进程中的所有线程共享程序中所注册的信号处理函数；
- 信号的发送既可针对整个进程，也可针对某个特定的线程。在满足以下三个条件中的任意一个时，信号的发送针对的是某个线程：
  - 产生了硬件异常相关信号，譬如 SIGBUS、SIGFPE、SIGILL 和 SIGSEGV 信号；这些硬件异常信号在某个线程执行指令的过程中产生，也就是说这些硬件异常信号是由某个线程所引起；那么在这种情况下，系统会将信号发送给该线程。
  - 当线程试图对已断开的管道进行写操作时所产生的 SIGPIPE 信号；
  - 由函数 pthread_kill()或 pthread_sigqueue()所发出的信号，稍后介绍这两个函数；这些函数允许线程向同一进程下的其它线程发送一个指定的信号。

  除了以上提到的三种情况之外，其它机制产生的信号均属于进程层面，譬如其它进程调用 kill()或 sigqueue()所发送的信号；用户在终端按下 Ctrl+C、Ctrl+\、Ctrl+Z 向前台进程发送的 SIGINT、SIGQUIT 以及 SIGTSTP 信号。
- 当一个多线程进程接收到一个信号时，且该信号绑定了信号处理函数时，内核会任选一个线程来接收这个信号，意味着由该线程接收信号并调用信号处理函数对其进行处理，并不是每个线程都会接收到该信号并调用信号处理函数；这种行为与信号的原始语义是保持一致的，让进程对单个信号接收重复处理多次是没有意义的。
- 信号掩码其实是属于线程层面的，也就是说信号掩码是针对每个线程而言。8.9 小节向大家介绍了信号掩码的概念，并介绍了 sigprocmask()函数，通过 sigprocmask()可以设置进程的信号掩码，事实上，信号掩码是并不是针对整个进程来说，而是针对线程，对于一个多线程应用程序来说，并不存在一个作用于整个进程范围内的信号掩码（管理进程中的所有线程）；那么在多线程环境下，各个线程可以调用 pthread_sigmask()函数来设置它们各自的信号掩码，譬如设置线程可以接收哪些信号、不接收哪些信号，各线程可独立阻止或放行各种信号。
- 针对整个进程所挂起的信号，以及针对每个线程所挂起的信号，内核都会分别进行维护、记录。8.11.1 小节介绍到，调用 sigpending()会返回进程中所有被挂起的信号，事实上，sigpending()会返回针对整个进程所挂起的信号，以及针对每个线程所挂起的信号的并集。

⑵、线程的信号掩码

对于一个单线程程序来说，使用 sigprocmask()函数设置进程的信号掩码，在多线程环境下，使用 pthread_sigmask()函数来设置各个线程的信号掩码，其函数原型如下所示：

```
#include <signal.h>
```

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

pthread_sigmask()函数就像 sigprocmask()一样，不同之处在于它在多线程程序中使用，所以 pthread_sigmask()函数的用法与 sigprocmask()完全一样，这里就不再重述！

每个刚创建的线程，会从其创建者处继承信号掩码，这个新的线程可以调用 pthread_sigmask()函数来改变它的信号掩码。

⑶、向线程发送信号

调用 kill()或 sigqueue()所发送的信号都是针对整个进程来说的，它属于进程层面，具体该目标进程中的哪一个线程会去处理信号，由内核进行选择。事实上，在多线程程序中，可以通过 pthread_kill()向同一进程中的某个指定线程发送信号，其函数原型如下所示：

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

参数 thread，也就是线程 ID，用于指定同一进程中的某个线程，调用 pthread_kill()将向参数 thread 指定的线程发送信号 sig。

如果参数 sig 为 0，则不发送信号，但仍会执行错误检查。函数调用成功返回 0，失败将返回一个错误编号，不会发送信号。

除了 pthread_kill()函数外，还可以调用 pthread_sigqueue()函数；pthread_sigqueue()函数执行与 sigqueue 类似的任务，但它不是向进程发送信号，而是向同一进程中的某个指定的线程发送信号。其函数原型如下所示：

```
#include <signal.h>
#include <pthread.h>

int pthread_sigqueue(pthread_t thread, int sig, const union sigval value);
```

参数 thread 为线程 ID，指定接收信号的目标线程（目标线程与调用 pthread_sigqueue()函数的线程是属于同一个进程），参数 sig 指定要发送的信号，参数 value 指定伴随数据，与 sigqueue()函数中的 value 参数意义相同。

pthread_sigqueue()函数的参数的含义与 sigqueue()函数中对应参数相同意义相同。它俩的唯一区别在于，sigqueue()函数发送的信号针对的是整个进程，而 pthread_sigqueue()函数发送的信号针对的是某个线程。

⑷、异步信号安全函数

应用程序中涉及信号处理函数时必须要非常小心，因为信号处理函数可能会在程序执行的任意时间点被调用，从而打断主程序。接下来介绍一个概念---异步信号安全函数（async-signal-safe function）。

前面介绍了线程安全函数，作为线程安全函数可以被多个线程同时调用，每次都能得到预期的结果，但是这里有前提条件，那就是没有信号处理函数参与；换句话说，线程安全函数不能在信号处理函数中被调用，否则就不能保证它一定是安全的。所以就出现了异步信号安全函数。

异步信号安全函数指的是可以在信号处理函数中可以被安全调用的线程安全函数，所以它比线程安全函数的要求更为严格！可重入函数满足这个要求，所以可重入函数一定是异步信号安全函数。而线程安全函数则不一定是异步信号安全函数了。

举个例子，下面列举出来的一个函数是线程安全函数：

```
static pthread_mutex_t mutex;
static int glob = 0;
```

```
static void func(int loops)
{
    int local;
    int j;

    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mutex); //互斥锁上锁

        local = glob;
        local++;
        glob = local;

        pthread_mutex_unlock(&mutex);//互斥锁解锁
    }
}
```

该函数虽然对全局变量进行读写操作，但是在访问全局变量时进行了加锁，避免了引发竞争冒险；它是一个线程安全函数，假设线程 1 正在执行函数 func，刚刚获得锁（也就是刚刚对互斥锁上锁），而这时进程收到信号，并分派给线程 1 处理，线程 1 接着跳转去执行信号处理函数，不巧的是，信号处理函数中也调用了 func()函数，同样它也去获取锁，由于此时锁处于锁住状态，所以信号处理函数中调用 func()获取锁将会陷入休眠、等待锁的释放。这时线程 1 就会陷入死锁状态，线程 1 无法执行，锁无法释放；如果其它线程也调用 func()，那它们也会陷入休眠、如此将会导致整个程序陷入死锁！

通过上面的分析，可知，涉及到信号处理函数时要非常小心。之所以涉及到信号处理函数时会出现安全问题，笔者认为主要原因在以下两个方面：

- 信号是异步的，信号可能会在任何时间点中断主程序的运行，跳转到信号处理函数处执行，从而形成一个新的执行流（信号处理函数执行流）。
- 信号处理函数执行流与线程执行流存在一些区别，信号处理函数所产生的执行流是由执行信号处理函数的线程所触发的，它俩是在同一个线程中，属于同一个线程执行流。

在异步信号安全函数、可重入函数以及线程安全函数三者中，可重入函数的要求是最严格的，所以通常会说可重入函数一定是线程安全函数、也一定是异步信号安全函数。通常对于上面所列举出的线程安全函数 func()，如果想将其实现为异步信号安全函数，可以在获取锁之前通过设置信号掩码，在锁期间禁止接收该信号，也就是说将函数实现为不可被信号中断。经过这样处理之后，函数 func()就是一个异步信号安全函数了。

Linux 标准 C 库和系统调用中以下函数被认为是异步信号安全函数：

_Exit()	_exit()	abort()	accept()
access()	aio_error()	aio_return()	aio_suspend()
alarm()	bind()	cfgetispeed()	cfgetospeed()
cfsetispeed()	cfsetospeed()	chdir()	chmod()
chown()	clock_gettime()	close()	connect()
creat()	dup()	dup2()	execle()
execve()	fchmod()	fchown()	fcntl()
fdatasync()	fork()	execl()	fstat()
fsync()	ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()	getpgrp()

getpid()	getppid()	getsockname()	getsockopt()
getuid()	kill()	link()	listen()
lseek()	lstat()	mkdir()	mkfifo()
open()	execv()	pause()	pipe()
poll()	posix_trace_event()	pselect()	raise()
read()	readlink()	recv()	recvfrom()
recvmsg()	rename()	rmdir()	select()
sem_post()	send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()	setsockopt()
setuid()	shutdown()	sigaction()	sigaddset()
sigdelset()	sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()	sigprocmask()
sigqueue()	sigset()	sigsuspend()	sleep()
sockatmark()	socket()	socketpair()	stat()
symlink()	faccessat()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()	tcsendbreak()
tcsetattr()	tcsetpgrp()	time()	timer_getoverrun()
timer_gettime()	timer_settime()	times()	umask()
uname()	unlink()	utime()	wait()
waitpid()	write()	fchmodat()	fchownat()
fexecve()	fstatat()	futimens()	linkat()
mkdirat()	mkfifoat()	mknod()	mknodat()
openat()	readlinkat()	renameat()	symlinkat()
unlinkat()	utimensat()	utimes()	fchdir()
pthread_kill()	pthread_self()	pthread_sigmask()	

表 11.11.1 异步信号安全函数

上表所列举出的这些函数被认为是异步信号安全函数，可以通过 man 手册查询，执行命令"man 7 signal"，如下所示：

图 11.11.1 异步信号安全函数

大家可以通过对比 man 手册查询到的这些异步信号安全函数，来确定自己调用的库函数或系统调用是不是异步信号安全函数，这里需要说，在本书的示例代码中，并没有完全按照安全性要求，在信号处理函数中使用异步信号安全函数，譬如在本书中的示例代码中，信号处理函数中调用了 printf()用于打印信息，事实上这个函数是一个非异步信号安全函数，当然在一个实际的项目应用程序当中不能这么用，但是本书只是为了方便输出打印信息而已。

所以对于一个安全的信号处理函数来说，需要做到以下几点：

- 首先确保信号处理函数本身的代码是可重入的，且只能调用异步信号安全函数；
- 当主程序执行不安全函数或是去操作信号处理函数也可能会更新的全局数据结构时，要阻塞信号的传递。

关于异步信号安全函数就给大家介绍这么多，多线程环境下涉及到信号处理时尤其要注意这些问题。

⑸、多线程环境下信号的处理

第十二章 线程同步

本章来聊一聊线程同步这个话题，对于一个单线程进程来说，它不需要处理线程同步的问题，所以线程同步是在多线程环境下可能需要注意的一个问题。线程的主要优势在于，资源的共享性，譬如通过全局变量来实现信息共享，不过这种便捷的共享是有代价的，那就是多个线程并发访问共享数据所导致的数据不一致的问题。

本章来学习如何使用线程同步机制来避免这样的问题！

本章将会讨论如下主题内容。

- 为什么需要线程同步；
- 线程同步之互斥锁；
- 线程同步之信号量；
- 线程同步之条件变量；
- 线程同步之读写锁。

12.1 为什么需要线程同步？

线程同步是为了对共享资源的访问进行保护。这里说的共享资源指的是多个线程都会进行访问的资源，譬如定义了一个全局变量 a，线程 1 访问了变量 a、同样在线程 2 中也访问了变量 a，那么此时变量 a 就是多个线程间的共享资源，大家都要访问它。

保护的目的是为了解决数据一致性的问题。当然什么情况下才会出现数据一致性的问题，根据不同的情况进行区分；如果每个线程访问的变量都是其它线程不会读取和修改的（譬如线程函数内定义的局部变量或者只有一个线程访问的全局变量），那么就不存在数据一致性的问题；同样，如果变量是只读的，多个线程同时读取该变量也不会有数据一致性的问题；但是，当一个线程可以修改的变量，其它的线程也可以读取或者修改的时候，这个时候就存在数据一致性的问题，需要对这些线程进行同步操作，确保它们在访问变量的存储内容时不会访问到无效的值。

出现数据一致性问题其本质在于进程中的多个线程对共享资源的并发访问（同时访问）。前面给大家介绍了，进程中的多个线程间是并发执行的，每个线程都是系统调用的基本单元，参与到系统调度队列中；对于多个线程间的共享资源，并发执行会导致对共享资源的并发访问，并发访问所带来的问题就是竞争（如果多个线程同时对共享资源进行访问就表示存在竞争，跟现实生活当中的竞争有一定的相似之处，譬如一个队伍当中需要选出一名队长，现在有两个人在候选名单中，那么意味着这两个人就存在竞争关系），并发访问就可能会出现数据一致性问题，所以就需要解决这个问题；要防止并发访问共享资源，那么就需要对共享资源的访问进行保护，防止出现并发访问共享资源。

当一个线程修改变量时，其它的线程在读取这个变量时可能会看到不一致的值，图 12.1.1 描述了两个线程读写相同变量（共享变量、共享资源）的假设例子。在这个例子当中，线程 A 读取变量的值，然后再给这个变量赋予一个新的值，但写操作需要 2 个时钟周期（这里只是假设）；当线程 B 在这两个写周期中间读取了这个变量，它就会得到不一致的值，这就出现了数据不一致的问题。

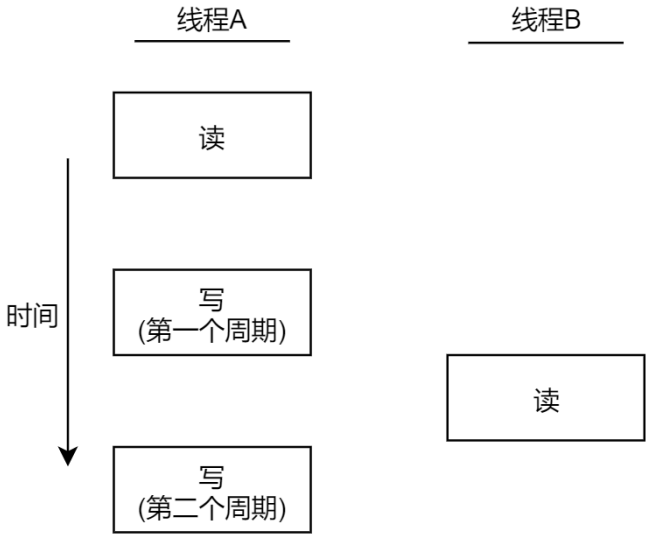


图 12.1.1 多线程并发访问数据不一致

我们可以编写一个简单地代码对此文件进行测试，示例代码 12.1.1 展示了在 2 个线程在常规方式下访问共享资源，这里的共享资源指的就是静态全局变量 g_count。该程序创建了两个线程，且均执行同一个函数，该函数执行一个循环，重复以下步骤：将全局变量 g_count 复制到本地变量 l_count 变量中，然后递增 l_count，再把 l_count 复制回 g_count，以此不断增加全局变量 g_count 的值。因为 l_count 是分配于线程栈中的自动变量（函数内定义的局部变量），所以每个线程都有一份。循环重复的次数要么由命令行参数指定，要么去默认值 1000 万次，循环结束之后线程终止，主线程回收两个线程之后，再将全局变量 g_count 的值打印出来。

示例代码 12.1.1 两个线程并发访问同一全局变量

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

static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
        l_count = g_count;
        l_count++;
        g_count = l_count;
    }

    return (void *)0;
}

static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000;     //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
```

```
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid2, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 打印结果 */
    printf("g_count = %d\n", g_count);
    exit(0);
}
```

编译代码，进行测试，首先执行代码，传入参数 1000，也就是让每个线程对全局变量 g_count 递增 1000 次，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp 1000
g_count = 2000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.1.2 1000 次测试结果

都打印结果看，得到了我们想象中的结果，每个线程递增 1000 次，最后的数值就是 2000；接着我们把递增次数加大，采用默认值 1000 万次，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
g_count = 10082309
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.1.3 1000 万次测试结果

可以发现，结果竟然不是我们想看到的样子，执行到最后，应该是 2000 万才对，这里其实就出现图 12.1.1 中所示的问题，数据不一致。

如何解决对共享资源的并发访问出现数据不一致的问题？

为了解决图 12.1.1 中数据不一致的问题，就得需要 Linux 提供的一些方法，也就是接下来将要向大家介绍的线程同步技术，来实现同一时间只允许一个线程访问该变量，防止出现并发访问的情况、消除数据不

一致的问题，图 12.1.4 描述了这种同步操作，从图中可知，线程 A 和线程 B 都不会同时访问这个变量，当线程 A 需要修改变量的值时，必须等到写操作完成之后（不能打断它的操作），才运行线程 B 去读取。

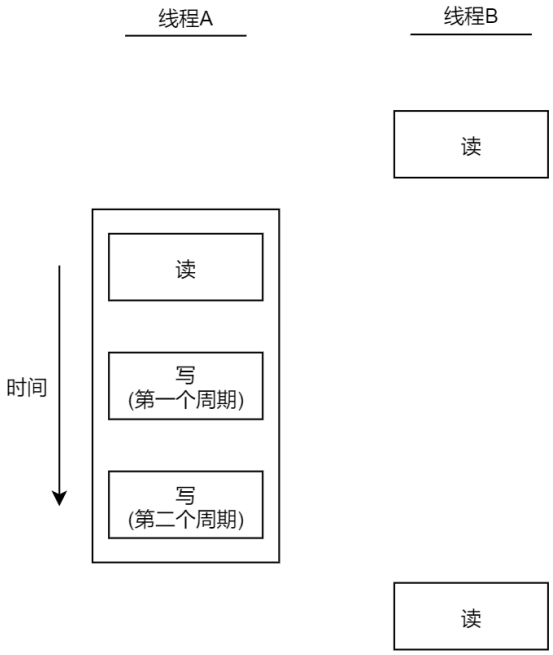


图 12.1.4 线程同步访问变量

线程的主要优势在于，资源的共享性，譬如通过全局变量来实现信息共享。不过这种便捷的共享是有代价的，必须确保多个线程不会同时修改同一变量、或者某一线程不会读取正由其它线程修改的变量，也就是必须确保不会出现对共享资源的并发访问。Linux 系统提供了多种用于实现线程同步的机制，常见的方法有：互斥锁、条件变量、自旋锁以及读写锁等，下面将向大家一一进行介绍。

12.2 互斥锁

互斥锁（mutex）又叫互斥量，从本质上说是一把锁，在访问共享资源之前对互斥锁进行上锁，在访问完成后释放互斥锁（解锁）；对互斥锁进行上锁之后，任何其它试图再次对互斥锁进行加锁的线程都会被阻塞，直到当前线程释放互斥锁。如果释放互斥锁时有一个以上的线程阻塞，那么这些阻塞的线程会被唤醒，它们都会尝试对互斥锁进行加锁，当有一个线程成功对互斥锁上锁之后，其它线程就不能再次上锁了，只能再次陷入阻塞，等待下一次解锁。

举一个非常简单容易理解的例子，就拿卫生间（共享资源）来说，当来了一个人（线程）看到卫生间没人，然后它进去了、并且从里边把门锁住（互斥锁上锁）了；此时又来了两个人（线程），它们也想进卫生间方便，发生此时门打不开（互斥锁上锁失败），因为里边有人，所以此时它们只能等待（陷入阻塞）；当里边的人方便完了之后（访问共享资源完成），把锁（互斥锁解锁）打开从里边出来，此时外边有两个人在等，当然它们都迫不及待想要进去（尝试对互斥锁进行上锁），自然两个人只能进去一个，进去的人再次把门锁住，另外一个人只能继续等待它出来。

在我们的程序设计当中，只有将所有线程访问共享资源都设计成相同的数据访问规则，互斥锁才能正常工作。如果允许其中的某个线程在没有得到锁的情况下也可以访问共享资源，那么即使其它的线程在使用共享资源前都申请锁，也还是会出现数据不一致的问题。

互斥锁使用 pthread_mutex_t 数据类型表示，在使用互斥锁之前，必须首先对它进行初始化操作，可以使用两种方式对互斥锁进行初始化操作。

12.2.1 互斥锁初始化

1、使用 PTHREAD_MUTEX_INITIALIZER 宏初始化互斥锁

互斥锁使用 pthread_mutex_t 数据类型表示，pthread_mutex_t 其实是一个结构体类型，而宏 PTHREAD_MUTEX_INITIALIZER 其实是一个对结构体赋值操作的封装，如下所示：

```
# define PTHREAD_MUTEX_INITIALIZER \
  { { 0, 0, 0, 0, 0, __PTHREAD_SPINS, { 0, 0 } } }
```

所以由此可知，使用 PTHREAD_MUTEX_INITIALIZER 宏初始化互斥锁的操作如下：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

PTHREAD_MUTEX_INITIALIZER 宏已经携带了互斥锁的默认属性。

2、使用 pthread_mutex_init()函数初始化互斥锁

使用 PTHREAD_MUTEX_INITIALIZER 宏只适用于在定义的时候就直接进行初始化，对于其它情况则不能使用这种方式，譬如先定义互斥锁，后再进行初始化，或者在堆中动态分配的互斥锁，譬如使用 malloc()函数申请分配的互斥锁对象，那么在这些情况下，可以使用 pthread_mutex_init()函数对互斥锁进行初始化，其函数原型如下所示：

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

使用该函数需要包含头文件<pthread.h>。

函数参数和返回值含义如下：

mutex：参数 mutex 是一个 pthread_mutex_t 类型指针，指向需要进行初始化操作的互斥锁对象；

attr：参数 attr 是一个 pthread_mutexattr_t 类型指针，指向一个 pthread_mutexattr_t 类型对象，该对象用于定义互斥锁的属性（在 12.2.6 小计中介绍），若将参数 attr 设置为 NULL，则表示将互斥锁的属性设置为默认值，在这种情况下其实就等价于 PTHREAD_MUTEX_INITIALIZER 这种方式初始化，而不同之处在于，使用宏不进行错误检查。

返回值：成功返回 0；失败将返回一个非 0 的错误码。

Tips：注意，当在 Ubuntu 系统下执行"man 3 pthread_mutex_init"命令时提示找不到该函数，并不是 Linux 下没有这个函数，而是该函数相关的 man 手册帮助信息没有被安装，这时我们只需执行"sudo apt-get install manpages-posix-dev"安装即可。

使用 pthread_mutex_init()函数对互斥锁进行初始化示例：

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

或者：

```
pthread_mutex_t *mutex = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(mutex, NULL);
```

12.2.2 互斥锁加锁和解锁

互斥锁初始化之后，处于一个未锁定状态，调用函数 pthread_mutex_lock()可以对互斥锁加锁、获取互斥锁，而调用函数 pthread_mutex_unlock()可以对互斥锁解锁、释放互斥锁。其函数原型如下所示：

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

使用这些函数需要包含头文件<pthread.h>，参数 mutex 指向互斥锁对象；pthread_mutex_lock()和 pthread_mutex_unlock()在调用成功时返回 0；失败将返回一个非 0 值的错误码。

调用 pthread_mutex_lock()函数对互斥锁进行上锁，如果互斥锁处于未锁定状态，则此次调用会上锁成功，函数调用将立马返回；如果互斥锁此时已经被其它线程锁定了，那么调用 pthread_mutex_lock()会一直阻塞，直到该互斥锁被解锁，到那时，调用将锁定互斥锁并返回。

调用 pthread_mutex_unlock()函数将已经处于锁定状态的互斥锁进行解锁。以下行为均属错误：

- 对处于未锁定状态的互斥锁进行解锁操作；
- 解锁由其它线程锁定的互斥锁。

如果有多个线程处于阻塞状态等待互斥锁被解锁，当互斥锁被当前锁定它的线程调用 pthread_mutex_unlock()函数解锁后，这些等待着的线程都会有机会对互斥锁上锁，但无法判断究竟哪个线程会如愿以偿！

使用示例

使用互斥锁的方式将示例代码 12.1.1 进行修改，修改之后如示例代码 12.2.1 所示，使用了一个互斥锁来保护对全局变量 g_count 的访问。

示例代码 12.2.1 使用互斥锁保护全局变量的访问

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

static pthread_mutex_t mutex;
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mutex); //互斥锁上锁

        l_count = g_count;
        l_count++;
        g_count = l_count;

        pthread_mutex_unlock(&mutex);//互斥锁解锁
    }

    return (void *)0;
}
```

```
static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000;    //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid2, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 打印结果 */
    printf("g_count = %d\n", g_count);
```

```
    exit(0);
}
```

在测试运行，使用默认值 1000 万次，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
g_count = 20000000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.2.1 测试结果

可以看到确实得到了我们想看到的正确结果，每次对 g_count 的累加总是能够保持正确，但是在运行程序的过程中，明显会感觉到锁消耗的时间会比较长，这就涉及到性能的问题了，后续会介绍！

12.2.3 pthread_mutex_trylock()函数

当互斥锁已经被其它线程锁住时，调用 pthread_mutex_lock()函数会被阻塞，直到互斥锁解锁；如果线程不希望被阻塞，可以使用 pthread_mutex_trylock()函数；调用 pthread_mutex_trylock()函数尝试对互斥锁进行加锁，如果互斥锁处于未锁住状态，那么调用 pthread_mutex_trylock()将会锁住互斥锁并立马返回，如果互斥锁已经被其它线程锁住，调用 pthread_mutex_trylock()加锁失败，但不会阻塞，而是返回错误码 EBUSY。

其函数原型如下所示：

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

参数 mutex 指向目标互斥锁，成功返回 0，失败返回一个非 0 值的错误码，如果目标互斥锁已经被其它线程锁住，则调用失败返回 EBUSY。

使用示例

对示例代码 12.2.1 进行修改，使用 pthread_mutex_trylock()替换 pthread_mutex_lock()。

示例代码 12.2.2 以非阻塞方式对互斥锁进行加锁

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

static pthread_mutex_t mutex;
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
```

```
        while(pthread_mutex_trylock(&mutex));     //以非阻塞方式上锁

        l_count = g_count;
        l_count++;
        g_count = l_count;

        pthread_mutex_unlock(&mutex);//互斥锁解锁
    }

    return (void *)0;
}

static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000;     //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
```

```
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid2, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 打印结果 */
    printf("g_count = %d\n", g_count);
    exit(0);
}
```

整个执行结果跟使用 pthread_mutex_lock()效果是一样的，大家可以自己测试。

12.2.4 销毁互斥锁

当不再需要互斥锁时，应该将其销毁，通过调用 pthread_mutex_destroy()函数来销毁互斥锁，其函数原型如下所示：

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

使用该函数需要包含头文件<pthread.h>，参数 mutex 指向目标互斥锁；同样在调用成功情况下返回 0，失败返回一个非 0 值的错误码。

- 不能销毁还没有解锁的互斥锁，否则将会出现错误；
- 没有初始化的互斥锁也不能销毁。

被 pthread_mutex_destroy()销毁之后的互斥锁，就不能再对它进行上锁和解锁了，需要再次调用 pthread_mutex_init()对互斥锁进行初始化之后才能使用。

使用示例

对示例代码 12.2.1 进行修改，在进程退出之前，使用 pthread_mutex_destroy()函数销毁互斥锁。

示例代码 12.2.3 销毁互斥锁

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

static pthread_mutex_t mutex;
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
```

```
    int l_count, j;

    for (j = 0; j < loops; j++) {
        pthread_mutex_lock(&mutex); //互斥锁上锁

        l_count = g_count;
        l_count++;
        g_count = l_count;

        pthread_mutex_unlock(&mutex);//互斥锁解锁
    }

    return (void *)0;
}

static int loops;
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int ret;

    /* 获取用户传递的参数 */
    if (2 > argc)
        loops = 10000000;     //没有传递参数默认为 1000 万次
    else
        loops = atoi(argv[1]);

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建 2 个新线程 */
    ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }
```

```
    /* 等待线程结束 */
    ret = pthread_join(tid1, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    ret = pthread_join(tid2, NULL);
    if (ret) {
        fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
        exit(-1);
    }

    /* 打印结果 */
    printf("g_count = %d\n", g_count);

    /* 销毁互斥锁 */
    pthread_mutex_destroy(&mutex);
    exit(0);
}
```

12.2.5 互斥锁死锁

试想一下，如果一个线程试图对同一个互斥锁加锁两次，会出现什么情况？情况就是该线程会陷入死锁状态，一直被阻塞永远出不来；这就是出现死锁的一种情况，除此之外，使用互斥锁还有其它很多种方式也能产生死锁。

有时，一个线程需要同时访问两个或更多不同的共享资源，而每个资源又由不同的互斥锁管理。当超过一个线程对同一组互斥锁（两个或两个以上的互斥锁）进行加锁时，就有可能发生死锁；譬如，程序中使用一个以上的互斥锁，如果允许一个线程一直占有第一个互斥锁，并且在试图锁住第二个互斥锁时处于阻塞状态，但是拥有第二个互斥锁的线程也在试图锁住第一个互斥锁。因为两个线程都在相互请求另一个线程拥有的资源，所以这两个线程都无法向前运行，会被一直阻塞，于是就产生了死锁。如下示例代码中所示：

```
// 线程 A
pthread_mutex_lock(mutex1);
pthread_mutex_lock(mutex2);

// 线程 B
pthread_mutex_lock(mutex2);
pthread_mutex_lock(mutex1);
```

这就好比是 C 语言中两个头文件相互包含的关系，那肯定编译报错！

在我们的程序当中，如果用到了多个互斥锁，要避免此类死锁的问题，最简单的方式就是定义互斥锁的层级关系，当多个线程对一组互斥锁操作时，总是应该按照相同的顺序对该组互斥锁进行锁定。譬如在上述场景中，如果两个线程总是先锁定 mutex1 在锁定 mutex2，死锁就不会出现。有时，互斥锁之间的层级关系逻辑不够清晰，即使是这样，依然可以设计出所有线程都必须遵循的强制层级顺序。

但有时候，应用程序的结构使得对互斥锁进行排序是很困难的，程序复杂、其中所涉及到的互斥锁以及共享资源比较多，程序设计实在无法按照相同的顺序对一组互斥锁进行锁定，那么就必须采用另外的方法。譬如使用 pthread_mutex_trylock()以不阻塞的方式尝试对互斥锁进行加锁，在这种方案中，线程先使用函数 pthread_mutex_lock()锁定第一个互斥锁，然后使用 pthread_mutex_trylock()来锁定其余的互斥锁。如果任一 pthread_mutex_trylock()调用失败（返回 EBUSY），那么该线程释放所有互斥锁，可以经过一段时间之后从头再试。与第一种按照层级关系来避免死锁的方法变比，这种方法效率要低一些，因为可能需要经历多次循环。

解决互斥锁死锁的问题还有很多方法，笔者也没详细地去学习过，当大家在实际编程应用中需要用到这些知识再去查阅相关资料、书籍进行学习。

使用示例

想了半天没有什么比较好的例子，暂时先歇下！

12.2.6 互斥锁的属性

如前所述，调用 pthread_mutex_init()函数初始化互斥锁时可以设置互斥锁的属性，通过参数 attr 指定。参数 attr 指向一个 pthread_mutexattr_t 类型对象，该对象对互斥锁的属性进行定义，当然，如果将参数 attr 设置为 NULL，则表示将互斥锁属性设置为默认值。关于互斥锁的属性本书不打算深入讨论互斥锁属性的细节，也不会将 pthread_mutexattr_t 类型中定义的属性一一列出。

如果不使用默认属性，在调用 pthread_mutex_init()函数时，参数 attr 必须要指向一个 pthread_mutexattr_t 对象，而不能使用 NULL。当定义 pthread_mutexattr_t 对象之后，需要使用 pthread_mutexattr_init()函数对该对象进行初始化操作，当对象不再使用时，需要使用 pthread_mutexattr_destroy()将其销毁，函数原型如下所示：

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

参数 attr 指向需要进行初始化的 pthread_mutexattr_t 对象，调用成功返回 0，失败将返回非 0 值的错误码。

pthread_mutexattr_init()函数将使用默认的互斥锁属性初始化参数 attr 指向的 pthread_mutexattr_t 对象。关于互斥锁的属性比较多，譬如进程共享属性、健壮属性、类型属性等等，本书并不会一一给大家进行介绍，本小节讨论下类型属性，其它的暂时不去解释了。

互斥锁的类型属性控制着互斥锁的锁定特性，一共有 4 中类型：

- **PTHREAD_MUTEX_NORMAL**：一种标准的互斥锁类型，不做任何的错误检查或死锁检测。如果线程试图对已经由自己锁定的互斥锁再次进行加锁，则发生死锁；互斥锁处于未锁定状态，或者已由其它线程锁定，对其解锁会导致不确定结果。
- **PTHREAD_MUTEX_ERRORCHECK**：此类互斥锁会提供错误检查。譬如这三种情况都会导致返回错误：线程试图对已经由自己锁定的互斥锁再次进行加锁（同一线程对同一互斥锁加锁两次），返回错误；线程对由其它线程锁定的互斥锁进行解锁，返回错误；线程对处于未锁定状态的互斥锁进行解锁，返回错误。这类互斥锁运行起来比较慢，因为它需要做错误检查，不过可将其作为调试工具，以发现程序哪里违反了互斥锁使用的基本原则。
- **PTHREAD_MUTEX_RECURSIVE**：此类互斥锁允许同一线程在互斥锁解锁之前对该互斥锁进行多次加锁，然后维护互斥锁加锁的次数，把这种互斥锁称为递归互斥锁，但是如果解锁次数不等于

加速次数，则是不会释放锁的；所以，如果对一个递归互斥锁加锁两次，然后解锁一次，那么这个互斥锁依然处于锁定状态，对它再次进行解锁之前不会释放该锁。

- **PTHREAD_MUTEX_DEFAULT**：此类互斥锁提供默认的行为和特性。使用宏 PTHREAD_MUTEX_INITIALIZER 初始化的互斥锁，或者调用参数 arg 为 NULL 的 pthread_mutexattr_init()函数所创建的互斥锁，都属于此类型。此类锁意在为互斥锁的实现保留最大灵活性，Linux 上，PTHREAD_MUTEX_DEFAULT 类型互斥锁的行为与 PTHREAD_MUTEX_NORMAL 类型相仿。

可以使用 pthread_mutexattr_gettype()函数得到互斥锁的类型属性，使用 pthread_mutexattr_settype()修改/设置互斥锁类型属性，其函数原型如下所示：

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

使用这些函数需要包含头文件<pthread.h>，参数 attr 指向 pthread_mutexattr_t 类型对象；对于 pthread_mutexattr_gettype()函数，函数调用成功会将互斥锁类型属性保存在参数 type 所指向的内存中，通过它返回出来；而对于 pthread_mutexattr_settype()函数，会将参数 attr 指向的 pthread_mutexattr_t 对象的类型属性设置为参数 type 指定的类型。使用方式如下：

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

/* 初始化互斥锁属性对象 */
pthread_mutexattr_init(&attr);

/* 将类型属性设置为 PTHREAD_MUTEX_NORMAL */
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);

/* 初始化互斥锁 */
pthread_mutex_init(&mutex, &attr);

.....

/* 使用完之后 */
pthread_mutexattr_destroy(&attr);
pthread_mutex_destroy(&mutex);
```

12.3 条件变量

本小节讨论第二种线程同步的方法---条件变量。

条件变量是线程可用的另一种同步机制。条件变量用于自动阻塞线程，知道某个特定事件发生或某个条件满足为止，通常情况下，条件变量是和互斥锁一起搭配使用的。使用条件变量主要包括两个动作：

- 一个线程等待某个条件满足而被阻塞；
- 另一个线程中，条件满足时发出“信号”。

为了说明这个问题，来看一个没有使用条件变量的例子，生产者---消费者模式，生产者这边负责生产产品、而消费者负责消费产品，对于消费者来说，没有产品的时候只能等待产品出来，有产品就使用它。

这里我们使用一个变量来表示这个这个产品，生产者生产一件产品变量加 1，消费者消费一次变量减 1，示例代码如下所示：

示例代码 12.3.1 生产者---消费者示例代码

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

static pthread_mutex_t mutex;
static int g_avail = 0;

/* 消费者线程 */
static void *consumer_thread(void *arg)
{
    for ( ; ; ) {
        pthread_mutex_lock(&mutex);//上锁

        while (g_avail > 0)
            g_avail--;          //消费

        pthread_mutex_unlock(&mutex);//解锁
    }

    return (void *)0;
}

/* 主线程（生产者） */
int main(int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    /* 初始化互斥锁 */
    pthread_mutex_init(&mutex, NULL);

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, consumer_thread, NULL);
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }
```

```
    for ( ; ; ) {
        pthread_mutex_lock(&mutex);//上锁
        g_avail++;          //生产
        pthread_mutex_unlock(&mutex);//解锁
    }

    exit(0);
}
```

此代码中，主线程作为“生产者”，新创建的线程作为“消费者”，运行之后它们都回处于死循环中，所以代码中没有加入销毁互斥锁、等待回收新线程相关的代码，进程终止时会自动被处理。

上述代码虽然可行，但由于新线程中会不停的循环检查全局变量 g_avail 是否大于 0，故而造成 CPU 资源的浪费。采用条件变量这一问题就可以迎刃而解！条件变量允许一个线程休眠（阻塞等待）直至获取到另一个线程的通知（收到信号）再去执行自己的操作，譬如上述代码中，当条件 g_avail > 0 不成立时，消费者线程会进入休眠状态，而生产者生成产品后（g_avail++，此时 g_avail 将会大于 0），向处于等待状态的线程发出“信号”，而其它线程收到“信号”之后，便会被唤醒！

Tips：这里提到的信号并不是第八章内容所指的信号，需要区分开来！

前面说到，条件变量通常搭配互斥锁来使用，是因为条件的检测是在互斥锁的保护下进行的，也就是说条件本身是由互斥锁保护的，线程在改变条件状态之前必须首先锁住互斥锁，不然就可能引发线程不安全的问题。

12.3.1 条件变量初始化

条件变量使用 pthread_cond_t 数据类型来表示，类似于互斥锁，在使用条件变量之前必须对其进行初始化。初始化方式同样也有两种：使用宏 PTHREAD_COND_INITIALIZER 或者使用函数 pthread_cond_init()，使用宏的初始化方法与互斥锁的初始化宏一样，这里就不再重述！譬如：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

pthread_cond_init()函数原型如下所示：

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

同样，使用这些函数需要包含头文件<pthread.h>，使用 pthread_cond_init()函数初始化条件变量，当不再使用时，使用 pthread_cond_destroy()销毁条件变量。

参数 cond 指向 pthread_cond_t 条件变量对象，对于 pthread_cond_init()函数，类似于互斥锁，在初始化条件变量时设置条件变量的属性，参数 attr 指向一个 pthread_condattr_t 类型对象，pthread_condattr_t 数据类型用于描述条件变量的属性。可将参数 attr 设置为 NULL，表示使用属性的默认值来初始化条件变量，与使用 PTHREAD_COND_INITIALIZER 宏相同。

函数调用成功返回 0，失败将返回一个非 0 值的错误码。

对于初始化与销毁操作，有以下问题需要注意：

- 在使用条件变量之前必须对条件变量进行初始化操作，使用 PTHREAD_COND_INITIALIZER 宏或者函数 pthread_cond_init()都行；
- 对已经初始化的条件变量再次进行初始化，将可能会导致未定义行为；
- 对没有进行初始化的条件变量进行销毁，也将可能会导致未定义行为；
- 对某个条件变量而言，仅当没有任何线程等待它时，将其销毁才是最安全的；

- 经 pthread_cond_destroy()销毁的条件变量，可以再次调用 pthread_cond_init()对其进行重新初始化。

12.3.2 通知和等待条件变量

条件变量的主要操作便是发送信号（signal）和等待。发送信号操作即是通知一个或多个处于等待状态的线程，某个共享变量的状态已经改变，这些处于等待状态的线程收到通知之后便会被唤醒，唤醒之后再检查条件是否满足。等待操作是指在收到一个通知前一直处于阻塞状态。

函数 pthread_cond_signal()和 pthread_cond_broadcast()均可向指定的条件变量发送信号，通知一个或多个处于等待状态的线程。调用 pthread_cond_wait()函数是线程阻塞，直到收到条件变量的通知。

pthread_cond_signal()和 pthread_cond_broadcast()函数原型如下所示：

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

使用这些函数需要包含头文件<pthread.h>，参数 cond 指向目标条件变量，向该条件变量发送信号。调用成功返回 0；失败将返回一个非 0 值的错误码。

pthread_cond_signal()和 pthread_cond_broadcast()的区别在于：二者对阻塞于 pthread_cond_wait()的多个线程对应的处理方式不同，pthread_cond_signal()函数至少能唤醒一个线程，而 pthread_cond_broadcast()函数则能唤醒所有线程。使用 pthread_cond_broadcast()函数总能产生正确的结果，唤醒所有等待状态的线程，但函数 pthread_cond_signal()会更为高效，因为它只需确保至少唤醒一个线程即可，所以如果我们的程序当中，只有一个处于等待状态的线程，使用 pthread_cond_signal()更好，具体使用哪个函数根据实际情况进行选择！

pthread_cond_wait()函数原型如下所示：

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

当程序当中使用条件变量，当判断某个条件不满足时，调用 pthread_cond_wait()函数将线程设置为等待状态（阻塞）。**pthread_cond_wait()函数包含两个参数：**

cond：指向需要等待的条件变量，目标条件变量；

mutex：参数 mutex 是一个 pthread_mutex_t 类型指针，指向一个互斥锁对象；前面开头便给大家介绍了，条件变量通常是和互斥锁一起使用，因为条件的检测（条件检测通常是需要访问共享资源的）是在互斥锁的保护下进行的，也就是说条件本身是由互斥锁保护的。

返回值：调用成功返回 0；失败将返回一个非 0 值的错误码。

在 pthread_cond_wait()函数内部会对参数 mutex 所指定的互斥锁进行操作，通常情况下，条件判断以及 pthread_cond_wait()函数调用均在互斥锁的保护下，也就是说，在此之前线程已经对互斥锁加锁了。调用 pthread_cond_wait()函数时，调用者把互斥锁传递给函数，函数会自动把调用线程放到等待条件的线程列表上，然后将互斥锁解锁；当 pthread_cond_wait()被唤醒返回时，会再次锁住互斥锁。

注意注意的是，条件变量并不保存状态信息，只是传递应用程序状态信息的一种通讯机制。如果调用 pthread_cond_signal()和 pthread_cond_broadcast()向指定条件变量发送信号时，若无任何线程等待该条件变量，这个信号也就会不了了之。

当调用 pthread_cond_broadcast()同时唤醒所有线程时，互斥锁也只能被某一线程锁住，其它线程获取锁失败又会陷入阻塞。

使用示例

使用条件变量对示例代码 12.3.1 进行修改，当消费者线程没有产品可消费时，让它处于等待状态，知道生产者把产品生产出来；当生产者把产品生产出来之后，再去通知消费者。

示例代码 12.3.2 使用条件变量和互斥锁实现线程同步

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

static pthread_mutex_t mutex;     //定义互斥锁
static pthread_cond_t cond;       //定义条件变量
static int g_avail = 0;           //全局共享资源

/* 消费者线程 */
static void *consumer_thread(void *arg)
{
    for ( ; ; ) {
        pthread_mutex_lock(&mutex);//上锁

        while (0 >= g_avail)
            pthread_cond_wait(&cond, &mutex);//等待条件满足

        while (0 < g_avail)
            g_avail--;          //消费

        pthread_mutex_unlock(&mutex);//解锁
    }

    return (void *)0;
}

/* 主线程（生产者） */
int main(int argc, char *argv[])
{
    pthread_t tid;
    int ret;

    /* 初始化互斥锁和条件变量 */
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    /* 创建新线程 */
    ret = pthread_create(&tid, NULL, consumer_thread, NULL);
```

```
    if (ret) {
        fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
        exit(-1);
    }

    for ( ; ; ) {
        pthread_mutex_lock(&mutex);//上锁
        g_avail++;          //生产
        pthread_mutex_unlock(&mutex);//解锁
        pthread_cond_signal(&cond);//向条件变量发送信号
    }

    exit(0);
}
```

全局变量 g_avail 作为主线程和新线程之间的共享资源，两个线程在访问它们之间首先会对互斥锁进行上锁，消费者线程中，当判断没有产品可被消费时（g_avail <= 0），调用 pthread_cond_wait()使得线程陷入等待状态，等待条件变量，等待生产者制造产品；调用 pthread_cond_wait()后线程阻塞并解锁互斥锁；而在生产者线程中，它的任务是生产产品（使用 g_avail++来模拟），产品生产完成之后，调用 pthread_mutex_unlock()将互斥锁解锁，并调用 pthread_cond_signal()向条件变量发送信号；这将会唤醒处于等待该条件变量的消费者线程，唤醒之后再次自动获取互斥锁，然后再对产品进行消费（g_avai--模拟）。

12.3.3 条件变量的判断条件

使用条件变量，都会有与之相关的判断条件，通常情况下，会涉及到一个或多个共享变量。譬如在示例代码 12.3.2 中，与条件变量相关的判断是(0 >= g_avail)。细心的读者会发现，在这份示例代码中，我们使用了 while 循环、而不是 if 语句，来控制对 pthread_cond_wait()的调用，这是为何呢？

必须使用 while 循环，而不是 if 语句，这是一种通用的设计原则：当线程从 pthread_cond_wait()返回时，并不能确定判断条件的状态，应该立即重新检查判断条件，如果条件不满足，那就继续休眠等待。

从 pthread_cond_wait()返回后，并不能确定判断条件是真还是假，其理由如下：

- 当有多于一个线程在等待条件变量时，任何线程都有可能会率先醒来获取互斥锁，率先醒来获取到互斥锁的线程可能会对共享变量进行修改，进而改变判断条件的状态。譬如示例代码 12.3.2 中，如果有两个或更多个消费者线程，当其中一个消费者线程从 pthread_cond_wait()返回后，它会将全局共享变量 g_avail 的值变成 0，导致判断条件的状态由真变成假。
- 可能会发出虚假的通知。

12.3.4 条件变量的属性

如前所述，调用 pthread_cond_init()函数初始化条件变量时，可以设置条件变量的属性，通过参数 attr 指定。参数 attr 指向一个 pthread_condattr_t 类型对象，该对象对条件变量的属性进行定义，当然，如果将参数 attr 设置为 NULL，表示使用默认值来初始化条件变量属性。

关于条件变量的属性本书不打算深入讨论，条件变量包括两个属性：进程共享属性和时钟属性。每个属性都提供了相应的 get 方法和 set 方法，各位读者如果有兴趣，可自行查阅资料学习，本书不再介绍！

12.4 自旋锁

自旋锁与互斥锁很相似，从本质上说也是一把锁，在访问共享资源之前对自旋锁进行上锁，在访问完成后释放自旋锁（解锁）；事实上，从实现方式上来说，互斥锁是基于自旋锁来实现的，所以自旋锁相较于互斥锁更加底层。

如果在获取自旋锁时，自旋锁处于未锁定状态，那么将立即获得锁（对自旋锁上锁）；如果在获取自旋锁时，自旋锁已经处于锁定状态了，那么获取锁操作将会在原地“自旋”，直到该自旋锁的持有者释放了锁。由此介绍可知，自旋锁与互斥锁相似，但是互斥锁在无法获取到锁时会让线程陷入阻塞等待状态；而自旋锁在无法获取到锁时，将会在原地“自旋”等待。“自旋”其实就是调用者一直在循环查看该自旋锁的持有者是否已经释放了锁，“自旋”一词因此得名。

自旋锁的不足之处在于：自旋锁一直占用的 CPU，它在未获得锁的情况下，一直处于运行状态（自旋），所以占着 CPU，如果不能在很短的时间内获取锁，这无疑会使 CPU 效率降低。

试图对同一自旋锁加锁两次必然会导致死锁，而试图对同一互斥锁加锁两次不一定会导致死锁，原因在于互斥锁有不同的类型，当设置为 PTHREAD_MUTEX_ERRORCHECK 类型时，会进行错误检查，第二次加锁会返回错误，所以不会进入死锁状态。

因此我们要谨慎使用自旋锁，自旋锁通常用于以下情况：需要保护的代码段执行时间很短，这样就会使得持有锁的线程会很快释放锁，而“自旋”等待的线程也只需等待很短的时间；在这种情况下就比较适合使用自旋锁，效率高！

综上所述，再来总结下自旋锁与互斥锁之间的区别：

- 实现方式上的区别：互斥锁是基于自旋锁而实现的，所以自旋锁相较于互斥锁更加底层；
- 开销上的区别：获取不到互斥锁会陷入阻塞状态（休眠），直到获取到锁时被唤醒；而获取不到自旋锁会在原地“自旋”，直到获取到锁；休眠与唤醒开销是很大的，所以互斥锁的开销要远高于自旋锁、自旋锁的效率远高于互斥锁；但如果长时间的“自旋”等待，会使得 CPU 使用效率降低，故自旋锁不适用于等待时间比较长的情况。
- 使用场景的区别：自旋锁在用户态应用程序中使用的比较少，通常在内核代码中使用比较多；因为自旋锁可以在中断服务函数中使用，而互斥锁则不行，在执行中断服务函数时要求不能休眠、不能被抢占（内核中使用自旋锁会自动禁止抢占），一旦休眠意味着执行中断服务函数时主动交出了 CPU 使用权，休眠结束时无法返回到中断服务函数中，这样就会导致死锁！

12.4.1 自旋锁初始化

自旋锁使用 pthread_spinlock_t 数据类型表示，当定义自旋锁后，需要使用 pthread_spin_init()函数对其进行初始化，当不再使用自旋锁时，调用 pthread_spin_destroy()函数将其销毁，其函数原型如下所示：

```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

使用这两个函数需要包含头文件<pthread.h>。

参数 lock 指向了需要进行初始化或销毁的自旋锁对象，参数 pshared 表示自旋锁的进程共享属性，可以取值如下：

- **PTHREAD_PROCESS_SHARED**：共享自旋锁。该自旋锁可以在多个进程中的线程之间共享；
- **PTHREAD_PROCESS_PRIVATE**：私有自旋锁。只有本进程内的线程才能够使用该自旋锁。

这两个函数在调用成功的情况下返回 0；失败将返回一个非 0 值的错误码。

12.4.2 自旋锁加锁和解锁

可以使用 pthread_spin_lock()函数或 pthread_spin_trylock()函数对自旋锁进行加锁，前者在未获取到锁时一直“自旋”；对于后者，如果未能获取到锁，就立刻返回错误，错误码为 EBUSY。不管以何种方式加锁，自旋锁都可以使用 pthread_spin_unlock()函数对自旋锁进行解锁。其函数原型如下所示：

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

使用这些函数需要包含头文件<pthread.h>。

参数 lock 指向自旋锁对象，调用成功返回 0，失败将返回一个非 0 值的错误码。

如果自旋锁处于未锁定状态，调用 pthread_spin_lock()会将其锁定（上锁），如果其它线程已经将自旋锁锁住了，那本次调用将会“自旋”等待；如果试图对同一自旋锁加锁两次必然会导致死锁。

使用示例

对示例代码 12.2.1 进行修改，使用自旋锁替换互斥锁来实现线程同步，对共享资源的访问进行保护。

示例代码 12.4.1 使用自旋锁实现线程同步

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

static pthread_spinlock_t spin;//定义自旋锁
static int g_count = 0;

static void *new_thread_start(void *arg)
{
    int loops = *((int *)arg);
    int l_count, j;

    for (j = 0; j < loops; j++) {
        pthread_spin_lock(&spin); //自旋锁上锁

        l_count = g_count;
        l_count++;
        g_count = l_count;

        pthread_spin_unlock(&spin);//自旋锁解锁
    }

    return (void *)0;
}
```

```
static int loops;
int main(int argc, char *argv[])
{
	pthread_t tid1, tid2;
	int ret;

	/* 获取用户传递的参数 */
	if (2 > argc)
		loops = 10000000;	//没有传递参数默认为 1000 万次
	else
		loops = atoi(argv[1]);

	/* 初始化自旋锁(私有) */
	pthread_spin_init(&spin, PTHREAD_PROCESS_PRIVATE);

	/* 创建 2 个新线程 */
	ret = pthread_create(&tid1, NULL, new_thread_start, &loops);
	if (ret) {
		fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
		exit(-1);
	}

	ret = pthread_create(&tid2, NULL, new_thread_start, &loops);
	if (ret) {
		fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
		exit(-1);
	}

	/* 等待线程结束 */
	ret = pthread_join(tid1, NULL);
	if (ret) {
		fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
		exit(-1);
	}

	ret = pthread_join(tid2, NULL);
	if (ret) {
		fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
		exit(-1);
	}

	/* 打印结果 */
```

```
    printf("g_count = %d\n", g_count);

    /* 销毁自旋锁 */
    pthread_spin_destroy(&spin);
    exit(0);
}
```

运行结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c -lpthread
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
g_count = 20000000
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 12.4.1 测试结果

将互斥锁替换为自旋锁之后，测试结果打印也是没有问题的，并且通过对比可以发现，替换为自旋锁之后，程序运行所耗费的时间明显变短了，说明自旋锁确实比互斥锁效率要高，但是一定要注意自旋锁所适用的场景。

12.5 读写锁

互斥锁或自旋锁要么是加锁状态、要么是不加锁状态，而且一次只有一个线程可以对其加锁。读写锁有 3 种状态：读模式下的加锁状态（以下简称读加锁状态）、写模式下的加锁状态（以下简称写加锁状态）和不加锁状态（见），一次只有一个线程可以占有写模式的读写锁，但是可以有多个线程同时占有读模式的读写锁。因此可知，读写锁比互斥锁具有更高的并行性！

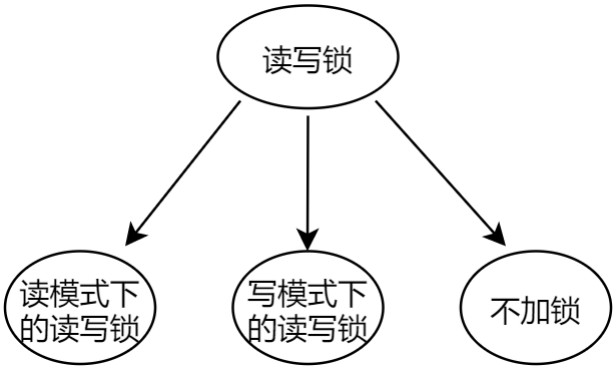


图 12.5.1 读写锁

读写锁有如下两个规则：

- 当读写锁处于写加锁状态时，在这个锁被解锁之前，所有试图对这个锁进行加锁操作（不管是以读模式加锁还是以写模式加锁）的线程都会被阻塞。
- 当读写锁处于读加锁状态时，所有试图以读模式对它进行加锁的线程都可以加锁成功；但是任何以写模式对它进行加锁的线程都会被阻塞，直到所有持有读模式锁的线程释放它们的锁为止。

虽然各操作系统对读写锁的实现各不相同，但当读写锁处于读模式加锁状态，而这时有一个线程试图以写模式获取锁时，该线程会被阻塞；而如果另一线程以读模式获取锁，则会成功获取到锁，对共享资源进行读操作。

所以，读写锁非常适合于对共享数据读的次数远大于写的次数的情况。当读写锁处于写模式加锁状态时，它所保护的数据可以被安全的修改，因为一次只有一个线程可以在写模式下拥有这个锁；当读写锁处于读模式加锁状态时，它所保护的数据就可以被多个获取读模式锁的线程读取。所以在应用程序当中，使用读写锁实现线程同步，当线程需要对共享数据进行读操作时，需要先获取读模式锁（对读模式锁进行加锁），当读取操作完成之后再释放读模式锁（对读模式锁进行解锁）；当线程需要对共享数据进行写操作时，需要先获取到写模式锁，当写操作完成之后再释放写模式锁。

读写锁也叫做共享互斥锁。当读写锁是读模式锁住时，就可以说成是共享模式锁住。当它是写模式锁住时，就可以说成是互斥模式锁住。

12.5.1 读写锁初始化

与互斥锁、自旋锁类似，在使用读写锁之前也必须对读写锁进行初始化操作，读写锁使用 pthread_rwlock_t 数据类型表示，读写锁的初始化可以使用宏 PTHREAD_RWLOCK_INITIALIZER 或者函数 pthread_rwlock_init()，其初始化方式与互斥锁相同，譬如使用宏 PTHREAD_RWLOCK_INITIALIZER 进行初始化必须在定义读写锁时就对其进行初始化：

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

对于其它方式可以使用 pthread_rwlock_init()函数对其进行初始化，当读写锁不再使用时，需要调用 pthread_rwlock_destroy()函数将其销毁，其函数原型如下所示：

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
```

使用这两个函数同样需要包含头文件<pthread.h>，调用成功返回 0，失败将返回一个非 0 值的错误码。

参数 rwlock 指向需要进行初始化或销毁的读写锁对象。对于 pthread_rwlock_init()函数，参数 attr 是一个 pthread_rwlockattr_t *类型指针，指向 pthread_rwlockattr_t 对象。pthread_rwlockattr_t 数据类型定义了读写锁的属性（在 12.5.3 小节中介绍），若将参数 attr 设置为 NULL，则表示将读写锁的属性设置为默认值，在这种情况下其实就等价于 PTHREAD_RWLOCK_INITIALIZER 这种方式初始化，而不同之处在于，使用宏不进行错误检查。

当读写锁不再使用时，需要调用 pthread_rwlock_destroy()函数将其销毁。

读写锁初始化使用示例：

```
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock, NULL);

......

pthread_rwlock_destroy(&rwlock);
```

12.5.2 读写锁上锁和解锁

以读模式对读写锁进行上锁，需要调用 pthread_rwlock_rdlock()函数；以写模式对读写锁进行上锁，需要调用 pthread_rwlock_wrlock()函数。不管是以何种方式锁住读写锁，均可以调用 pthread_rwlock_unlock()函数解锁，其函数原型如下所示：

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

使用这些函数需要包含头文件<pthread.h>，参数 rwlock 指向读写锁对象。调用成功返回 0，失败返回一个非 0 值的错误码。

当读写锁处于写模式加锁状态时，其它线程调用 pthread_rwlock_rdlock()或 pthread_rwlock_wrlock()函数均会获取锁失败，从而陷入阻塞等待状态；当读写锁处于读模式加锁状态时，其它线程调用 pthread_rwlock_rdlock()函数可以成功获取到锁，如果调用 pthread_rwlock_wrlock()函数则不能获取到锁，从而陷入阻塞等待状态。

如果线程不希望被阻塞，可以调用 pthread_rwlock_tryrdlock()和 pthread_rwlock_trywrlock()来尝试加锁，如果不可以获取锁时。这两个函数都会立马返回错误，错误码为 EBUSY。其函数原型如下所示：

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

参数 rwlock 指向需要加锁的读写锁，加锁成功返回 0，加锁失败则返回 EBUSY。

使用示例

示例代码 12.5.1 演示了使用读写锁来实现线程同步，全局变量 g_count 作为线程间的共享变量，主线程中创建了 5 个读取 g_count 变量的线程，它们使用同一个函数 read_thread，这 5 个线程仅仅对 g_count 变量进行读取，并将其打印出来，连带打印线程的编号（1~5）；主线程中还创建了 5 个写 g_count 变量的线程，它们使用同一个函数 write_thread，write_thread 函数中会将 g_count 变量的值进行累加，循环 10 次，每次将 g_count 变量的值在原来的基础上增加 20，并将其打印出来，连带打印线程的编号（1~5）。

示例代码 12.5.1 使用读写锁实现线程同步

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

static pthread_rwlock_t rwlock;//定义读写锁
static int g_count = 0;

static void *read_thread(void *arg)
{
    int number = *((int *)arg);
    int j;

    for (j = 0; j < 10; j++) {
        pthread_rwlock_rdlock(&rwlock); //以读模式获取锁
        printf("读线程<%d>, g_count=%d\n", number+1, g_count);
        pthread_rwlock_unlock(&rwlock);//解锁
        sleep(1);
    }
```

```
    return (void *)0;
}

static void *write_thread(void *arg)
{
    int number = *((int *)arg);
    int j;

    for (j = 0; j < 10; j++) {
        pthread_rwlock_wrlock(&rwlock); //以写模式获取锁
        printf("写线程<%d>, g_count=%d\n", number+1, g_count+=20);
        pthread_rwlock_unlock(&rwlock);//解锁
        sleep(1);
    }

    return (void *)0;
}

static int nums[5] = {0, 1, 2, 3, 4};
int main(int argc, char *argv[])
{
    pthread_t tid[10];
    int j;

    /* 对读写锁进行初始化 */
    pthread_rwlock_init(&rwlock, NULL);

    /* 创建 5 个读 g_count 变量的线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j], NULL, read_thread, &nums[j]);

    /* 创建 5 个写 g_count 变量的线程 */
    for (j = 0; j < 5; j++)
        pthread_create(&tid[j+5], NULL, write_thread, &nums[j]);

    /* 等待线程结束 */
    for (j = 0; j < 10; j++)
        pthread_join(tid[j], NULL);//回收线程

    /* 销毁自旋锁 */
    pthread_rwlock_destroy(&rwlock);
    exit(0);
}
```

编译测试，其打印结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
读线程<2>, g_count=0
读线程<3>, g_count=0
读线程<4>, g_count=0
读线程<5>, g_count=0
读线程<1>, g_count=0
写线程<2>, g_count=20
写线程<3>, g_count=40
写线程<4>, g_count=60
写线程<5>, g_count=80
写线程<1>, g_count=100
读线程<4>, g_count=100
读线程<3>, g_count=100
读线程<5>, g_count=100
读线程<1>, g_count=100
读线程<2>, g_count=100
写线程<3>, g_count=120
写线程<5>, g_count=140
写线程<1>, g_count=160
写线程<2>, g_count=180
写线程<4>, g_count=200
```

图 12.5.2 测试结果

在这个例子中，我们演示了读写锁的使用，但仅作为演示使用，在实际的应用编程中，需要根据应用场景来选择是否使用读写锁。

12.5.3 读写锁的属性

读写锁与互斥锁类似，也是有属性的，读写锁的属性使用 pthread_rwlockattr_t 数据类型来表示，当定义 pthread_rwlockattr_t 对象时，需要使用 pthread_rwlockattr_init()函数对其进行初始化操作，初始化会将 pthread_rwlockattr_t 对象定义的各个读写锁属性初始化为默认值；当不再使用 pthread_rwlockattr_t 对象时，需要调用 pthread_rwlockattr_destroy()函数将其销毁，其函数原型如下所示：

```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

参数 attr 指向需要进行初始化或销毁的 pthread_rwlockattr_t 对象；函数调用成功返回 0，失败将返回一个非 0 值的错误码。

读写锁只有一个属性，那便是进程共享属性，它与互斥锁以及自旋锁的进程共享属性相同。Linux 下提供了相应的函数用于设置或获取读写锁的共享属性。函数 pthread_rwlockattr_getpshared()用于从 pthread_rwlockattr_t 对象中获取共享属性，函数 pthread_rwlockattr_setpshared()用于设置 pthread_rwlockattr_t 对象中的共享属性，其函数原型如下所示：

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

函数 pthread_rwlockattr_getpshared()参数和返回值：

attr：指向 pthread_rwlockattr_t 对象；

pshared：调用 pthread_rwlockattr_getpshared()获取共享属性，将其保存在参数 pshared 所指向的内存中；

返回值：成功返回 0，失败将返回一个非 0 值的错误码。

函数 pthread_rwlockattr_setpshared()参数和返回值：

attr：指向 pthread_rwlockattr_t 对象；

pshared：调用 pthread_rwlockattr_setpshared()设置读写锁的共享属性，将其设置为参数 pshared 指定的值。参数 pshared 可取值如下：

- **PTHREAD_PROCESS_SHARED：**共享读写锁。该读写锁可以在多个进程中的线程之间共享；
- **PTHREAD_PROCESS_PRIVATE：**私有读写锁。只有本进程内的线程才能够使用该读写锁，这是读写锁共享属性的默认值。

返回值：调用成功的情况下返回 0；失败将返回一个非 0 值的错误码。

使用方式如下：

```
pthread_rwlock_t rwlock;      //定义读写锁
pthread_rwlockattr_t attr;    //定义读写锁属性

/* 初始化读写锁属性对象 */
pthread_rwlockattr_init(&attr);

/* 将进程共享属性设置为 PTHREAD_PROCESS_PRIVATE */
pthread_rwlockattr_setpshared(&attr, PTHREAD_PROCESS_PRIVATE);

/* 初始化读写锁 */
pthread_rwlock_init(&rwlock, &attr);

......

/* 使用完之后 */
pthread_rwlock_destroy(&rwlock);      //销毁读写锁
pthread_rwlockattr_destroy(&attr);    //销毁读写锁属性对象
```

12.6 总结

本章介绍了线程同步的几种不同的方法，包括互斥锁、条件变量、自旋锁以及读写锁，当然，除此之外，线程同步的方法其实还有很多，譬如信号量、屏障等等，如果大家有兴趣可以自己查阅相关书籍进行学习。在实际应用开发当中，用的最多的还是互斥锁和条件变量，当然具体使用哪一种线程同步方法还是得根据场景来进行选择，方能达到事半功倍的效果！

第十三章 高级 I/O

本章再次回到文件 I/O 相关话题的讨论，将会介绍文件 I/O 当中的一些高级用法，以应对不同应用场合的需求，主要包括：非阻塞 I/O、I/O 多路复用、异步 I/O、存储映射 I/O 以及文件锁，我们统统把它们放到本章《高级 I/O》中进行讨论、学习。

本章将会讨论如下主题内容。

- 阻塞 I/O 与非阻塞 I/O；
- 阻塞 I/O 所带来的困境；
- 非阻塞 I/O 以轮训方式访问多个设备；
- 何为 I/O 多路复用以及原理；
- 何为异步 I/O 以及原理；
- 存储映射 I/O；
- 文件加锁。

13.1 非阻塞 I/O

关于“阻塞”一词前面已经给大家多次提到，阻塞其实就是进入了休眠状态，交出了 CPU 控制权。前面所学习过的函数，譬如 wait()、pause()、sleep()等函数都会进入阻塞，本小节来聊一聊关于阻塞式 I/O 与非阻塞式 I/O。

阻塞式 I/O 顾名思义就是对文件的 I/O 操作（读写操作）是阻塞式的，非阻塞式 I/O 同理就是对文件的 I/O 操作是非阻塞的。这样说大家可能不太明白，这里举个例子，譬如对于某些文件类型（读管道文件、网络设备文件和字符设备文件），当对文件进行读操作时，如果数据未准备好、文件当前无数据可读，那么读操作可能会使调用者阻塞，直到有数据可读时才会被唤醒，这就是阻塞式 I/O 常见的一种表现；如果是非阻塞式 I/O，即使没有数据可读，也不会被阻塞、而是会立马返回错误！

普通文件的读写操作是不会阻塞的，不管读写多少个字节数据，read()或 write()一定会在有限的时间内返回，所以普通文件一定是以非阻塞的方式进行 I/O 操作，这是普通文件本质上决定的；但是对于某些文件类型，譬如上面所介绍的管道文件、设备文件等，它们既可以使用阻塞式 I/O 操作，也可以使用非阻塞式 I/O 进行操作。

13.1.1 阻塞 I/O 与非阻塞 I/O 读文件

本小节我们将分别演示使用阻塞式 I/O 和非阻塞式 I/O 对文件进行读操作，在调用 open()函数打开文件时，为参数 flags 指定 O_NONBLOCK 标志，open()调用成功后，后续的 I/O 操作将以非阻塞式方式进行；这就是非阻塞 I/O 的打开方式，如果未指定 O_NONBLOCK 标志，则默认使用阻塞式 I/O 进行操作。

对于普通文件来说，指定与未指定 O_NONBLOCK 标志对其是没有影响，普通文件的读写操作是不会阻塞的，它总是以非阻塞的方式进行 I/O 操作，这是普通文件本质上决定的，前面已经给大家进行了说明。

本小节我们将以读取鼠标为例，使用两种 I/O 方式进行读取，来进行对比，鼠标是一种输入设备，其对应的设备文件在/dev/input/目录下，如下所示：

```
dt@dt-virtual-machine:/dev/input$ pwd
/dev/input
dt@dt-virtual-machine:/dev/input$
dt@dt-virtual-machine:/dev/input$ ls -lh
总用量 0
drwxr-xr-x 2 root root      80 1月   26 09:55 by-id
drwxr-xr-x 2 root root     140 1月   26 09:55 by-path
crw-rw---- 1 root input 13, 64 1月   26 09:55 event0
crw-rw---- 1 root input 13, 65 1月   26 09:55 event1
crw-rw---- 1 root input 13, 66 1月   26 09:55 event2
crw-rw---- 1 root input 13, 67 1月   26 09:55 event3
crw-rw---- 1 root input 13, 68 1月   26 09:55 event4
crw-rw---- 1 root input 13, 63 1月   26 09:55 mice
crw-rw---- 1 root input 13, 32 1月   26 09:55 mouse0
crw-rw---- 1 root input 13, 33 1月   26 09:55 mouse1
crw-rw---- 1 root input 13, 34 1月   26 09:55 mouse2
dt@dt-virtual-machine:/dev/input$
```

图 13.1.1 输入设备对应的设备文件

通常情况下是 mouseX（X 表示序号 0、1、2），但也不一定，也有可能是 eventX，如何确定到底是哪个设备文件，可以通过对设备文件进行读取来判断，譬如使用 od 命令：

```
sudo od -x /dev/input/event3
```

Tips：需要添加 sudo，在 Ubuntu 系统下，普通用户是无法对设备文件进行读取或写入操作。

当执行命令之后，移动鼠标或按下鼠标、松开鼠标都会在终端打印出相应的数据，如下所示：

```
dt@dt-virtual-machine:~$
dt@dt-virtual-machine:~$ sudo od -x /dev/input/event3
0000000 688b 607a 0000 0000 37df 000b 0000 0000
0000020 0003 0000 6e29 0000 688b 607a 0000 0000
0000040 37df 000b 0000 0000 0003 0001 4fab 0000
0000060 688b 607a 0000 0000 37df 000b 0000 0000
0000100 0000 0000 0000 0000 688b 607a 0000 0000
0000120 59bf 000b 0000 0000 0003 0000 6f18 0000
0000140 688b 607a 0000 0000 59bf 000b 0000 0000
0000160 0000 0000 0000 0000 688b 607a 0000 0000
0000200 96a7 000b 0000 0000 0003 0000 726f 0000
0000220 688b 607a 0000 0000 96a7 000b 0000 0000
0000240 0003 0001 4dbc 0000 688b 607a 0000 0000
0000260 96a7 000b 0000 0000 0000 0000 0000 0000
0000300 688b 607a 0000 0000 b895 000b 0000 0000
0000320 0003 0000 73c5 0000 688b 607a 0000 0000
0000340 b895 000b 0000 0000 0003 0001 4ce8 0000
0000360 688b 607a 0000 0000 b895 000b 0000 0000
0000400 0000 0000 0000 0000 688b 607a 0000 0000
0000420 da7b 000b 0000 0000 0003 0000 74d7 0000
0000440 688b 607a 0000 0000 da7b 000b 0000 0000
0000460 0003 0001 4c5b 0000 688b 607a 0000 0000
0000500 da7b 000b 0000 0000 0000 0000 0000 0000
0000520 688b 607a 0000 0000 fd38 000b 0000 0000
0000540 0003 0000 7560 0000 688b 607a 0000 0000
0000560 fd38 000b 0000 0000 0003 0001 4c14 0000
```

图 13.1.2 读取鼠标打印信息

如果没有打印信息，那么这个设备文件就不是鼠标对应的设备文件，那么就换一个设备文件再次测试，这样就会帮助你找到鼠标设备文件。笔者使用的 Ubuntu 系统，对应的鼠标设备文件是/dev/input/event3。接下来我们编写一个测试程序，使用阻塞式 I/O 读取鼠标。

示例代码 13.1.1 演示了以阻塞方式读取鼠标，调用 open()函数打开鼠标设备文件"/dev/input/event3"，以只读方式打开，没有指定 O_NONBLOCK 标志，说明使用的是阻塞式 I/O；程序中只调用了一次 read()读取鼠标。

示例代码 13.1.1 阻塞式 I/O 读取鼠标数据

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

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开文件 */
    fd = open("/dev/input/event3", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
```

```
    }

    /* 读文件 */
    memset(buf, 0, sizeof(buf));
    ret = read(fd, buf, sizeof(buf));
    if (0 > ret) {
        perror("read error");
        close(fd);
        exit(-1);
    }

    printf("成功读取<%d>个字节数据\n", ret);

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

编译上述示例代码进行测试：

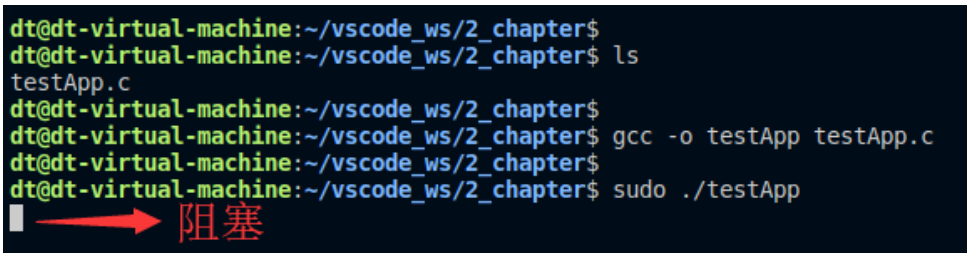


图 13.1.3 阻塞

执行程序之后，发现程序没有立即结束，而是一直占用了终端，没有输出信息，原因在于调用 read()之后进入了阻塞状态，因为当前鼠标没有数据可读；如果此时我们移动鼠标、或者按下鼠标上的任何一个按键，阻塞会结束，read()会成功读取到数据并返回，如下所示：

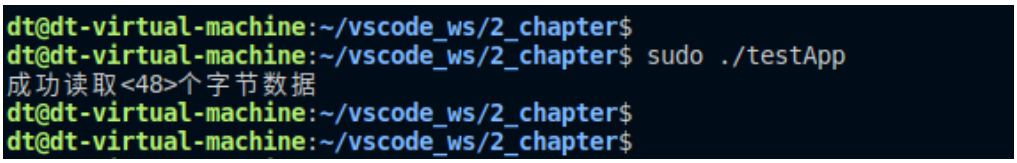


图 13.1.4 移动鼠标

打印信息提示，此次 read 成功读取了 48 个字节，程序当中我们明明要求读取的是 100 个字节，为什么这里只读取到了 48 个字节？这里暂时先不去理会这个问题。

接下来，我们将示例代码 13.1.1 修改成非阻塞式 I/O，如下所示：

示例代码 13.1.2 非阻塞式 I/O 读取鼠标数据

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

```
#include <string.h>

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开文件 */
    fd = open("/dev/input/event3", O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 读文件 */
    memset(buf, 0, sizeof(buf));
    ret = read(fd, buf, sizeof(buf));
    if (0 > ret) {
        perror("read error");
        close(fd);
        exit(-1);
    }

    printf("成功读取<%d>个字节数据\n", ret);

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

修改方法很简单，只需在调用 open()函数时指定 O_NONBLOCK 标志即可，对上述示例代码进行编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
read error: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.1.5 非阻塞

执行程序之后，程序立马就结束了，并且调用 read()返回错误，提示信息为"Resource temporarily unavailable"，意思就是说资源暂时不可用；原因在于调用 read()时，如果鼠标并没有移动或者被按下（没有发生输入事件），是没有数据可读，故而导致失败返回，这就是非阻塞 I/O。

可以对示例代码 13.1.2 进行修改，使用轮训方式不断地去读取，直到鼠标有数据可读，read()将会成功返回：

示例代码 13.1.3 轮训+非阻塞方式读取鼠标

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

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开文件 */
    fd = open("/dev/input/event3", O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 读文件 */
    memset(buf, 0, sizeof(buf));
    for ( ; ; ) {
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret) {
            printf("成功读取<%d>个字节数据\n", ret);
            close(fd);
            exit(0);
        }
    }
}
```

具体的执行的效果便不再演示了，各位读者自己动手试试。

13.1.2 阻塞 I/O 的优点与缺点

当对文件进行读取操作时，如果文件当前无数据可读，那么阻塞式 I/O 会将调用者应用程序挂起、进入休眠阻塞状态，直到有数据可读时才会解除阻塞；而对于非阻塞 I/O，应用程序不会被挂起，而是会立即返回，它要么一直轮训等待，直到数据可读，要么直接放弃！

所以阻塞式 I/O 的优点在于能够提升 CPU 的处理效率，当自身条件不满足时，进入阻塞状态，交出 CPU 资源，将 CPU 资源让给别人使用；而非阻塞式则是抓紧利用 CPU 资源，譬如不断地去轮训，这样就会导致该程序占用了非常高的 CPU 使用率！

执行示例代码 13.1.3 对应的程序时，通过 top 命令可以发现该程序的占用了非常高的 CPU 使用率，如下所示：

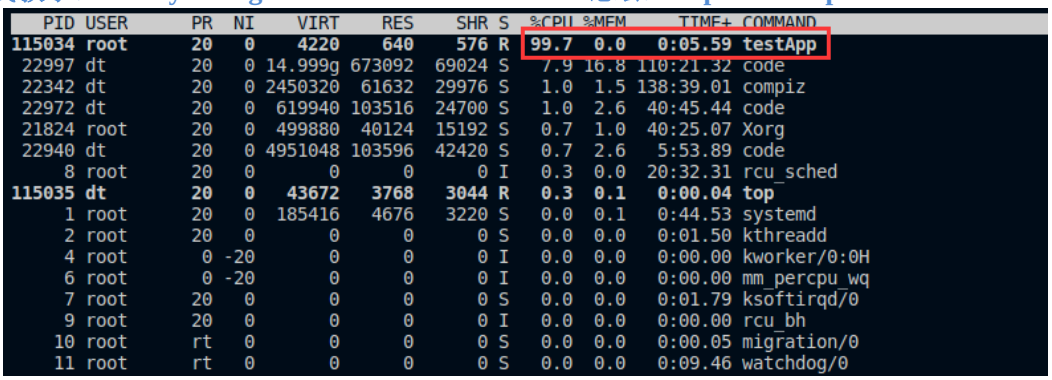


图 13.1.6 CPU 占用率

其 CPU 占用率几乎达到了 100%，在一个系统当中，一个进程的 CPU 占用率这么高是一件非常危险的事情。而示例代码 13.1.1 这种阻塞式方式，其 CPU 占用率几乎为 0，所以就本章的所举例子来说，阻塞式 I/O 绝地要优于非阻塞式 I/O，那既然如此，我们为何还要介绍非阻塞式 I/O 呢？下一小节我们将通过一个例子给大家介绍，阻塞式 I/O 的困境！

13.1.3 使用非阻塞 I/O 实现并发读取

上一小节给大家所举的例子当中，只读取了鼠标的数据，如果要在程序当中同时读取鼠标和键盘，那又该如何呢？本小节我们将分别演示使用阻塞式 I/O 和非阻塞式 I/O 同时读取鼠标和键盘；同理键盘也是一种输入类设备，但是键盘是标准输入设备 stdin，进程会自动从父进程中继承标准输入、标准输出以及标准错误，标准输入设备对应的文件描述符为 0，所以在程序当中直接使用即可，不需要再调用 open 打开。

首先我们使用阻塞式方式同时读取鼠标和键盘，示例代码如下所示：

示例代码 13.1.4 阻塞式同时读取鼠标和键盘

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

#define MOUSE           "/dev/input/event3"

int main(void)
{
    char buf[100];
    int fd, ret;

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }
```

```
    /* 读鼠标 */
    memset(buf, 0, sizeof(buf));
    ret = read(fd, buf, sizeof(buf));
    printf("鼠标: 成功读取<%d>个字节数据\n", ret);

    /* 读键盘 */
    memset(buf, 0, sizeof(buf));
    ret = read(0, buf, sizeof(buf));
    printf("键盘: 成功读取<%d>个字节数据\n", ret);

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

上述程序中先读了鼠标，在接着读键盘，所以由此可知，在实际测试当中，需要先动鼠标在按键盘（按下键盘上的按键、按完之后按下回车），这样才能既成功读取鼠标、又成功读取键盘，程序才能够顺利运行结束。因为 read 此时是阻塞式读取，先读取了鼠标，没有数据可读将会一直被阻塞，后面的读取键盘将得不到执行。

这就是阻塞式 I/O 的一个困境，无法实现并发读取（同时读取），主要原因在于阻塞，那如何解决这个问题呢？当然大家可能会想到使用多线程，一个线程读取鼠标、另一个线程读取键盘，亦或者创建一个子进程，父进程读取鼠标、子进程读取键盘等方法，当然这些方法自然可以解决，但不是我们要学习的重点。

既然阻塞 I/O 存在这样一个困境，那我们可以使用非阻塞式 I/O 解决它，将示例代码 13.1.4 修改为非阻塞式方式同时读取鼠标和键盘。使用 open()打开得到的文件描述符，调用 open()时指定 O_NONBLOCK 标志将其设置为非阻塞式 I/O；因为标准输入文件描述符（键盘）是从其父进程进程而来，并不是在我们的程序中调用 open()打开得到的，那如何将标准输入设置为非阻塞 I/O，可以使用 3.10.1 小节中给大家介绍的 fcntl()函数，具体使用方法在该小节中已有详细介绍，这里不再重述！可通过如下代码将标准输入（键盘）设置为非阻塞方式：

```
int flag;

flag = fcntl(0, F_GETFL);          //先获取原来的 flag
flag |= O_NONBLOCK;                //将 O_NONBLOCK 标志添加到 flag
fcntl(0, F_SETFL, flag);           //重新设置 flag
```

示例代码 13.1.5 演示了以非阻塞方式同时读取鼠标和键盘。

示例代码 13.1.5 非阻塞式方式同时读取鼠标和键盘

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

#define MOUSE          "/dev/input/event3"
```

```
int main(void)
{
    char buf[100];
    int fd, ret, flag;

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将键盘设置为非阻塞方式 */
    flag = fcntl(0, F_GETFL);     //先获取原来的 flag
    flag |= O_NONBLOCK;              //将 O_NONBLOCK 标准添加到 flag
    fcntl(0, F_SETFL, flag);      //重新设置 flag

    for ( ; ; ) {
        /* 读鼠标 */
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret)
            printf("鼠标: 成功读取<%d>个字节数据\n", ret);

        /* 读键盘 */
        ret = read(0, buf, sizeof(buf));
        if (0 < ret)
            printf("键盘: 成功读取<%d>个字节数据\n", ret);
    }

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

将读取鼠标和读取键盘操作放入到一个循环中，通过轮训方式来实现并发读取鼠标和键盘，对上述代码进行编译，测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
das
键盘： 成功读取<4>个字节数据
dasd
键盘： 成功读取<5>个字节数据
鼠标： 成功读取<48>个字节数据
鼠标： 成功读取<48>个字节数据
鼠标： 成功读取<48>个字节数据
鼠标： 成功读取<48>个字节数据
das
键盘： 成功读取<4>个字节数据
das
键盘： 成功读取<4>个字节数据
鼠标： 成功读取<48>个字节数据
鼠标： 成功读取<48>个字节数据
das
键盘： 成功读取<4>个字节数据
```

图 13.1.7 测试结果

这样就解决了示例代码 13.1.4 所出现的问题，不管是先动鼠标还是先按键盘都可以成功读取到相应数据。

虽然使用非阻塞 I/O 方式解决了示例代码 13.1.4 出现的问题，但由于程序当中使用轮训方式，故而会使得该程序的 CPU 占用率特别高，终归还是不太安全，会对整个系统产生很大的副作用，如何解决这样的问题呢？我们将在下一小节向大家介绍。

13.2 I/O 多路复用

上一小节虽然使用非阻塞式 I/O 解决了阻塞式 I/O 情况下并发读取文件所出现的问题，但依然不够完美，使得程序的 CPU 占用率特别高。解决这个问题，就要用到本小节将要介绍的 I/O 多路复用方法。

13.2.1 何为 I/O 多路复用

I/O 多路复用（IO multiplexing）它通过一种机制，可以监视多个文件描述符，一旦某个文件描述符（也就是某个文件）可以执行 I/O 操作时，能够通知应用程序进行相应的读写操作。I/O 多路复用技术是为了解决：在并发式 I/O 场景中进程或线程阻塞到某个 I/O 系统调用而出现的技术，使进程不阻塞于某个特定的 I/O 系统调用。

由此可知，I/O 多路复用一般用于并发式的非阻塞 I/O，也就是多路非阻塞 I/O，譬如程序中既要读取鼠标、又要读取键盘，多路读取。

我们可以采用两个功能几乎相同的系统调用来执行 I/O 多路复用操作，分别是系统调用 select()和 poll()。这两个函数基本是一样的，细节特征上存在些许差别！

I/O 多路复用存在一个非常明显的特征：外部阻塞式，内部监视多路 I/O。

13.2.2 select()函数介绍

系统调用 select()可用于执行 I/O 多路复用操作，调用 select()会一直阻塞，直到某一个或多个文件描述符成为就绪态（可以读或写）。其函数原型如下所示：

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

使用该函数需要包含头文件<sys/select.h>。

可以看出 select()函数的参数比较多，其中参数 readfds、writefds 以及 exceptfds 都是 fd_set 类型指针，指向一个 fd_set 类型对象，fd_set 数据类型是一个文件描述符的集合体，所以参数 readfds、writefds 以及 exceptfds 都是指向文件描述符集合的指针，这些参数按照如下方式使用：

- readfds 是用来检测读是否就绪（是否可读）的文件描述符集合；
- writefds 是用来检测写是否就绪（是否可写）的文件描述符集合；
- exceptfds 是用来检测异常情况是否发生的文件描述符集合。

Tips：异常情况并不是在文件描述符上出现了一些错误。

fd_set 数据类型是以位掩码的形式来实现的，但是，我们并不需要关心这些细节、无需关心该结构体成员信息，因为 Linux 提供了四个宏用于对 fd_set 类型对象进行操作，所有关于文件描述符集合的操作都是通过这四个宏来完成的：FD_CLR()、FD_ISSET()、FD_SET()、FD_ZERO()，稍后介绍！

如果对 readfds、writefds 以及 exceptfds 中的某些事件不感兴趣，可将其设置为 NULL，这表示对相应条件不关心。如果这三个参数都设置为 NULL，则可以将 select()当做为一个类似于 sleep()休眠的函数来使用，通过 select()函数的最后一个参数 timeout 来设置休眠时间。

select()函数的第一个参数 nfds 通常表示最大文件描述符编号值加 1，考虑 readfds、writefds 以及 exceptfds 这三个文件描述符集合，在 3 个描述符集中找出最大描述符编号值，然后加 1，这就是参数 nfds。

select()函数的最后一个参数 timeout 可用于设定 select()阻塞的时间上限，控制 select 的阻塞行为，可将 timeout 参数设置为 NULL，表示 select()将会一直阻塞、直到某一个或多个文件描述符成为就绪态；也可将其指向一个 struct timeval 结构体对象，该结构体在示例代码 5.6.3 有详细介绍，这里不再重述！

如果参数 timeout 指向的 struct timeval 结构体对象中的两个成员变量都为 0，那么此时 select()函数不会阻塞，它只是简单地轮训指定的文件描述符集合，看看其中是否有就绪的文件描述符并立刻返回。否则，参数 timeout 将为 select()指定一个等待（阻塞）时间的上限值，如果在阻塞期间内，文件描述符集合中的某一个或多个文件描述符成为就绪态，将会结束阻塞并返回；如果超过了阻塞时间的上限值，select()函数将会返回！

select()函数将阻塞知道有以下事情发生：

- readfds、writefds 或 exceptfds 指定的文件描述符中至少有一个称为就绪态；
- 该调用被信号处理函数中断；
- 参数 timeout 中指定的时间上限已经超时。

FD_CLR()、FD_ISSET()、FD_SET()、FD_ZERO()

文件描述符集合的所有操作都可以通过这四个宏来完成，这些宏定义如下所示：

```
#include <sys/select.h>

void FD_CLR(int fd, fd_set *set);
int   FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

这些宏按照如下方式工作：

- FD_ZERO()将参数 set 所指向的集合初始化为空；
- FD_SET()将文件描述符 fd 添加到参数 set 所指向的集合中；
- FD_CLR()将文件描述符 fd 从参数 set 所指向的集合中移除；
- 如果文件描述符 fd 是参数 set 所指向的集合中的成员，则 FD_ISSET()返回 true，否则返回 false。

文件描述符集合有一个最大容量限制，有常量 FD_SETSIZE 来决定，在 Linux 系统下，该常量的值为 1024。在定义一个文件描述符集合之后，必须用 FD_ZERO()宏将其进行初始化操作，然后再向集合中添加我们关心的各个文件描述符，例如：

```
fd_set fset;            //定义文件描述符集合

FD_ZERO(&fset);         //将集合初始化为空
FD_SET(3, &fset); //向集合中添加文件描述符 3
FD_SET(4, &fset); //向集合中添加文件描述符 4
FD_SET(5, &fset); //向集合中添加文件描述符 5
```

在调用 select()函数之后，select()函数内部会修改 readfds、writefds、exceptfds 这些集合，当 select()函数返回时，它们包含的就是已处于就绪态的文件描述符集合了。譬如在调用 select()函数之前，readfds 所指向的集合中包含了 3、4、5 这三个文件描述符，当调用 select()函数之后，假设 select()返回时，只有文件描述符 4 已经处于就绪态了，那么此时 readfds 指向的集合中就只包含了文件描述符 4。所以由此可知，如果要在循环中重复调用 select()，我们必须保证每次都要重新初始化并设置 readfds、writefds、exceptfds 这些集合。

select()函数的返回值

select()函数有三种可能的返回值，会返回如下三种情况中的一种：

- 返回-1 表示有错误发生，并且会设置 errno。可能的错误码包括 EBADF、EINTR、EINVAL、EINVAL 以及 ENOMEM，EBADF 表示 readfds、writefds 或 exceptfds 中有一个文件描述符是非法的；EINTR 表示该函数被信号处理函数中断了，其它错误大家可以自己去看，在 man 手册都有相信的记录。
- 返回 0 表示在任何文件描述符成为就绪态之前 select()调用已经超时，在这种情况下，readfds，writefds 以及 exceptfds 所指向的文件描述符集合都会被清空。
- 返回一个正整数表示有一个或多个文件描述符已达到就绪态。返回值表示处于就绪态的文件描述符的个数，在这种情况下，每个返回的文件描述符集合都需要检查，通过 FD_ISSET()宏进行检查，以此找出发生的 I/O 事件是什么。如果同一个文件描述符在 readfds，writefds 以及 exceptfds 中同时被指定，且它多于多个 I/O 事件都处于就绪态的话，那么就会被统计多次，换句话说，select()返回三个集合中被标记为就绪态的文件描述符的总数。

使用示例

示例代码 13.2.1 演示了使用 select()函数来实现 I/O 多路复用操作，同时读取键盘和鼠标。程序中将鼠标和键盘配置为非阻塞 I/O 方式，本程序对数据进行了 5 次读取，通过 while 循环来实现。由于在 while 循环中会重复调用 select()函数，所以每次调用之前需要对 rdfds 进行初始化以及添加鼠标和键盘对应的文件描述符。

该程序中，select()函数的参数 timeout 被设置为 NULL，并且我们只关心鼠标或键盘是否有数据可读，所以将参数 writefds 和 exceptfds 也设置为 NULL。执行 select()函数时，如果鼠标和键盘均无数据可读，则 select()调用会陷入阻塞，直到发生输入事件（鼠标移动、键盘上的按键按下或松开）才会返回。

示例代码 13.2.1 使用 select 实现同时读取键盘和鼠标

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

#define MOUSE           "/dev/input/event3"
```

```
int main(void)
{
    char buf[100];
    int fd, ret = 0, flag;
    fd_set rdfds;
    int loops = 5;

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将键盘设置为非阻塞方式 */
    flag = fcntl(0, F_GETFL);     //先获取原来的 flag
    flag |= O_NONBLOCK;           //将 O_NONBLOCK 标准添加到 flag
    fcntl(0, F_SETFL, flag);      //重新设置 flag

    /* 同时读取键盘和鼠标 */
    while (loops--) {
        FD_ZERO(&rdfds);
        FD_SET(0, &rdfds);   //添加键盘
        FD_SET(fd, &rdfds); //添加鼠标

        ret = select(fd + 1, &rdfds, NULL, NULL, NULL);
        if (0 > ret) {
            perror("select error");
            goto out;
        }
        else if (0 == ret) {
            fprintf(stderr, "select timeout.\n");
            continue;
        }

        /* 检查键盘是否为就绪态 */
        if(FD_ISSET(0, &rdfds)) {
            ret = read(0, buf, sizeof(buf));
            if (0 < ret)
                printf("键盘: 成功读取<%d>个字节数据\n", ret);
        }

        /* 检查鼠标是否为就绪态 */
```

```
        if(FD_ISSET(fd, &rdfds)) {
            ret = read(fd, buf, sizeof(buf));
            if (0 < ret)
                printf("鼠标: 成功读取<%d>个字节数据\n", ret);
        }
    }

out:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

程序中分析 select()函数的返回值 ret，只有当 ret 大于 0 时才表示有文件描述符处于就绪态，并将这些处于就绪态的文件描述符通过 rdfds 集合返回出来，程序中使用 FD_ISSET()宏检查返回的 rdfds 集合中是否包含鼠标文件描述符以及键盘文件描述符，如果包含则表示可以读取数据了。

编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码：
das
select ret = 1
键盘：成功读取<4>个字节数据
das
select ret = 1
键盘：成功读取<4>个字节数据
select ret = 1
鼠标：成功读取<48>个字节数据
select ret = 1
鼠标：成功读取<48>个字节数据
sa
select ret = 1
键盘：成功读取<3>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.2.1 测试结果

示例代码 13.2.1 将鼠标和键盘都设置为了非阻塞 I/O 方式，其实设置为阻塞 I/O 方式也是可以的，因为 select()返回时意味着此时数据是可读取的，所以以非阻塞和阻塞两种方式读取数据均不会发生阻塞。

13.2.3 poll()函数介绍

系统调用 poll()与 select()函数很相似，但函数接口有所不同。在 select()函数中，我们提供三个 fd_set 集合，在每个集合中添加我们关心的文件描述符；而在 poll()函数中，则需要构造一个 struct pollfd 类型的数组，每个数组元素指定一个文件描述符以及我们对该文件描述符所关心的条件（数据可读、可写或异常情况）。poll()函数原型如下所示：

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

使用该函数需要包含头文件<poll.h>。

函数参数含义如下：

fds：指向一个 struct pollfd 类型的数组，数组中的每个元素都会指定一个文件描述符以及我们对该文件描述符所关心的条件，稍后介绍 struct pollfd 结构体类型。

nfds：参数 nfds 指定了 fds 数组中的元素个数，数据类型 nfds_t 实际为无符号整形。

timeout：该参数与 select()函数的 timeout 参数相似，用于决定 poll()函数的阻塞行为，具体用法如下：

- 如果 timeout 等于-1，则 poll()会一直阻塞（与 select()函数的 timeout 等于 NULL 相同），直到 fds 数组中列出的文件描述符有一个达到就绪态或者捕获到一个信号时返回。
- 如果 timeout 等于 0，poll()不会阻塞，只是执行一次检查看看哪个文件描述符处于就绪态。
- 如果 timeout 大于 0，则表示设置 poll()函数阻塞时间的上限值，意味着 poll()函数最多阻塞 timeout 毫秒，直到 fds 数组中列出的文件描述符有一个达到就绪态或者捕获到一个信号为止。

struct pollfd 结构体

struct pollfd 结构体如下所示：

示例代码 13.2.2 struct pollfd 结构体

```
struct pollfd {
    int     fd;         /* file descriptor */
    short events;       /* requested events */
    short revents;      /* returned events */
};
```

fd 是一个文件描述符，struct pollfd 结构体中的 events 和 revents 都是位掩码，调用者初始化 events 来指定需要为文件描述符 fd 做检查的事件。当 poll()函数返回时，revents 变量由 poll()函数内部进行设置，用于说明文件描述符 fd 发生了哪些事件（注意，poll()没有更改 events 变量），我们可以对 revents 进行检查，判断文件描述符 fd 发生了什么事件。

应将每个数组元素的 events 成员设置为表 13.2.1 中所示的一个或几个标志，多个标志通过位或运算符(|)组合起来，通过这些值告诉内核我们关心的是该文件描述符的哪些事件。同样，返回时，revents 变量由内核设置为表 13.2.1 中所示的一个或几个标志。

表 13.2.1 poll 的 events 和 revents 标志

标志名	输入至 events	从 revents 得到结果	说明
POLLIN	●	●	有数据可以读取
POLLRDNORM	●	●	相等于 POLLIN
POLLRDBAND	●	●	可以读取优先级数据（Linux 上通常不使用）
POLLPRI	●	●	可读取高优先级数据
POLLRDHUP	●	●	对端套接字关闭
POLLOUT	●	●	可写入数据
POLLWRNORM	●	●	相等于 POLLOUT
POLLWRBAND	●	●	优先级数据可写入
POLLERR		●	有错误发生
POLLHUP		●	出现挂断
POLLNVAL		●	文件描述符未打开
POLLMSG			Linux 中不使用

表 13.2.1 中第一组标志（POLLIN、POLLRDNORM、POLLRDBAND、POLLPRI、POLLRDHUP）与数据可读相关；第二组标志（POLLOUT、POLLWRNORM、POLLWRBAND）与可写数据相关；而第三组

标志（POLLERR、POLLHUP、POLLNVAL）是设定在 revents 变量中用来返回有关文件描述符的附加信息，如果在 events 变量中指定了这三个标志，则会被忽略。

如果我们对某个文件描述符上的事件不感兴趣，则可将 events 变量设置为 0；另外，将 fd 变量设置为文件描述符的负值（取文件描述符 fd 的相反数-fd），将导致对应的 events 变量被 poll()忽略，并且 revents 变量将总是返回 0，这两种方法都可用来关闭对某个文件描述符的检查。

在实际应用编程中，一般用的最多的还是 POLLIN 和 POLLOUT。对于其它标志这里不再进行介绍了，后面章节内容中，如果需要使用时再给大家介绍！

poll()函数返回值

poll()函数返回值含义与 select()函数的返回值是一样的，有如下几种情况：

- 返回-1 表示有错误发生，并且会设置 errno。
- 返回 0 表示该调用在任意一个文件描述符成为就绪态之前就超时了。
- 返回一个正整数表示有一个或多个文件描述符处于就绪态了，返回值表示 fds 数组中返回的 revents 变量不为 0 的 struct pollfd 对象的数量。

使用示例

示例代码 13.2.3 演示了使用 poll()函数来实现 I/O 多路复用操作，同时读取键盘和鼠标。其实就是将示例代码 13.2.1 进行了修改，使用 poll 替换 select。

示例代码 13.2.3 使用 poll 实现同时读取鼠标和键盘

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

#define MOUSE           "/dev/input/event3"

int main(void)
{
    char buf[100];
    int fd, ret = 0, flag;
    int loops = 5;
    struct pollfd fds[2];

    /* 打开鼠标设备文件 */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将键盘设置为非阻塞方式 */
```

```
    flag = fcntl(0, F_GETFL);     //先获取原来的 flag
    flag |= O_NONBLOCK;             //将 O_NONBLOCK 标准添加到 flag
    fcntl(0, F_SETFL, flag);      //重新设置 flag

    /* 同时读取键盘和鼠标 */
    fds[0].fd = 0;
    fds[0].events = POLLIN; //只关心数据可读
    fds[0].revents = 0;
    fds[1].fd = fd;
    fds[1].events = POLLIN; //只关心数据可读
    fds[1].revents = 0;

    while (loops--) {
        ret = poll(fds, 2, -1);
        if (0 > ret) {
            perror("poll error");
            goto out;
        }
        else if (0 == ret) {
            fprintf(stderr, "poll timeout.\n");
            continue;
        }

        /* 检查键盘是否为就绪态 */
        if(fds[0].revents & POLLIN) {
            ret = read(0, buf, sizeof(buf));
            if (0 < ret)
                printf("键盘: 成功读取<%d>个字节数据\n", ret);
        }

        /* 检查鼠标是否为就绪态 */
        if(fds[1].revents & POLLIN) {
            ret = read(fd, buf, sizeof(buf));
            if (0 < ret)
                printf("鼠标: 成功读取<%d>个字节数据\n", ret);
        }
    }

out:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

struct pollfd 结构体的 events 变量和 revents 变量都是位掩码，所以可以使用"revents & POLLIN"按位与的方式来检查是否发生了相应的 POLLIN 事件，判断鼠标或键盘数据是否可读。测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码：
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
dasdasdasd
键盘：成功读取<11>个字节数据
hedasdas
键盘：成功读取<9>个字节数据
鼠标：成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.2.2 测试结果

13.2.4 总结

在使用 select()或 poll()时需要注意一个问题，当监测到某一个或多个文件描述符成为就绪态（可以读或写）时，需要执行相应的 I/O 操作，以清除该状态，否则该状态将会一直存在；譬如示例代码 13.2.1 中，调用 select()函数监测鼠标和键盘这两个文件描述符，当 select()返回时，通过 FD_ISSET()宏判断文件描述符上是否可执行 I/O 操作；如果可以执行 I/O 操作时，应在应用程序中对该文件描述符执行 I/O 操作，以清除文件描述符的就绪态，如果不清除就绪态，那么该状态将会一直存在，那么下一次调用 select()时，文件描述符已经处于就绪态了，将直接返回。

同理对于 poll()函数来说亦是如此，譬如示例代码 13.2.3，当 poll()成功返回时，检查文件描述符是否称为就绪态，如果文件描述符上可执行 I/O 操作时，也需要对文件描述符执行 I/O 操作，以清除就绪状态。

13.3 异步 IO

在 I/O 多路复用中，进程通过系统调用 select()或 poll()来主动查询文件描述符上是否可以执行 I/O 操作。而在异步 I/O 中，当文件描述符上可以执行 I/O 操作时，进程可以请求内核为自己发送一个信号。之后进程就可以执行任何其它的任务直到文件描述符可以执行 I/O 操作为止，此时内核会发送信号给进程。所以要使用异步 I/O，还得结合前面所学习的信号相关的内容，所以异步 I/O 通常也称为信号驱动 I/O。

要使用异步 I/O，程序需要按照如下步骤来执行：

- 通过指定 O_NONBLOCK 标志使能非阻塞 I/O。
- 通过指定 O_ASYNC 标志使能异步 I/O。
- 设置异步 I/O 事件的接收进程。也就是当文件描述符上可执行 I/O 操作时会发送信号通知该进程，通常将调用进程设置为异步 I/O 事件的接收进程。
- 为内核发送的通知信号注册一个信号处理函数。默认情况下，异步 I/O 的通知信号是 SIGIO，所以内核会给进程发送信号 SIGIO。在 8.2 小节中简单地提到过该信号。
- 以上步骤完成之后，进程就可以执行其它任务了，当 I/O 操作就绪时，内核会向进程发送一个 SIGIO 信号，当进程接收到信号时，会执行预先注册好的信号处理函数，我们就可以在信号处理函数中进行 I/O 操作。

O_ASYNC 标志

O_ASYNC 标志可用于使能文件描述符的异步 I/O 事件，当文件描述符可执行 I/O 操作时，内核会向异步 I/O 事件的接收进程发送 SIGIO 信号（默认情况下）。在 2.3 小节介绍 open()函数时，给大家提到过该标志，但并未介绍该标志的作用，该标志主要用于异步 I/O。

需要注意的是：在调用 open()时无法通过指定 O_ASYNC 标志来使能异步 I/O，但可以使用 fcntl()函数添加 O_ASYNC 标志使能异步 I/O，譬如：

```
int flag;

flag = fcntl(0, F_GETFL);          //先获取原来的 flag
flag |= O_ASYNC;                   //将 O_ASYNC 标志添加到 flag
fcntl(fd, F_SETFL, flag);          //重新设置 flag
```

设置异步 I/O 事件的接收进程

为文件描述符设置异步 I/O 事件的接收进程，也就是设置异步 I/O 的所有者。同样也是通过 fcntl()函数进行设置，操作命令 cmd 设置为 F_SETOWN，第三个参数传入接收进程的进程 ID（PID），通常将调用进程的 PID 传入，譬如：

```
fcntl(fd, F_SETOWN, getpid());
```

注册 SIGIO 信号的处理函数

通过 signal()或 sigaction()函数为 SIGIO 信号注册一个信号处理函数，当进程接收到内核发送过来的 SIGIO 信号时，会执行该处理函数，所以我们应该在处理函数当中执行相应的 I/O 操作。

使用示例

示例代码 13.3.1 演示了以异步 I/O 方式读取鼠标，当进程接收到 SIGIO 信号时，执行信号处理函数 sigio_handler()，在该函数中调用 read()读取鼠标数据。

示例代码 13.3.1 以异步 I/O 方式读取鼠标

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

#define MOUSE          "/dev/input/event3"
static int fd;

static void sigio_handler(int sig)
{
    static int loops = 5;
    char buf[100] = {0};
    int ret;

    if(SIGIO != sig)
        return;
```

```
    ret = read(fd, buf, sizeof(buf));
    if (0 < ret)
        printf("鼠标: 成功读取<%d>个字节数据\n", ret);

    loops--;
    if (0 >= loops) {
        close(fd);
        exit(0);
    }
}

int main(void)
{
    int flag;

    /* 打开鼠标设备文件<使能非阻塞 I/O> */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 使能异步 I/O */
    flag = fcntl(fd, F_GETFL);
    flag |= O_ASYNC;
    fcntl(fd, F_SETFL, flag);

    /* 设置异步 I/O 的所有者 */
    fcntl(fd, F_SETOWN, getpid());

    /* 为 SIGIO 信号注册信号处理函数 */
    signal(SIGIO, sigio_handler);

    for ( ; ; )
        sleep(1);
}
```

代码比较简单，这里我们进行编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.3.1 测试结果

13.4 优化异步 I/O

上一小节介绍了异步 I/O 的原理以及使用方法，在一个需要同时检查大量文件描述符（譬如数千个）的应用程序中，例如某种类型的网络服务端程序，与 select()和 poll()相比，异步 I/O 能够提供显著的性能优势。之所以如此，原因在于：对于异步 I/O，内核可以“记住”要检查的文件描述符，且仅当这些文件描述符上可执行 I/O 操作时，内核才会向应用程序发送信号。

而对于 select()或 poll()函数来说，内部实现原理其实是通过轮训的方式来检查多个文件描述符是否可执行 I/O 操作，所以，当需要检查的文件描述符数量较多时，随之也将会消耗大量的 CPU 资源来实现轮训检查操作。当需要检查的文件描述符并不是很多时，使用 select()或 poll()是一种非常不错的方案！

Tips：当需要检查大量文件描述符时，可以使用 epoll 解决 select()或 poll()性能低的问题，本书并不会介绍 epoll 相关内容，如果读者有兴趣可以自行查阅书籍进行学习。在性能表现上，epoll 与异步 I/O 方式相似，但是 epoll 有一些胜过异步 I/O 的优点。

不管是异步 I/O、还是 epoll，在需要检查大量文件描述符的应用程序当中，在这种情况下，它们的性能相比于 select()或 poll()有着显著的优势！

本小节将对上一小节所讲述的异步 I/O 进行优化，既然要对其进行优化，那必然存在着一些缺陷，如下所示：

- 默认的异步 I/O 通知信号 SIGIO 是非排队信号。SIGIO 信号是标准信号（非实时信号、不可靠信号），所以它不支持信号排队机制，譬如当前正在执行 SIGIO 信号的处理函数，此时内核又发送多次 SIGIO 信号给进程，这些信号将会被阻塞，只有当信号处理函数执行完毕之后才会传递给进程，并且只能传递一次，而其它后续的信号都会丢失。
- 无法得知文件描述符发生了什么事件。在示例代码 13.3.1 的信号处理函数 sigio_handler()中，直接调用了 read()函数读取鼠标，而并未判断文件描述符是否处于可读就绪态，事实上，示例代码 13.3.1 这种异步 I/O 方式并未告知应用程序文件描述符上发生了什么事件，是可读取还是可写入亦或者发生异常等。

所以本小节我们将会针对以上列举出的两个缺陷进行优化。

13.4.1 使用实时信号替换默认信号 SIGIO

SIGIO 作为异步 I/O 通知的默认信号，是一个非实时信号，我们可以设置不使用默认信号，指定一个实时信号作为异步 I/O 通知信号，如何指定呢？同样也是使用 fcntl()函数进行设置，调用函数时将操作命令 cmd 参数设置为 F_SETSIG，第三个参数 arg 指定一个实时信号编号即可，表示将该信号作为异步 I/O 通知信号，譬如：

```
fcntl(fd, F_SETSIG, SIGRTMIN);
```

上述代码指定了 SIGRTMIN 实时信号作为文件描述符 fd 的异步 I/O 通知信号，而不再使用默认的 SIGIO 信号。当文件描述符 fd 可执行 I/O 操作时，内核会发送实时信号 SIGRTMIN 给调用进程。

如果第三个参数 arg 设置为 0，则表示指定 SIGIO 信号作为异步 I/O 通知信号，也就是回到了默认状态。

13.4.2 使用 sigaction()函数注册信号处理函数

在应用程序当中需要为实时信号注册信号处理函数，使用 sigaction 函数进行注册，并为 sa_flags 参数指定 SA_SIGINFO，表示使用 sa_sigaction 指向的函数作为信号处理函数，而不使用 sa_handler 指向的函数。因为 sa_sigaction 指向的函数作为信号处理函数提供了更多的参数，可以获取到更多信息，函数定义参考示例代码 8.4.2 中关于 struct sigaction 结构体的描述。

函数参数中包括一个 siginfo_t 指针，指向 siginfo_t 类型对象，当触发信号时该对象由内核构建。siginfo_t 结构体中提供了很多信息，我们可以在信号处理函数中使用这些信息，具体定义请参考示例代码 8.4.3，就对于异步 I/O 事件而言，传递给信号处理函数的 siginfo_t 结构体中与之相关的字段如下：

- si_signo：引发处理函数被调用的信号。这个值与信号处理函数的第一个参数一致。
- si_fd：表示发生异步 I/O 事件的文件描述符；
- si_code：表示文件描述符 si_fd 发生了什么事件，读就绪态、写就绪态或者是异常事件等。该字段中可能出现的值以及它们对应的描述信息参见表 13.4.1。
- si_band：是一个位掩码，其中包含的值与系统调用 poll()中返回的 revents 字段中的值相同。如表 13.4.1 所示，si_code 中可能出现的值与 si_band 中的位掩码有着一一对应关系。

表 13.4.1 siginfo_t 结构体中的 si_code 和 si_band 的可能值

si_code	si_band 掩码值	描述/说明
POLL_IN	POLLIN | POLLRDNORM	可读取数据
POLL_OUT	POLLOUT | POLLWRNORM | POLLWRBAND	可写入数据
POLL_MSG	POLLIN | POLLRDNORM | POLLMSG	不使用
POLL_ERR	POLLERR	I/O 错误
POLL_PRI	POLLPRI | POLLRDNORM	可读取高优先级数据
POLL_HUP	POLLHUP | POLLERR	出现宕机

所以，由此可知，可以在信号处理函数中通过对比 siginfo_t 结构体的 si_code 变量来检查文件描述符发生了什么事件，以采取相应的 I/O 操作。

13.4.3 使用示例

通过 13.4.1 小节和 13.4.2 小节的学习，我们已经知道了如何针对 13.4 小节开头提出的异步 I/O 存在的两个缺陷进行优化。示例代码 13.4.1 是对示例代码 13.3.1 进行了优化，使用实时信号+sigaction 解决：默认异步 I/O 通知信号 SIGIO 可能存在丢失以及信号处理函数中无法判断文件描述符所发生的 I/O 事件这两个问题。

调用 sigaction()注册信号处理函数时，sa_flags 指定了 SA_SIGINFO，所以将使用 sa_sigaction 指向的函数 io_handler 作为信号处理函数，io_handler 共有 3 个参数，参数 sig 等于引发信号处理函数被调用的信号值，参数 info 附加了很多信息，前面已有介绍，这里不再重述。

示例代码 13.4.1 读取鼠标--优化异步 I/O

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

```
#include <unistd.h>
#include <signal.h>

#define MOUSE           "/dev/input/event3"
static int fd;

static void io_handler(int sig,
                       siginfo_t *info,
                       void *context)
{
    static int loops = 5;
    char buf[100] = {0};
    int ret;

    if(SIGRTMIN != sig)
        return;

    /* 判断鼠标是否可读 */
    if (POLL_IN == info->si_code) {
        ret = read(fd, buf, sizeof(buf));
        if (0 < ret)
            printf("鼠标: 成功读取<%d>个字节数据\n", ret);

        loops--;
        if (0 >= loops) {
            close(fd);
            exit(0);
        }
    }
}

int main(void)
{
    struct sigaction act;
    int flag;

    /* 打开鼠标设备文件<使能非阻塞 I/O> */
    fd = open(MOUSE, O_RDONLY | O_NONBLOCK);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }
```

```
    /* 使能异步 I/O */
    flag = fcntl(fd, F_GETFL);
    flag |= O_ASYNC;
    fcntl(fd, F_SETFL, flag);

    /* 设置异步 I/O 的所有者 */
    fcntl(fd, F_SETOWN, getpid());

    /* 指定实时信号 SIGRTMIN 作为异步 I/O 通知信号 */
    fcntl(fd, F_SETSIG, SIGRTMIN);

    /* 为实时信号 SIGRTMIN 注册信号处理函数 */
    act.sa_sigaction = io_handler;
    act.sa_flags = SA_SIGINFO;
    sigemptyset(&act.sa_mask);
    sigaction(SIGRTMIN, &act, NULL);

    for ( ; ; )
        sleep(1);
}
```

对上述示例代码进行编译时，出现了一些报错信息，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
testApp.c: In function 'main':
testApp.c:58:12: error: 'F_SETSIG' undeclared (first use in this function)
  fcntl(fd, F_SETSIG, SIGRTMIN);
            ^
testApp.c:58:12: note: each undeclared identifier is reported only once for each function it appears in
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.4.1 编译报错

报错提示没有定义F_SETSIG，确实如此，我们需要定义了_GNU_SOURCE宏之后才能使用F_SETSIG，这个宏在 4.9.3 小节向大家介绍过，这里不再重述！

这里笔者选择直接在源文件中使用#define 定义_GNU_SOURCE 宏，如下所示：

```
#define _GNU_SOURCE          //在源文件开头定义_GNU_SOURCE 宏

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

再次进行编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -D_GNU_SOURCE -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ sudo ./testApp
[sudo] dt 的密码：
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
鼠标：成功读取<48>个字节数据
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.4.2 测试结果

13.5 存储映射 I/O

存储映射 I/O（memory-mapped I/O）是一种基于内存区域的高级 I/O 操作，它能将一个文件映射到进程地址空间中的一块内存区域中，当从这段内存中读数据时，就相当于读文件中的数据（对文件进行 read 操作），将数据写入这段内存时，则相当于将数据直接写入文件中（对文件进行 write 操作）。这样就可以在不使用基本 I/O 操作函数 read()和 write()的情况下执行 I/O 操作。

13.5.1 mmap()和 munmap()函数

为了实现存储映射 I/O 这一功能，我们需要告诉内核将一个给定的文件映射到进程地址空间中的一块内存区域中，这由系统调用 mmap()来实现。其函数原型如下所示：

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

使用该函数需要包含头文件<sys/mman.h>。

函数参数和返回值含义如下：

addr：参数 addr 用于指定映射到内存区域的起始地址。通常将其设置为 NULL，这表示由系统选择该映射区的起始地址，这是最常见的设置方式；如果参数 addr 不为 NULL，则表示由自己指定映射区的起始地址，此函数的返回值是该映射区的起始地址。

length：参数 length 指定映射长度，表示将文件中的多大部分映射到内存区域中，以字节为单位，譬如 length=1024 * 4，表示将文件的 4K 字节大小映射到内存区域中。

offset：文件映射的偏移量，通常将其设置为 0，表示从文件头部开始映射；所以参数 offset 和参数 length 就确定了文件的起始位置和长度，将文件的这部分映射到内存区域中，如图 13.5.1 所示。

fd：文件描述符，指定要映射到内存区域中的文件。

prot：参数 prot 指定了映射区的保护要求，可取值如下：

- **PROT_EXEC：**映射区可执行；
- **PROT_READ：**映射区可读；
- **PROT_WRITE：**映射区可写；
- **PROT_NONE：**映射区不可访问。

可将 prot 指定为为 PROT_NONE，也可将其设置为 PROT_EXEC、PROT_READ、PROT_WRITE 中一个或多个（通过按位或运算符任意组合）。对指定映射区的保护要求不能超过文件 open()时的访问权限，譬如，文件是以只读权限方式打开的，那么对映射区的不能指定为 PROT_WRITE。

flags：参数 flags 可影响映射区的多种属性，参数 flags 必须要指定以下两种标志之一：

- **MAP_SHARED**：此标志指定当对映射区写入数据时，数据会写入到文件中，也就是会将写入到映射区中的数据更新到文件中，并且允许其它进程共享。
- **MAP_PRIVATE**：此标志指定当对映射区写入数据时，会创建映射文件的一个私人副本（copy-on-write），对映射区的任何操作都不会更新到文件中，仅仅只是对文件副本进行读写。

除此之外，还可将以下标志中的 0 个或多个组合到参数 flags 中，通过按位或运算符进行组合：

- **MAP_FIXED**：在未指定该标志的情况下，如果参数 addr 不等于 NULL，表示由调用者自己指定映射区的起始地址，但这只是一种建议、而并非强制，所以内核并不会保证使用参数 addr 指定的值作为映射区的起始地址；如果指定了 MAP_FIXED 标志，则表示要求必须使用参数 addr 指定的值作为起始地址，如果使用指定值无法成功建立映射时，则放弃！通常，不建议使用此标志，因为这不利于移植。
- **MAP_ANONYMOUS**：建立匿名映射，此时会忽略参数 fd 和 offset，不涉及文件，而且映射区域无法和其它进程共享。
- **MAP_ANON**：与 MAP_ANONYMOUS 标志同义，不建议使用。
- **MAP_DENYWRITE**：该标志被忽略。
- **MAP_EXECUTABLE**：该标志被忽略。
- **MAP_FILE**：兼容性标志，已被忽略。
- **MAP_LOCKED**：对映射区域进行上锁。

除了以上标志之外，还有其它一些标志，这里便不再介绍，可通过 man 手册进行查看。在众多标志当中，通常情况下，参数 flags 中只指定了 MAP_SHARED。

返回值：成功情况下，函数的返回值便是映射区的起始地址；发生错误时，返回(void *)-1，通常使用 MAP_FAILED 来表示，并且会设置 errno 来指示错误原因。

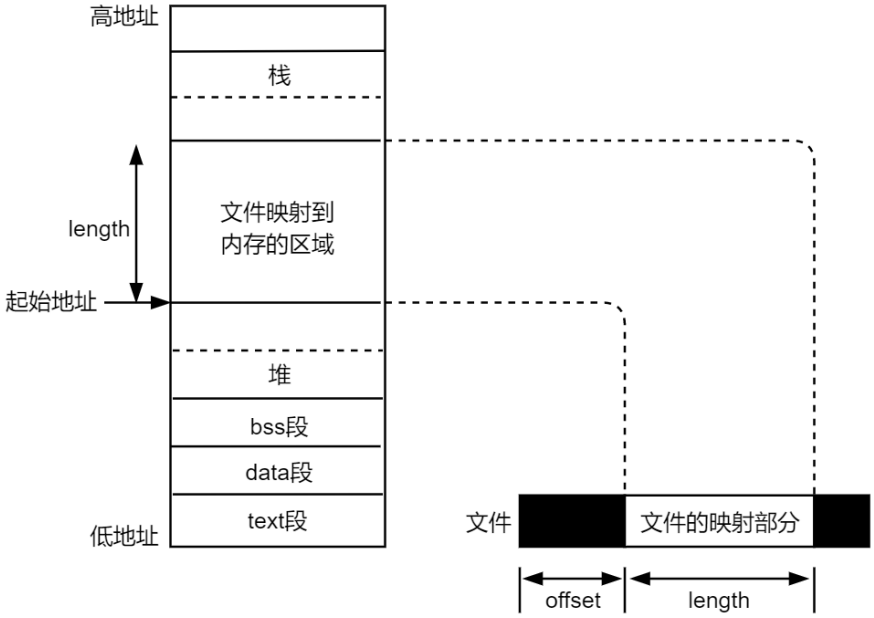


图 13.5.1 存储映射 I/O 示意图

对于 mmap()函数，参数 addr 和 offset 在不为 NULL 和 0 的情况下，addr 和 offset 的值通常被要求是系统页大小的整数倍，可通过 sysconf()函数获取页大小，如下所示（以字节为单位）：

```
sysconf(_SC_PAGE_SIZE)
或
sysconf(_SC_PAGESIZE)
```

虽然对 addr 和 offset 有这种限制，但对于参数 length 长度来说，却没有这种要求，如果映射区的长度不是页长度的整数倍时，会怎么样呢？对于这个问题的答案，我们首先需要了解到，对于 mmap()函数来说，当文件成功被映射到内存区域时，这段内存区域（映射区）的大小通常是页大小的整数倍，即使参数 length 并不是页大小的整数倍。如果文件大小为 96 个字节，我们调用 mmap()时参数 length 也是设置为 96，假设系统页大小为 4096 字节（4K），则系统通常会提供 4096 个字节的映射区，其中后 4000 个字节会被设置为 0，可以修改后面的这 4000 个字节，但是并不会影响到文件。但如果访问 4000 个字节后面的内存区域，将会导致异常情况发生，产生 SIGBUS 信号。

对于参数 length 任需要注意，参数 length 的值不能大于文件大小，即文件被映射的部分不能超出文件。

与映射区相关的两个信号

- **SIGSEGV：**如果映射区被 mmap()指定成了只读的，那么进程试图将数据写入到该映射区时，将会产生 SIGSEGV 信号，此信号由内核发送给进程。在第八章中给大家介绍过该信号，该信号的系统默认操作是终止进程、并生成核心可用于调试的核心转储文件。
- **SIGBUS：**如果映射区的某个部分在访问时已不存在，则会产生 SIGBUS 信号。例如，调用 mmap()进行映射时，将参数 length 设置为文件长度，但在访问映射区之前，另一个进程已将该文件截断（譬如调用 ftruncate()函数进行截断），此时如果进程试图访问对应于该文件已截去部分的映射区，进程将会受到内核发送过来的 SIGBUS 信号，同样，该信号的系统默认操作是终止进程、并生成核心可用于调试的核心转储文件。

munmap()解除映射

通过 open()打开文件，需要使用 close()将将其关闭；同理，通过 mmap()将文件映射到进程地址空间中的一块内存区域中，当不再需要时，必须解除映射，使用 munmap()解除映射关系，其函数原型如下所示：

```
#include <sys/mman.h>

int munmap(void *addr, size_t length);
```

同样，使用该函数需要包含头文件<sys/mman.h>。

munmap()系统调用解除指定地址范围内的映射，参数 addr 指定待解除映射地址范围的起始地址，它必须是系统页大小的整数倍；参数 length 是一个非负整数，指定了待解除映射区域的大小（字节数），被解除映射的区域对应的大小也必须是系统页大小的整数倍，即使参数 length 并不等于系统页大小的整数倍，与 mmap()函数相似。

需要注意的是，当进程终止时也会自动解除映射（如果程序中没有显式调用 munmap()），但调用 close()关闭文件时并不会解除映射。

通常将参数 addr 设置为 mmap()函数的返回值，将参数 length 设置为 mmap()函数的参数 length，表示解除整个由 mmap()函数所创建的映射。

使用示例

通过以上介绍，接下来我们编写一个简单地示例代码，使用存储映射 I/O 进行文件复制。

示例代码 13.5.1 演示了使用存储映射 I/O 实现文件复制操作，将源文件中的内容全部复制到另一个目标文件中，其效果类似于 cp 命令。

示例代码 13.5.1 使用存储映射 I/O 复制文件

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

```
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>

int main(int argc, char *argv[])
{
	int srcfd, dstfd;
	void *srcaddr;
	void *dstaddr;
	int ret;
	struct stat sbuf;

	if (3 != argc) {
		fprintf(stderr, "usage: %s <srcfile> <dstfile>\n", argv[0]);
		exit(-1);
	}

	/* 打开源文件 */
	srcfd = open(argv[1], O_RDONLY);
	if (-1 == srcfd) {
		perror("open error");
		exit(-1);
	}

	/* 打开目标文件 */
	dstfd = open(argv[2], O_RDWR |
			O_CREAT | O_TRUNC, 0664);
	if (-1 == dstfd) {
		perror("open error");
		ret = -1;
		goto out1;
	}

	/* 获取源文件的大小 */
	fstat(srcfd, &sbuf);

	/* 设置目标文件的大小 */
	ftruncate(dstfd, sbuf.st_size);

	/* 将源文件映射到内存区域中 */
	srcaddr = mmap(NULL, sbuf.st_size,
		PROT_READ, MAP_SHARED, srcfd, 0);
	if (MAP_FAILED == srcaddr) {
```

```
            perror("mmap error");
            ret = -1;
            goto out2;
        }

        /* 将目标文件映射到内存区域中 */
        dstaddr = mmap(NULL, sbuf.st_size,
            PROT_WRITE, MAP_SHARED, dstfd, 0);
        if (MAP_FAILED == dstaddr) {
            perror("mmap error");
            ret = -1;
            goto out3;
        }

        /* 将源文件中的内容复制到目标文件中 */
        memcpy(dstaddr, srcaddr, sbuf.st_size);

        /* 程序退出前清理工作 */
out4:
        /* 解除目标文件映射 */
        munmap(dstaddr, sbuf.st_size);
out3:
        /* 解除源文件映射 */
        munmap(srcaddr, sbuf.st_size);
out2:
        /* 关闭目标文件 */
        close(dstfd);
out1:
        /* 关闭源文件并退出 */
        close(srcfd);
        exit(ret);
}
```

当执行程序的时候，将源文件和目标文件传递给应用程序，该程序首先会将源文件和目标文件打开，源文件以只读方式打开，而目标文件以可读、可写方式打开，如果目标文件不存在则创建它，并且将文件的大小截断为 0。

然后使用 fstat()函数获取源文件的大小，接着调用 ftruncate()函数设置目标文件的大小与源文件大小保持一致。

然后对源文件和目标文件分别调用 mmap()，将文件映射到内存当中；对于源文件，调用 mmap()时将参数 prot 指定为 PROT_READ，表示对它的映射区会进行读取操作；对于目标文件，调用 mmap()时将参数 port 指定为 PROT_WRITE，表示对它的映射区会进行写入操作。最后调用 memcpy()将源文件映射区中的内容复制到目标文件映射区中，完成文件的复制操作。

接下来我们进行测试，笔者使用当前目录下的 srcfile 作为源文件，dstfile 作为目标文件，先看看源文件 srcfile 的内容，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
srcfile  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat srcfile
Linux高级I/O之存储映射I/O测试
使用存储映射I/O实现文件复制
将源文件中的内容复制到目标文件中
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.5.2 源文件中的内容

目标文件 dstfile 并不存在，我们需要在程序中进行创建，编译程序、运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
srcfile  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./srcfile ./dstfile
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
dstfile  srcfile  testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ cat dstfile
Linux高级I/O之存储映射I/O测试
使用存储映射I/O实现文件复制
将源文件中的内容复制到目标文件中
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.5.3 测试结果

由打印信息可知，程序运行完之后，生成了目标文件 dstfile，使用 cat 命令查看到其内容与源文件 srcfile 相同，本测试程序成功实现了文件复制功能！

13.5.2 mprotect()函数

使用系统调用 mprotect()可以更改一个现有映射区的保护要求，其函数原型如下所示：

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

使用该函数，同样需要包含头文件<sys/mman.h>。

参数 prot 的取值与 mmap()函数的 prot 参数的一样，mprotect()函数会将指定地址范围的保护要求更改为参数 prot 所指定的类型，参数 addr 指定该地址范围的起始地址，addr 的值必须是系统页大小的整数倍；参数 len 指定该地址范围的大小。

mprotect()函数调用成功返回 0；失败将返回-1，并且会设置 errno 来只是错误原因。

13.5.3 msync()函数

在第四章中提到过，read()和 write()系统调用在操作磁盘文件时不会直接发起磁盘访问（读写磁盘硬件），而是仅仅在用户空间缓冲区和内核缓冲区之间复制数据，在后续的某个时刻，内核会将其缓冲区中的数据写入（刷新至）磁盘中，所以由此可知，调用 write()写入到磁盘文件中的数据并不会立马写入磁盘，而是会先缓存在内核缓冲区中，所以就会出现 write()操作与磁盘操作并不同步，也就是数据不同步。

对于存储 I/O 来说亦是如此，写入到文件映射区中的数据也不会立马刷新至磁盘设备中，而是会在我们将数据写入到映射区之后的某个时刻将映射区中的数据写入磁盘中。所以会导致映射区中的内容与磁盘文件中的内容不同步。我们可以调用 msync()函数将映射区中的数据刷写、更新至磁盘文件中（同步操作），系统调用 msync()类似于 fsync()函数，不过 msync()作用于映射区。该函数原型如下所示：

```
#include <sys/mman.h>

int msync(void *addr, size_t length, int flags);
```

使用该函数，同样需要包含头文件<sys/mman.h>。

参数 addr 和 length 指定了需同步的内存区域的起始地址和大小。对于参数 addr 来说，同样也要求必须是系统页大小的整数倍，也就是与系统页大小对齐。譬如，调用 msync()时，将 addr 设置为 mmap()函数的返回值，将 length 设置为 mmap()函数的 length 参数，将对文件的整个映射区进行同步操作。

参数 flags 应指定为 MS_ASYNC 和 MS_SYNC 两个标志之一，除此之外，还可以根据需求选择是否指定 MS_INVALIDATE 标志，作为一个可选标志。

- **MS_ASYNC：**以异步方式进行同步操作。调用 msync()函数之后，并不会等待数据完全写入磁盘之后才返回。
- **MS_SYNC：**以同步方式进行同步操作。调用 msync()函数之后，需等待数据全部写入磁盘之后才返回。
- **MS_INVALIDATE：**是一个可选标志，请求使同一文件的其它映射无效（以便可以用刚写入的新值更新它们）。

msync()函数在调用成功情况下返回 0；失败将返回-1、并设置 errno。

munmap()函数并不影响被映射的文件，也就是说，当调用 munmap()解除映射时并不会将映射区中的内容写到磁盘文件中。如果 mmap()指定了 MAP_SHARED 标志，对于文件的更新，会在我们将数据写入到映射区之后的某个时刻将映射区中的数据更新到磁盘文件中，由内核根据虚拟存储算法自动进行。

如果 mmap()指定了 MAP_PRIVATE 标志，在解除映射之后，进程对映射区的修改将会丢弃！

13.5.4 普通 I/O 与存储映射 I/O 比较

通过前面的介绍，相信大家对存储映射 I/O 之间有了一个新的认识，本小节我们再来对普通 I/O 方式和存储映射 I/O 做一个简单的总结。

普通 I/O 方式的缺点

普通 I/O 方式一般是通过调用 read()和 write()函数来实现对文件的读写，使用 read()和 write()读写文件时，函数经过层层的调用后，才能够最终操作到文件，中间涉及到很多的函数调用过程，数据需要在不同的缓存间倒腾，效率会比较低。同样使用标准 I/O（库函数 fread()、fwrite()）也是如此，本身标准 I/O 就是对普通 I/O 的一种封装。

那既然效率较低，为啥还要使用这种方式呢？原因在于，只有当数据量比较大时，效率的影响才会比较明显，如果数据量比较小，影响并不大，使用普通的 I/O 方式还是非常方便的。

存储映射 I/O 的优点

存储映射 I/O 的实质其实是共享，与 IPC 之内存共享很相似。譬如执行一个文件复制操作来说，对于普通 I/O 方式，首先需要将源文件中的数据读取出来存放在一个应用层缓冲区中，接着再将缓冲区中的数据写入到目标文件中，如下所示：

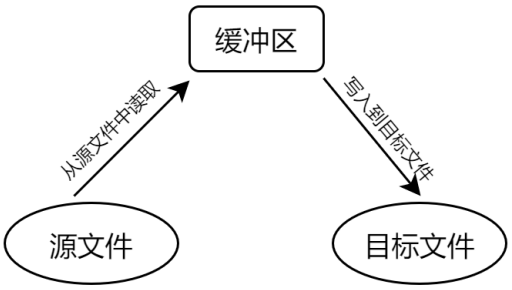


图 13.5.4 普通 I/O 实现文件复制示例图

而对于存储映射 I/O 来说，由于源文件和目标文件都已映射到了应用层的内存区域中，所以直接操作映射区来实现文件复制，如下所示：

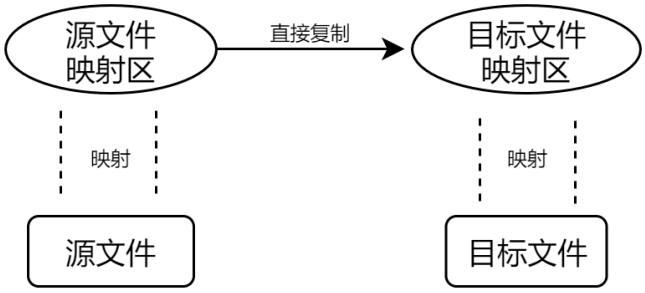


图 13.5.5 存储映射 I/O 实现文件复制

首先非常直观的一点就是，使用存储映射 I/O 减少了数据的复制操作，所以在效率上会比普通 I/O 要高，其次上面也讲了，普通 I/O 中间涉及到了很多的函数调用过程，这些都会导致普通 I/O 在效率上会比存储映射 I/O 要低。

前面提到存储映射 I/O 的实质其实是共享，如何理解共享呢？其实非常简单，我们知道，应用层与内核层是不能直接进行交互的，必须要通过操作系统提供的系统调用或库函数来与内核进行数据交互，包括操作硬件。通过存储映射 I/O 将文件直接映射到应用程序地址空间中的一块内存区域中，也就是映射区；直接将磁盘文件直接与映射区关联起来，不用调用 read()、write()系统调用，直接对映射区进行读写操作即可操作磁盘上的文件，而磁盘文件中的数据也可反应到映射区中，这就是一种共享，可以认为映射区就是应用层与内核层之间的共享内存。

存储映射 I/O 的不足

存储映射 I/O 方式并不是完美的，它所映射的文件只能是固定大小，因为文件所映射的区域已经在调用 mmap()函数时通过 length 参数指定了。另外，文件映射的内存区域的大小必须是系统页大小的整数倍，譬如映射文件的大小为 96 字节，假定系统页大小为 4096 字节，那么剩余的 4000 字节全部填充为 0，虽然可以通过映射地址访问剩余的这些字节数据，但不能在映射文件中反应出来，由此可知，使用存储映射 I/O 在进行大数据量操作时比较有效；对于少量数据，使用普通 I/O 方式更加方便！

存储映射 I/O 的应用场景

由上面介绍可知，存储映射 I/O 在处理大量数据时效率高，对于少量数据处理不是很划算，所以通常来说，存储映射 I/O 会在视频图像处理方面用的比较多，譬如 Framebuffer 编程，通俗点说就是 LCD 编程，就会使用到存储映射 I/O。

13.6 文件锁

现象一下，当两个人同时编辑磁盘中同一份文件时，其后果将会如何呢？在 Linux 系统中，该文件的最后状态通常取决于写该文件的最后一个进程。多个进程同时操作同一文件，很容易导致文件中的数据发生混乱，因为多个进程对文件进行 I/O 操作时，容易产生竞争状态、导致文件中的内容与预想的不一致！

对于有些应用程序，进程有时需要确保只有它自己能够对某一文件进行 I/O 操作，在这段时间内不允许其它进程对该文件进行 I/O 操作。为了向进程提供这种功能，Linux 系统提供了文件锁机制。

前面学习过互斥锁、自旋锁以及读写锁，文件锁与这些锁一样，都是内核提供的锁机制，锁机制实现用于对共享资源的访问进行保护；只不过互斥锁、自旋锁、读写锁与文件锁的应用场景不一样，互斥锁、自旋锁、读写锁主要用在多线程环境下，对共享资源的访问进行保护，做到线程同步。

而文件锁，顾名思义是一种应用于文件的锁机制，当多个进程同时操作同一文件时，我们怎么保证文件数据的正确性，linux 通常采用的方法是对文件上锁，来避免多个进程同时操作同一文件时产生竞争状态。譬如进程对文件进行 I/O 操作时，首先对文件进行上锁，将其锁住，然后再进行读写操作；只要进程没有对文件进行解锁，那么其它的进程将无法对其进行操作；这样就可以保证，文件被锁住期间，只有它（该进程）可以对其进行读写操作。

一个文件既然可以被多个进程同时操作，那说明文件必然是一种共享资源，所以由此可知，归根结底，文件锁也是一种用于对共享资源的访问进行保护的机制，通过对文件上锁，来避免访问共享资源产生竞争状态。

文件锁的分类

文件锁可以分为建议性锁和强制性锁两种：

- **建议性锁**

建议性锁本质上是一种协议，程序访问文件之前，先对文件上锁，上锁成功之后再访问文件，这是建议性锁的一种用法；但是如果你的程序不管三七二十一，在没有对文件上锁的情况下直接访问文件，也是可以访问的，并非无法访问文件；如果是这样，那么建议性锁就没有起到任何作用，如果要使得建议性锁起作用，那么大家就要遵守协议，访问文件之前先对文件上锁。这就好比交通信号灯，规定红灯不能通行，绿灯才可以通行，但如果你非要在红灯的时候通行，谁也拦不住你，那么后果将会导致发生交通事故；所以必须要大家共同遵守交通规则，交通信号灯才能起到作用。

- **强制性锁：**

强制性锁比较好理解，它是一种强制性的要求，如果进程对文件上了强制性锁，其它的进程在没有获取到文件锁的情况下是无法对文件进行访问的。其本质原因在于，强制性锁会让内核检查每一个 I/O 操作（譬如 read()、write()），验证调用进程是否是该文件锁的拥有者，如果不是将无法访问文件。当一个文件被上锁进行写入操作的时候，内核将阻止其它进程对其进行读写操作。采取强制性锁对性能的影响很大，每次进行读写操作都必须检查文件锁。

在 Linux 系统中，可以调用 flock()、fcntl()以及 lockf()这三个函数对文件上锁，接下来将向大家介绍每个函数的使用方法。

13.6.1 flock()函数加锁

先来学习系统调用 flock()，使用该函数可以对文件加锁或者解锁，但是 flock()函数只能产生建议性锁，其函数原型如下所示：

```
#include <sys/file.h>

int flock(int fd, int operation);
```

使用该函数需要包含头文件<sys/file.h>。

函数参数和返回值含义如下：

fd：参数 fd 为文件描述符，指定需要加锁的文件。

operation：参数 operation 指定了操作方式，可以设置为以下值的其中一个：

- **LOCK_SH：**在 fd 引用的文件上放置一把共享锁。所谓共享，指的便是多个进程可以拥有对同一个文件的共享锁，该共享锁可被多个进程同时拥有。
- **LOCK_EX：**在 fd 引用的文件上放置一把排它锁（或叫互斥锁）。所谓互斥，指的便是互斥锁只能同时被一个进程所拥有。
- **LOCK_UN：**解除文件锁定状态，解锁、释放锁。

除了以上三个标志外，还有一个标志：

- **LOCK_NB：**表示以非阻塞方式获取锁。默认情况下，调用 flock()无法获取到文件锁时会阻塞、直到其它进程释放锁为止，如果不想让程序被阻塞，可以指定 LOCK_NB 标志，如果无法获取到锁应立刻返回（错误返回，并将 errno 设置为 EWOULDBLOCK），通常与 LOCK_SH 或 LOCK_EX 一起使用，通过位或运算符组合在一起。

返回值：成功将返回 0；失败返回-1、并会设置 errno，

对于 flock()，需要注意的是，同一个文件不会同时具有共享锁和互斥锁。

使用示例

示例代码 13.6.1 演示了使用 flock()函数对一个文件加锁和解锁（建议性锁）。程序首先调用 open()函数将文件打开，文件路径通过传参的方式传递进来；文件打开成功之后，调用 flock()函数对文件加锁（非阻塞方式、排它锁），并打印出“文件加锁成功”信息，如果加锁失败便会打印出“文件加锁失败”信息。然后调用 signal 函数为 SIGINT 信号注册了一个信号处理函数，当进程接收到 SIGINT 信号后会执行 sigint_handler()函数，在信号处理函数中对文件进行解锁，然后终止进程。

示例代码 13.6.1 使用 flock()对文件加锁/解锁

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

static int fd = -1; //文件描述符

/* 信号处理函数 */
static void sigint_handler(int sig)
{
    if (SIGINT != sig)
        return;

    /* 解锁 */
    flock(fd, LOCK_UN);
    close(fd);
    printf("进程 1: 文件已解锁!\n");
```

```
}

int main(int argc, char *argv[])
{
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_WRONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 以非阻塞方式对文件加锁(排它锁) */
    if (-1 == flock(fd, LOCK_EX | LOCK_NB)) {
        perror("进程 1: 文件加锁失败");
        exit(-1);
    }

    printf("进程 1: 文件加锁成功!\n");

    /* 为 SIGINT 信号注册处理函数 */
    signal(SIGINT, sigint_handler);

    for ( ; ; )
        sleep(1);
}
```

加锁成功之后，程序进入了 for 死循环，一直持有锁；此时我们可以执行另一个程序，如示例代码 13.6.2 所示，该程序首先也会打开文件，文件路径通过传参的方式传递进来，同样在程序中也会调用 flock()函数对文件加锁（排它锁、非阻塞方式），不管加锁成功与否都会执行下面的 I/O 操作，将数据写入文件、在读取出来并打印。

示例代码 13.6.2 未获取锁情况下读写文件

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

```
int main(int argc, char *argv[])
{
    char buf[100] = "Hello World!";
    int fd;
    int len;

    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 以非阻塞方式对文件加锁(排它锁) */
    if (-1 == flock(fd, LOCK_EX | LOCK_NB))
        perror("进程 2: 文件加锁失败");
    else
        printf("进程 2: 文件加锁成功!\n");

    /* 写文件 */
    len = strlen(buf);
    if (0 > write(fd, buf, len)) {
        perror("write error");
        exit(-1);
    }
    printf("进程 2: 写入到文件的字符串<%s>\n", buf);

    /* 将文件读写位置移动到文件头 */
    if (0 > lseek(fd, 0x0, SEEK_SET)) {
        perror("lseek error");
        exit(-1);
    }

    /* 读文件 */
    memset(buf, 0x0, sizeof(buf));   //清理 buf
    if (0 > read(fd, buf, len)) {
        perror("read error");
```

```
        exit(-1);
    }
    printf("进程 2: 从文件读取的字符串<%s>\n", buf);

    /* 解锁、退出 */
    flock(fd, LOCK_UN);
    close(fd);
    exit(0);
}
```

把示例代码 13.6.1 作为应用程序 1，把示例代码 13.6.2 作为应用程序 2，将它们分别编译成不同的可执行文件 testApp1 和 testApp2，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp1  testApp2  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.1 两份可执行文件

在进行测试之前，创建一个测试用的文件 infile，直接使用 touch 命令创建即可，首先执行 testApp1 应用程序，将 infile 文件作为输入文件，并将其放置在后台运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
infile  testApp1  testApp2  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp1 ./infile &
[1] 20710
进程1: 文件加锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
    PID TTY          TIME CMD
   6535 pts/21   00:00:02 bash
  20710 pts/21   00:00:00 testApp1
  20713 pts/21   00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.2 执行 testApp1

testApp1 会在后台运行，由 ps 命令可查看到其 pid 为 20710。接着执行 testApp2 应用程序，传入相同的文件 infile，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp2 ./infile
进程2: 文件加锁失败: Resource temporarily unavailable
进程2: 写入到文件的字符串<Hello World!>
进程2: 从文件读取的字符串<Hello World!>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.3 执行 testApp2

从打印信息可知，testApp2 进程对 infile 文件加锁失败，原因在于锁已经被 testApp1 进程所持有，所以 testApp2 加锁自然会失败；但是可以发现虽然加锁失败，但是 testApp2 对文件的读写操作是没有问题的，是成功的，这就是建议性锁的特点；正确的使用方式是，在加锁失败之后不要再对文件进行 I/O 操作了，遵循这个协议。

接着我们向 testApp1 进程发送一个 SIGIO 信号，让其对文件 infile 解锁，接着再执行一次 testApp2，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ kill -2 20710
进程1: 文件已解锁!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp2 infile
进程2: 文件加锁成功!
进程2: 写入到文件的字符串<Hello World!>
进程2: 从文件读取的字符串<Hello World!>
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.4 测试结果

使用 kill 命令向 testApp1 进程发送编号为 2 的信号，也就是 SIGIO 信号，testApp1 接收到信号之后，对 infile 文件进行解锁、然后退出；接着再次执行 testApp2 程序，从打印信息可知，这次能够成功对 infile 文件加锁了，读写也是没有问题的。

关于 flock()的几条规则

- **同一进程对文件多次加锁不会导致死锁。**当进程调用 flock()对文件加锁成功，再次调用 flock()对文件（同一文件描述符）加锁，这样不会导致死锁，新加的锁会替换旧的锁。譬如调用 flock()对文件加共享锁，再次调用 flock()对文件加排它锁，最终文件锁会由共享锁替换为排它锁。
- **文件关闭的时候，会自动解锁。**进程调用 flock()对文件加锁，如果在未解锁之前将文件关闭，则会导致文件锁自动解锁，也就是说，文件锁会在相应的文件描述符被关闭之后自动释放。同理，当一个进程终止时，它所建立的锁将全部释放。
- **一个进程不可以对另一个进程持有的文件锁进行解锁。**
- **由 fork()创建的子进程不会继承父进程所创建的锁。**这意味着，若一个进程对文件加锁成功，然后该进程调用 fork()创建了子进程，那么对父进程创建的锁而言，子进程被视为另一个进程，虽然子进程从父进程继承了其文件描述符，但不能继承文件锁。这个约束是有道理的，因为锁的作用就是阻止多个进程同时写同一个文件，如果子进程通过 fork()继承了父进程的锁，则父进程和子进程就可以同时写同一个文件了。

除此之外，当一个文件描述符被复制时（譬如使用 dup()、dup2()或 fcntl()F_DUPFD 操作），这些通过复制得到的文件描述符和源文件描述符都会引用同一个文件锁，使用这些文件描述符中的任何一个进行解锁都可以，如下所示：

```
flock(fd, LOCK_EX);          //加锁
new_fd = dup(fd);
flock(new_fd, LOCK_UN);  //解锁
```

这段代码先在 fd 上设置一个排它锁，然后使用 dup()对 fd 进行复制得到新文件描述符 new_fd，最后通过 new_fd 来解锁，这样可以解锁成功。但是，如果不显示的调用一个解锁操作，只有当所有文件描述符都被关闭之后锁才会被释放。譬如上面的例子中，如果不调用 flock(new_fd, LOCK_UN)进行解锁，只有当 fd 和 new_fd 都被关闭之后锁才会自动释放。

关于本小节内容就暂时到这里为止！接下来我们将学习使用 fcntl()对文件上锁。

13.6.2 fcntl()函数加锁

fcntl()函数在前面章节内容中已经多次用到了，它是一个多功能文件描述符管理工具箱，通过配合不同的 cmd 操作命令来实现不同的功能。为了方便述说，这里再重申一次：

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

与锁相关的 cmd 为 F_SETLK、F_SETLKW、F_GETLK，第三个参数 flockptr 是一个 struct flock 结构体指针。使用 fcntl()实现文件锁功能与 flock()有两个比较大的区别：

- **flock()仅支持对整个文件进行加锁/解锁；而 fcntl()可以对文件的某个区域（某部分内容）进行加锁/解锁，可以精确到某一个字节数据。**
- **flock()仅支持建议性锁类型；而 fcntl()可支持建议性锁和强制性锁两种类型。**

我们先来看看 struct flock 结构体，如下所示：

示例代码 13.6.3 struct flock 结构体

```
struct flock {
    ...
    short l_type;       /* Type of lock: F_RDLCK,F_WRLCK, F_UNLCK */
    short l_whence;     /* How to interpret l_start: SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;      /* Starting offset for lock */
    off_t l_len;        /* Number of bytes to lock */
    pid_t l_pid;        /* PID of process blocking our lock(set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

对 struct flock 结构体说明如下：

- **l_type：**所希望的锁类型，可以设置为 F_RDLCK、F_WRLCK 和 F_UNLCK 三种类型之一，F_RDLCK 表示共享性质的读锁，F_WRLCK 表示独占性质的写锁，F_UNLCK 表示解锁一个区域。
- **l_whence 和 l_start：**这两个变量用于指定要加锁或解锁区域的起始字节偏移量，与 2.7 小节所学的 lseek()函数中的 offset 和 whence 参数相同，这里不再重述，如果忘记了，可以回到 2.7 小节再看看。
- **l_len：**需要加锁或解锁区域的字节长度。
- **l_pid：**一个 pid，指向一个进程，表示该进程持有的锁能阻塞当前进程，当 cmd=F_GETLK 时有效。

以上便是对 struct flock 结构体各成员变量的简单介绍，对于加锁和解锁区域的说明，还需要注意以下几项规则：

- 锁区域可以在当前文件末尾处开始或者越过末尾处开始，但是不能在文件起始位置之前开始。
- 若参数 l_len 设置为 0，表示将锁区域扩大到最大范围，也就是说从锁区域的起始位置开始，到文件的最大偏移量处（也就是文件末尾）都处于锁区域范围内。而且是动态的，这意味着不管向该文件追加写了多少数据，它们都处于锁区域范围，起始位置可以是文件的任意位置。
- 如果我们需要对整个文件加锁，可以将 l_whence 和 l_start 设置为指向文件的起始位置，并且指定参数 l_len 等于 0。

两种类型的锁：F_RDLCK 和 F_WRLCK

上面我们提到了两种类型的锁，分别为共享性读锁（F_RDLCK）和独占性写锁（F_WRLCK）。基本的规则与 12.5 小节所介绍的线程同步读写锁很相似，任意多个进程在一个给定的字节上可以有一把共享的读锁，但是在一个给定的字节上只能有一个进程有一把独占写锁，进一步而言，如果在一个给定的字节上已经有一把或多把读锁，则不能在该字节上加写锁；如果在一个字节上已经有一把独占性写锁，则不能再对它加任何锁（包括读锁和写锁），下图显示了这些兼容性规则：

		请求	
		读锁	写锁
当前区域	无锁	允许	允许
	有一把或多把读锁	允许	拒绝
	有一把写锁	拒绝	拒绝

图 13.6.5 不同类型锁彼此之间的兼容性

如果一个进程对文件的某个区域已经上了一把锁，后来该进程又试图在该区域再加一把锁，那么通常新加的锁将替换旧的锁。譬如，若某一进程在文件的 100~200 字节区间有一把写锁，然后又试图在 100~200 字节区间再加一把读锁，那么该请求将会成功执行，原来的写锁会替换为读锁。

还需要注意另外一个问题，当对文件的某一区域加读锁时，调用进程必须对该文件有读权限，譬如 open()时 flags 参数指定了 O_RDONLY 或 O_RDWR；当对文件的某一区域加写锁时，调用进程必须对该文件有写权限，譬如 open()时 flags 参数指定了 O_WRONLY 或 O_RDWR。

F_SETLK、F_SETLKW 和 F_GETLK

我们来看看与文件锁相关的三个 cmd 它们的作用：

- **F_GETLK：**这种用法一般用于测试，测试调用进程对文件加一把由参数 flockptr 指向的 struct flock 对象所描述的锁是否会加锁成功。如果加锁不成功，意味着该文件的这部分区域已经存在一把锁，并且由另一进程所持有，并且调用进程加的锁与现有锁之间存在排斥关系，现有锁会阻止调用进程想要加的锁，并且现有锁的信息将会重写参数 flockptr 指向的对象信息。如果不存在这种情况，也就是说 flockptr 指向的 struct flock 对象所描述的锁会加锁成功，则除了将 struct flock 对象的 l_type 修改为 F_UNLCK 之外，结构体中的其它信息保持不变。
- **F_SETLK：**对文件添加由 flockptr 指向的 struct flock 对象所描述的锁。譬如试图对文件的某一区域加读锁（l_type 等于 F_RDLCK）或写锁（l_type 等于 F_WRLCK），如果加锁失败，那么 fcntl()将立即出错返回，此时将 errno 设置为 EACCES 或 EAGAIN。也可用于清除由 flockptr 指向的 struct flock 对象所描述的锁（l_type 等于 F_UNLCK）。
- **F_SETLKW：**此命令是 F_SETLK 的阻塞版本（命令名中的 W 表示等待 wait），如果所请求的读锁或写锁因另一个进程当前已经对所请求区域的某部分进行了加锁，而导致请求失败，那么调用进程将会进入阻塞状态。只有当请求的锁可用时，进程才会被唤醒。

F_GETLK 命令一般很少用，事先用 F_GETLK 命令测试是否能够对文件加锁，然后再用 F_SETLK 或 F_SETLKW 命令对文件加锁，但这两者并不是原子操作，所以即使测试结果表明可以加锁成功，但是在使用 F_SETLK 或 F_SETLKW 命令对文件加锁之前也有可能被其它进程锁住。

使用示例与测试

示例代码 13.6.4 演示了使用 fcntl()对文件加锁和解锁的操作。需要加锁的文件通过外部传参传入，先调用 open()函数以只写方式打开文件；接着对 struct flock 类型对象 lock 进行填充，l_type 设置为 F_WRLCK 表示加一个写锁，通过 l_whence 和 l_start 两个变量将加锁区域的起始位置设置为文件头部，接着将 l_len 设置为 0 表示对整个文件加锁。

示例代码 13.6.4 使用 fcntl()对文件加锁/解锁使用示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

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

int main(int argc, char *argv[])
{
    struct flock lock = {0};
    int fd = -1;
    char buf[] = "Hello World!";

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_WRONLY);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 对文件加锁 */
    lock.l_type = F_WRLCK;   //独占性写锁
    lock.l_whence = SEEK_SET;    //文件头部
    lock.l_start = 0;                //偏移量为 0
    lock.l_len = 0;
    if (-1 == fcntl(fd, F_SETLK, &lock)) {
        perror("加锁失败");
        exit(-1);
    }

    printf("对文件加锁成功!\n");

    /* 对文件进行写操作 */
    if (0 > write(fd, buf, strlen(buf))) {
        perror("write error");
        exit(-1);
    }

    /* 解锁 */
```

```
    lock.l_type = F_UNLCK;  //解锁
    fcntl(fd, F_SETLK, &lock);

    /* 退出 */
    close(fd);
    exit(0);
}
```

整个代码很简单，比较容易理解，具体执行的结果就不再给大家演示了。

一个进程可以对同一个文件的不同区域进行加锁，当然这两个区域不能有重叠的情况。示例代码 13.6.5 演示了一个进程对同一文件的两个不同区域分别加读锁和写锁，对文件的 100~200 字节区间加了一个写锁，对文件的 400~500 字节区间加了一个读锁。

示例代码 13.6.5 对文件的不同区域进行加锁

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

int main(int argc, char *argv[])
{
    struct flock wr_lock = {0};
    struct flock rd_lock = {0};
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件大小截断为 1024 字节 */
    ftruncate(fd, 1024);

    /* 对 100~200 字节区间加写锁 */
    wr_lock.l_type = F_WRLCK;
```

```
    wr_lock.l_whence = SEEK_SET;
    wr_lock.l_start = 100;
    wr_lock.l_len = 100;
    if (-1 == fcntl(fd, F_SETLK, &wr_lock)) {
        perror("加写锁失败");
        exit(-1);
    }

    printf("加写锁成功!\n");

    /* 对 400~500 字节区间加读锁 */
    rd_lock.l_type = F_RDLCK;
    rd_lock.l_whence = SEEK_SET;
    rd_lock.l_start = 400;
    rd_lock.l_len = 100;
    if (-1 == fcntl(fd, F_SETLK, &rd_lock)) {
        perror("加读锁失败");
        exit(-1);
    }

    printf("加读锁成功!\n");

    /* 对文件进行 I/O 操作 */
    // ......
    // ......

    /* 解锁 */
    wr_lock.l_type = F_UNLCK;     //写锁解锁
    fcntl(fd, F_SETLK, &wr_lock);

    rd_lock.l_type = F_UNLCK;     //读锁解锁
    fcntl(fd, F_SETLK, &rd_lock);

    /* 退出 */
    close(fd);
    exit(0);
}
```

如果两个区域出现了重叠，譬如 100~200 字节区间和 150~250 字节区间，150~200 就是它们的重叠部分，一个进程对同一文件的相同区域不可能同时加两把锁，新加的锁会把旧的锁替换掉，譬如先对 100~200 字节区间加写锁、再对 150~250 字节区间加读锁，那么 150~200 字节区间最终是读锁控制的，关于这个问题，大家可以自己去验证、测试。

接下来对读锁和写锁彼此之间的兼容性进行测试，使用示例代码 13.6.6 测试读锁的共享性。

示例代码 13.6.6 读锁的共享性测试

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

int main(int argc, char *argv[])
{
    struct flock lock = {0};
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件大小截断为 1024 字节 */
    ftruncate(fd, 1024);

    /* 对 400~500 字节区间加读锁 */
    lock.l_type = F_RDLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 400;
    lock.l_len = 100;
    if (-1 == fcntl(fd, F_SETLK, &lock)) {
        perror("加读锁失败");
        exit(-1);
    }

    printf("加读锁成功!\n");
    for ( ; ; )
        sleep(1);
}
```

首先运行上述示例代码，程序加读锁之后会进入死循环，进程一直在运行着、持有读锁。接着多次运行上述示例代码，启动多个进程加读锁，测试结果如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
infile  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[1] 38277
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[2] 38278
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[3] 38279
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[4] 38280
加读锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ps
    PID TTY          TIME CMD
   6535 pts/21   00:00:05 bash
  38277 pts/21   00:00:00 testApp
  38278 pts/21   00:00:00 testApp
  38279 pts/21   00:00:00 testApp
  38280 pts/21   00:00:00 testApp
  38283 pts/21   00:00:00 ps
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.6 读锁共享性测试

从打印信息可以发现，多个进程对同一文件的相同区域都可以加读锁，说明读锁是共享性的。由于程序是放置在后台运行的，测试完毕之后，可以使用 kill 命令将这些进程杀死，或者直接关闭当前终端，重新启动新的终端。

使用示例代码 13.6.7 测试写锁的独占性。

示例代码 13.6.7 写锁的独占性测试

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

int main(int argc, char *argv[])
{
    struct flock lock = {0};
    int fd = -1;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR);
    if (-1 == fd) {
```

```
        perror("open error");
        exit(-1);
    }

    /* 将文件大小截断为 1024 字节 */
    ftruncate(fd, 1024);

    /* 对 400~500 字节区间加写锁 */
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 400;
    lock.l_len = 100;
    if (-1 == fcntl(fd, F_SETLK, &lock)) {
        perror("加写锁失败");
        exit(-1);
    }

    printf("加写锁成功!\n");
    for ( ; ; )
        sleep(1);
}
```

测试方法与读锁测试方法一样，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
infile  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile &
[1] 38531
加写锁成功!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile
加写锁失败: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile
加写锁失败: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp ./infile
加写锁失败: Resource temporarily unavailable
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.7 写锁的独占性测试

由打印信息可知，但第一次启动的进程对文件加写锁之后，后面再启动进程对同一文件的相同区域加写锁发现都会失败，所以由此可知，写锁是独占性的。

几条规则

关于使用 fcntl()创建锁的几条规则与 flock()相似，如下所示：

- **文件关闭的时候，会自动解锁。**
- **一个进程不可以对另一个进程持有的文件锁进行解锁。**
- **由 fork()创建的子进程不会继承父进程所创建的锁。**

除此之外，当一个文件描述符被复制时（譬如使用 dup()、dup2()或 fcntl()F_DUPFD 操作），这些通过复制得到的文件描述符和源文件描述符都会引用同一个文件锁，使用这些文件描述符中的任何一个进行解锁都可以，这点与 flock()是一样的，如下所示：

```
lock.l_type = F_RDLCK;
fcntl(fd, F_SETLK, &lock);//加锁

new_fd = dup(fd);

lock.l_type = F_UNLCK;
fcntl(new_fd, F_SETLK, &lock);//解锁
```

这段代码先在 fd 上设置一个读锁，然后使用 dup()对 fd 进行复制得到新文件描述符 new_fd，最后通过 new_fd 来解锁，这样可以解锁成功。如果不显示的调用一个解锁操作，任何一个文件描述符被关闭之后锁都会自动释放，那么这点与 flock()是不同的。譬如上面的例子中，如果不调用 flock(new_fd, LOCK_UN)进行解锁，当 fd 或 new_fd 两个文件描述符中的任何一个被关闭之后锁都会自动释放。

建议性锁和强制性锁

前面我们提到了 fcntl()支持强制性锁和建议性锁，但是一般不建议使用强制性锁，所以大部分情况下使用的都是建议性锁，那如何使能强制性锁呢？

对于一个特定的文件，开启它的强制性锁机制其实非常简单，主要跟文件的权限位有关系，在 5.5 小节对文件的权限进行了比较详细的介绍，这里不再重述！如果要开启强制性锁机制，需要设置文件的 Set-Group-ID（S_ISGID）位为 1，并且禁止文件的组用户执行权限（S_IXGRP），也就是将其设置为 0。

但是，有些 Linux/Unix 发行版系统并不支持强制性锁机制，可以通过示例代码 13.6.8 进行测试。

示例代码 13.6.8 测试系统是否支持强制性锁机制

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

int main(int argc, char *argv[])
{
    struct stat sbuf = {0};
    int fd = -1;
    pid_t pid;

    /* 校验传参 */
    if (2 != argc) {
        fprintf(stderr, "usage: %s <file>\n", argv[0]);
        exit(-1);
    }

    /* 打开文件 */
    fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, 0664);
    if (-1 == fd) {
        perror("open error");
```

```
        exit(-1);
    }

    /* 写入一行字符串 */
    if (12 != write(fd, "Hello World!", 12)) {
        perror("write error");
        exit(-1);
    }

    /* 开启强制性锁机制 */
    if (0 > fstat(fd, &sbuf)) {//获取文件属性
        perror("fstat error");
        exit(-1);
    }
    if (0 > fchmod(fd, (sbuf.st_mode & ~S_IXGRP)
            | S_ISGID)) {
        perror("fchmod error");
        exit(-1);
    }

    /* fork 创建子进程 */
    if (0 > (pid = fork()))        //出错
        perror("fork error");
    else if (0 < pid) { //父进程
        struct flock lock = {0};

        /* 对整个文件加写锁 */
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 0;
        lock.l_len = 0;
        if (0 > fcntl(fd, F_SETLK, &lock))
            perror("父进程: 加写锁失败");
        else
            printf("父进程: 加写锁成功!\n");

        printf("~~~~~~~~~~~~~~~~~~~~\n");
        if (0 > wait(NULL))
            perror("wait error");
    }
    else {    //子进程
        struct flock lock = {0};
        int flag;
```

```
        char buf[20] = {0};

        sleep(1);    //休眠 1 秒钟，让父进程先运行

        /* 设置为非阻塞方式 */
        flag = fcntl(fd, F_GETFL);
        flag |= O_NONBLOCK;
        fcntl(fd, F_SETFL, flag);

        /* 对整个文件加读锁 */
        lock.l_type = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 0;
        lock.l_len = 0;
        if (-1 == fcntl(fd, F_SETLK, &lock))
            perror("子进程: 加读锁失败");
        else
            printf("子进程: 加读锁成功!\n");

        /* 读文件 */
        if (0 > lseek(fd, 0, SEEK_SET))
            perror("lseek error");
        if (0 > read(fd, buf, 12))
            perror("子进程: read error");
        else
            printf("子进程: read OK, buf = %s\n", buf);
    }

    exit(0);
}
```

此程序首先创建了一个文件，文件路径通过传参的方式传递给应用程序，如果不存在该文件则创建它。接着向文件中写入数据，开启文件的强制性锁机制。接下来程序调用 fork()创建了一个子进程，在父进程分支中，对文件的所有区域加了一把独占性质的写锁，接着调用 wait()等到回收子进程；在子进程分支中先是休眠了一秒钟以保证父进程先执行，子进程将文件设置为非阻塞方式，这里大家可能会有疑问？普通文件不都是非阻塞的吗？这里为什么要设置非阻塞呢？并不是多此一举，原因在于这里涉及到了强制性锁的问题，在强制性锁机制下，如果文件被进程添加了强制性写锁，其它进程读或写该文件将会被阻塞，所以我们需要显式设置为非阻塞方式。

设置为非阻塞之后，子进程试图对文件设置一把读锁，接着子进程将文件读、写位置移动到文件头，并试图 read 读该文件。

由于父进程已经对文件设置了写锁，子进程试图对文件设置读锁时，将会失败；子进程在没有获取到读锁的情况下，调用 read()读取文件将会出现两种情况：如果系统支持强制性锁机制，那么 read()将会失败；如果系统不支持强制性锁机制，read()将会成功！

接下来我们进行测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp infile
父进程：加写锁成功！
~~~~~~~~~~~~~~~~~~~~
子进程：加读锁失败：Resource temporarily unavailable
子进程：read OK, buf = Hello World!
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 13.6.8 Ubuntu 系统下测试结果

从打印信息可以发现，父进程设置了写锁的情况下，子进程再次对其设置读锁是不成功的，也就是子进程没有获取到读锁，但是读文件却是成功的，由此可知，我们测试所使用的 Ubuntu 系统不支持强制性锁机制。

13.6.3 lockf()函数加锁

lockf()函数是一个库函数，其内部是基于 fcntl()来实现的，所以 lockf()是对 fcntl 锁的一种封装，具体的使用方法这里便不再介绍。

13.7 小结

本章向大家介绍了几种高级 I/O 功能，非阻塞 I/O、I/O 多路复用、异步 I/O、存储映射 I/O、以及文件锁：

- 非阻塞 I/O：进程向文件发起 I/O 操作，使其不会被阻塞。
- I/O 多路复用：select()和 poll()函数。
- 异步 I/O：当文件描述符上可以执行 I/O 操作时，内核会向进程发送信号通知它。
- 存储映射 I/O：mmap()函数。
- 文件锁：flock()、fcntl()以及 lockf()函数。