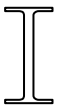

Linux 系统中的高级 UNIX 编程

第一章：起步

本章将向你展示如何在 Linux 系统中完成一个 C/C++ 程序的基本步骤。具体来说，本章讲解了如何在 Linux 系统中编辑 C 和 C++ 源码，编译并调试得到的程序。如果你已经对 Linux 环境下的程序编写相当熟悉，则完全可以跳过本章内容，直接开始阅读第二章，“编写优质的 GNU/Linux 软件”。第二章中 2.3 节“编写及使用程序库”中包含了对静态和动态库的比较，这也许是你还不知道的内容，值得关注。

我们在编写本书的时候，假定你已经对 C 或 C++ 程序设计语言以及标准 C 库的函数相当熟悉。除了为展示有关 C++ 独有的特性的情况时，书中的示例代码均用 C 语言写就。同时，我们还假定你知道如何在 Linux shell 中执行一些基本操作，例如创建文件夹和复制文件等。因为许多 Linux 程序员都是在 Windows 环境下开始的编程，我们会在一些时候特别指出两个平台上的不同点。

1.1 用 Emacs 进行编辑

*编辑器（editor）*是用于编辑代码的工具程序。Linux 平台上有各种不同的编辑器，但是最流行的、提供了最丰富特色的，当属 GNU Emacs 了。

> **关于 Emacs**
>
> Emacs 决不仅仅是一个编辑器。它是一个出奇强大的程序。在 CodeSourcery，它被亲切地称为“the One True Program”（译者注：记得 Matrix 里的 The One 吧^_^）或者直接简称 OTP。在 Emacs 中你可以查阅、发送电子邮件，你可以将 Emacs 进行任意的定制与扩充；可能性太多以至于不适合在这里进行讨论了。你甚至可以在 Emacs 中浏览网页！

如果你熟悉其它的编辑器，你当然可以选择使用它们。本书中的任何内容都不会依赖 Emacs 的特性。不过，如果你仍然没有一个习惯使用的 Linux 下的编辑器，那么你应该跟随这篇不长的教程，尝试学习一下 Emacs 的使用。

如果你喜欢 Emacs 并希望对它的高级特性了解得更多，你或许应该考虑阅读其它一些关于 Emacs 的书籍。有一篇非常不错的教程，*《学习 Emacs》（Learning GNU Emacs）*，作者是 Debra Cameron、Bill Rosenblatt 和 Eric S. Raymond（O’Reilly 公司于 1996 年出版。该书已由机械工业出版社翻译并出版，书名《学习 GNU Emacs（第二版）》）。

1.1.1 打开 C/C++代码文件

要运行Emacs，你只需在终端窗口中输入**emacs**并回车。当Emacs开始运行之后，你可以利用窗口顶部的菜单创建一个新的文件。点击“文件File”菜单，选择“打开文件Open Files”，然后在窗口底部的“minibuffer”中输入你希望打开的文件的名字。[1]如果你要创建的是一篇 C 代码，则后缀名应该选择 **.c** 或 **.h**。如果创建的是C++ 代码，后缀名应在 **.cpp**、**.hpp**、**.cxx**、**.hxx**、**.C**或者 **.H**中选择。当文件被打开之后，你可以像是使用其它任何字处理程序一样进行输入。保存文件只需要从文件菜单中选择“保存缓冲区Save Buffer”即可。当你准备退出Emacs的时候，只需从文件菜单选择“退出Emacs Exit Emacs”就可以。

[1] 如果你不是在X窗口系统中使用Emacs，你需要通过F10 键来访问菜单。

如果你不喜欢用鼠标指点江山，你可以选择使用键盘快捷键完成这些操作。输入 **C-x C-f** 可以打开文件（**C-x** 的意思是按下 Ctrl 键的同时按 x 键）。**C-x C-s** 是保存文件，而 **C-x C-c** 则是退出 Emacs。想要进一步熟悉 Emacs，可以从帮助菜单中选择 Emacs 指南（Emacs Tutorial）。这份文档中提供了无数帮助你更快捷有效地使用 Emacs 的技巧。

1.1.2 自动化排版

如果你已经习惯了在*集成开发环境（Integrated Development Environment, IDE）*中编写程序，你一定乐意由编辑器自动帮助你对代码进行排版。Emacs 同样提供了这种功能。当你打开一个 C/C++ 代码的时候，Emacs 自动识别出这是一篇代码而不仅是普通文本文件。当你在一个空行中点下 Tab 键的时候，Emacs 会将光标移动到合适的缩进位置。如果你在一个已经包含了内容的行中点击 Tab 键，Emacs 会将该行文字缩进到合适的地方。假设你输入了下面几行文字：

```
int main ()
{
printf (“Hello, world\n”);
}
```

当你在调用 **printf** 的一行点下 Tab 键的时候，Emacs 会将代码重新排版成这个样式：

```
int main ()
{
  printf (“Hello, world\n”);
}
```

注意中间一行被添加了合适的缩进。

当你更多地使用Emacs之后，你会发现它会帮你解决各种复杂的排版问题。如果你有兴趣，你甚至可以对Emacs进行程序控制，让它完成任何你可以想象得到的自动排版工作。人们利用Emacs的这个能力，为几乎任何种类的文档实现了Emacs编辑模式，甚至实现了游戏[2]和数据库前端。

1.1.3 语法高亮

除了对代码进行排版，Emacs 可以通过对 C 或 C++ 程序的不同元素加以染色以方便阅读。例如，Emacs 可以将关键字转为一种颜色，**int** 等内置类型使用第二种颜色，而对注释使用第三种颜色等。通过染色，你可以很轻松地发现一些简单的语法错误。

最简单的打开语法染色功能的途径是在 ~/.emacs 文件中插入下面一行文字：

```
(global-font-lock-mode t)
```

将这个文件保存，然后退出并重新启动 Emacs，再打开那些 C/C++ 代码，开始享受吧！

你可能注意到，刚才插入 .emacs 文件的文字看起来像是 LISP 程序语言的代码。这是因为，那*根本就是* LISP 代码！Emacs 的很大部分都是用 LISP 实现的。你可以通过编写 LISP 代码为 Emacs 加入更多的功能。

[2]如果你对那些老式的文本模式冒险游戏有兴趣的话，试着运行**M-x dunnet**命令。

1.2 用 GCC 编译

*编译器*可以将人类可读的程序代码转化为机器可以解析执行的对象代码。Linux系统中提供的编译器全部来自GNU编译器集合（GNU Compiler Collection），通常被称为GCC。[3] GCC中包含了C、C++、Java、Objective-C、Fortran和Chill语言的编译器。本书中我们主要关注的是C和C++ 语言的程序设计。

假设你有一个项目，其中包含一个如列表 1.2 中所示的C++ 程序（**reciprocal.cpp**）和一个如列表 1.1 所示的C程序（**main.c**）。这两个文件需要被编译并链接成为一个单独的程序**reciprocal**。[4] 这个程序可以计算一个整数的倒数。

代码列表 1.1 （main.c）C 源码——main.c

```
#include <stdlib.h>
#include <stdio.h>
#include “reciprocal.hpp”

int main (int argc, char **argv)
{
  int i;

  i = atoi (argv[1]);
  printf (“The reciprocal of %d is %g\n”, i, reciprocal (i));
  return 0;
}
```

代码列表 1.2 （reciprocal.cpp）C++源码——reciprocal.cpp

```
#include <cassert>
#include “reciprocal.hpp”

double reciprocal (int i) {
  // i 不能为 0
  assert (i != 0);
  return 1.0/i;
}
```

还有一个包含文件 **reciprocal.hpp**（列表 1.3 中）。

代码列表 1.3 （reciprocal.hpp）包含文件——reciprocal.hpp

[3] 请访问http://gcc.gnu.org 获取更多GCC相关的信息。

[4] 在Windows系统中，可执行程序的名称通常以 **.exe**结尾，而在Linux中通常没有后缀名。因此在Windows中，这个程序可能被称为**reciprocal.exe**而Linux版本则是简单的**reciprocal**。

```
#ifdef __cplusplus
extern "C" {
#endif

extern  double reciprocal (int i);

#ifdef __cplusplus
}
#endif
```

我们要做的第一步，就是将代码文件转化为对象文件。

1.2.1 编译单个代码文件

C 程序编译器是 **gcc**。可以通过指定 **–c** 选项编译 C 源码文件。因此，输入下面这一条命令可以将 **main.c** 文件编译成名为 **main.o** 的对象文件：

```
% gcc –c main.c
```

C++编译器是 **g++**。它的操作方式与 gcc 非常相似。下面一行命令可以完成对 **reciprocal.cpp** 的编译：

```
% g++ -c reciprocal.cpp
```

在这里，选项 **–c** 通知编译器只产生对象文件；否则编译器会尝试链接程序并产生最终的可执行文件。在执行完第二个命令之后你应该得到的是一个名为 **reciprocal.o** 的对象文件。

要构建一个大型的程序，你可能还需要熟悉其它一些选项。**-I** 选项会告诉编译器去哪里寻找包含文件。默认情况下，GCC 会在当前目录及标准库的包含文件所在的路径搜索程序所需的包含文件。如果你需要从其它的路径中搜索包含文件，你就需要通过 **–I** 选项指定这个路径。假设你的项目中包含一个用于保存源码文件的 **src** 目录，以及一个用于存放包含文件的 **include** 目录。你必须这样编译 **reciprocal.cpp**，以使 **g++**能够从**../include** 文件夹中搜索 **reciprocal.hpp** 文件：

```
% g++ -c –I ../include reciprocal.cpp
```

有时你会希望从编译命令中定义一些宏。例如，在发布版的程序中，你不希望在 **reciprocal.cpp** 中出现多余的断言检查——断言只有在程序的调试阶段才能起到相应的作用。**NDEBUG** 宏可以用于关闭断言检查。你可以在 **reciprocal.cpp** 中添加 **#define** 语句，但是这要求对源码的修改。更简单的方法是像这样直接通过命令行定义 **NDEBUG** 宏：

```
% g++ -c –D NDEBUG reciprocal.cpp
```

如果你希望将 **NDEBUG** 宏定义为某个特定的值，下面这个命令行可以做到：

```
% g++ -c –D NDEBUG=3 reciprocal.cpp
```

如果你现在编译的正是准备发布的版本，你或许希望 GCC 能将得到的代码尽量优化以提高运行速度。你可以通过指明 **-O2** 选项要求 GCC 进行代码优化。（GCC 有许多不同等级的代码优化，不过 2 级对多数情况都是适当的。）下面的命令可以打开优化并编译 **reciprocal.cpp**：

```
% g++ -c –O2 reciprocal.cpp
```

需要注意的是，优化选项会导致你的程序难以被调试程序（参考 1.4 节，“用 GDB 进行调试”）调试。此外，在某些特例中，启用了优化的编译可能会将一些之前没有被发现的 bug

显现出来。

Gcc 和 **g++** 还支持其它许多选项。获取完整列表的最佳方式是阅读相应的在线文档。可以在命令行下输入以下命令以获取文档：

```
% info gcc
```

1.3 用 GNU Make 自动完成编译过程

如果你习惯于 Windows 环境下的程序设计，你一定非常熟悉各种继承开发环境（IDE）的使用。你只需将代码文件加入工程，IDE 会自动帮你完成编译构建的过程。尽管在 Linux 平台上也有一些 IDE 实现，本书中我们不会进行讨论。相反的，我们要介绍的是如何利用 GNU Make 程序自动编译你的代码——这才是每个 Linux 程序员的工作方式。

Make 程序背后隐藏的理念是非常简单的。你告诉 **make** 程序你需要完成什么*目标（target）*，以及达成这些目标的*规则（rules）*。你还可以通过指定*依赖关系（dependency）*指明需要重新构建某些目标的条件。

在我们的示例项目 **reciprocal** 中，有三个目标是非常明显的：**reciprocal.o**、**main.o** 和 **reciprocal** 程序自己。在之前手工编译的过程中，你已经理清了构建这些目标的规则。依赖性则需要进一步的理解。很清楚的，**reciprocal** 依赖于 **main.o** 和 **reciprocal.o**，因为没有这两个对象文件的话，你无法进行链接。而每当源码文件被修改之后，对象文件都应被重新编译。还有一个需要注意的地方，就是如果 **reciprocal.hpp** 文件被修改，则两个对象文件均应被重新编译。这是因为两个对象对应的源码文件都包含了这个文件。

除了这些显而易见的目标，你始终应该指定一个名为 **clean** 的目标。这个目标对应的规则是删除所有编译生成的对象文件和程序文件，因此下次编译将是从头开始。这个目标的规则通常使用 **rm** 命令进行删除操作。

你可以通过一个 **Makefile** 将这些信息告诉 **make** 程序。这个 **Makefile** 可以写成这样：

```
reciprocal: main.o reciprocal.o
	g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp
	gcc $(CFLAGS) -c main.c

reciprocal.o: reciprocal.cpp reciprocal.hpp
	g++ $(CFLAGS) -c reciprocal.cpp

clean:
	rm -f *.o reciprocal
```

可以看出，目标写在最左侧一列，之后跟随一个冒号，然后是所有依赖项。用于构建目标的对应规则写在紧接着的下一行。（暂时忽略 **$(CFLAGS)** 这些东西，我们稍候会讲解。）包含规则的行必须以一个 Tab 字符（退格键）起头，否则 **make** 程序将无法识别。如果你用 Emacs 编辑 **Makefile**，Emacs 会帮助你打理排版方面的细节。

如果你已经删除了编译得到的对象文件，你只需要在命令行中输入

```
% make
```

然后你就会看见这样的输出：

```
% make
```

```
gcc  –c main.c
g++  -c reciprocal.cpp
g++  -o reciprocal main.o reciprocal.o
```

可以看出，**make** 自动建立了这些对象文件并完成了链接的步骤。如果你现在对 **main.c** 稍加修改，然后重新输入 **make** 命令，你将会看见下面的输出：

```
% make
gcc  -c main.c
g++  -o reciprocal main.o reciprocal.o
```

可以看出，**make** 正确地选择了重新编译生成 **main.o** 并重新链接整个程序，而没有去碰 **reciprocal.cpp**，因为 **reciprocal.o** 的任何一个依赖项都没有发生改变。

前面看到的 **$(CFLAGS)** 是一个 **make** 变量。你可以在 **Makefile** 中定义这个变量，也可以在命令行中定义。GNU **make** 会在执行这个规则的时候将变量的值代入。因此，假如要让编译器打开优化并重新编译程序，你可以这样操作：

```
% make clean
rm –f *.o reciprocal
% make CFLAGS=-O2
gcc -O2 –c main.c
g++ -O2 –c reciprocal.o
g++ -O2 –o reciprocal main.o reciprocal.o
```

应该注意到，**-O2** 被替换到了先前每个 **$(CFLAGS)** 出现的地方。

在这一小节中，我们只向你展示了 **make** 最基本的用途。你可以通过下面这个命令获取更多的信息：

```
% info make
```

在这份手册中，你会看到如何简化对 **Makefile** 的维护，如何降低必须写的规则数量，以及如何自动计算依赖性关系等。你还可以从《GNU, Autoconf, Automake, and Libtool》一书中找到更多的相关信息（作者：Gary V.Vaughan, Ben Elliston, Tom Tromey 以及 Ian Lance Taylor，2000 年由 New Riders 出版社出版）。

1.4 用 GDB 进行调试

*调试器*是一个工具，可以用来帮助你检查为什么程序行为与预期不同。你将经常进行这样的检查工作[5]。GNU调试器（The GNU Debugger, GDB）是一个被多数Linux程序员使用的调试器程序。利用GDB，你可以单步跟踪你的程序，设置断点以及检查局部变量的值。

1.4.1 在编译时加入调试信息

要使用 GDB，你必须在编译时为对象文件加入调试信息。只需在编译器选项中加入 **–g** 就可以做到这点。如果你使用前面介绍的 **Makefile**，你可以在运行 **make** 的时候将 **CFLAGS**

[5] ……除非你的程序每次都在第一遍就工作正常。

设置为 **–g**，如下所示：

```
% make CFLAGS=-g
gcc –g –c main.c
g++ -g –c reciprocal.cpp
g++ -g –o reciprocal main.o reciprocal.o
```

当你指定 **-g** 选项编译的时候，编译器会在生成的对象文件和执行文件中加入额外的信息。调试器则可以通过这些信息获取地址与源码位置的对应关系、局部变量的输出方法等信息。

1.4.2 运行 GDB

你可以通过输入下面的命令启动 **gdb**：

```
% gdb reciprocal
```

当 **gdb** 启动之后，你应该会看见它的提示符：

```
(gdb)
```

第一步要做的就是在调试器中运行你的程序：输入命令 **run**，之后跟着运行程序需要的所有参数。试着这样不提供参数运行这个程序：

```
(gdb) run
Starting program: reciprocal

Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287     strtol.c: No such file or directory.
(gdb)
```

出现这个问题的原因在于 **main** 函数中没有用于检查错误情况的代码。程序希望得到一个参数，而在这次调用过程中它没有得到需要的参数。那条 **SIGSEGV** 的消息标志着程序的崩溃。GDB 知道程序实际是在 **__strtol_internal** 这个函数中崩溃的。这个函数属于标准哭的一部分，而标准库的源码并没有安装在系统中，所以会出现“No such file or directory”（找不到文件或目录）的提示信息。你可以通过 **where** 命令查看调用堆栈：

```
(gdb) where
#0  _strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
   at strtol.c:287
#1  0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2  0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

可以看出，**main** 函数以一个 NULL 指针为参数调用了 **atoi**，而正是它导致了问题的出现。

你可以通过 **up** 命令沿着调用堆栈向上回溯两层，回到 **main** 函数中：

```
(gdb) up 2
#2  0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1]);
```

可以发现，**gdb** 找到了 **main.c**，并且现实了出错的函数调用所在处的代码。你可以用 **print** 命令查看变量的值：

```
(gdb) print argv[1]
$2 = 0x0
```

这再次证实了出错的原因是传递给 **atoi** 的 NULL 指针在作怪。

你可以利用 **break** 命令为程序设置断点：

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

这个命令在**main**函数的第一行设置了断点。[6]现在试着运行这个程序。这次我们传递一个参数：

```
(gdb) run 7
Starting program: reciprocal 7

Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8         i = aatoi (argv[1]);
```

你会看见，调试器在断点处停止了程序的运行。

利用 **next** 命令你可以单步追踪程序直到调用 **atoi** 的位置：

```
(gdb) next
9         printf (“The reciprocal of %d is %g\n”, i, reciprocal (i));
```

如果你希望看到 **reciprocal** 函数里面做了什么，则应改用 **step** 命令：

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6         assert (i != 0);
```

你现在所处的就是 **reciprocal** 函数的内部。

你也许会发现，从 Emacs 中运行 **gdb** 比在命令行中直接运行 **gdb** 更方便。可以通过命令 **M-x gdb** 在一个 Emacs 窗口中启动 **gdb** 调试器。当你在一个断点处停止的时候，Emacs 会自动现实对应的源码文件。通常来说，相比只能看见一行程序的情况，能够通读整个文件更容易让你找到头绪。

1.5 获取更多信息

几乎所有 Linux 发行版都会提供大量的有用的文档。你可以通过阅读你的发行版提供的文档以理解本书中我们提到的大多数问题（尽管这样可能会耗去你更多的时间）。这些文档可能并未被很好地组织，所以难处就在于，如何找到你想要的信息。有时候，文档可能会过期，所以在阅读的时候要抱着怀疑论的观点。假如你发现系统的工作方式与 *man page*(manual pages，手册页)所说的不同，很可能这些手册页就是过期的。

[6] 有人说**break main**（打破main）这个说法有些可笑，因为通常来说只有当**main**函数已经坏掉（broken）的时候你才需要做这件事。

为帮助你更快地上手，这里我们列出了一些有关 Linux 程序设计的有用信息：

1.5.1 手册页

Linux 发行版包含了有关常用命令、系统调用和库函数的手册页。手册页被分成不同的章节并分别标以序号；对于程序员而言，最重要的是这些：

（1）用户命令
（2）系统调用
（3）标准库函数
（8）系统/管理员命令

这些数字就是手册页所在的章节。Linux 的手册页已经被安装在系统中；你可以通过 **man** 命令查看它们。要查看一个手册页，只需要执行 **man *name***，这里 ***name*** 是一个命令或函数的名字。在某些情况下，不同章节中可能包含具有相同名字的手册页；你可以通过在 ***name*** 之前插入指定的章节号。例如，当运行下面的命令的时候，你会得到 **sleep** 命令的手册页（在 Linux 手册第一节中）：

```
% man sleep
```

如果要查看 **sleep** 库函数的手册页，则需要使用下面的命令：

```
% man 3 sleep
```

每个手册页都包含了一行对命令或函数的介绍。运行 **whatis *name*** 会显示系统中所有名称匹配的、位于任意章节中的所有手册页的介绍。如果你不清楚你要找的命令或函数的名字，你可以通过 **man –k keyword** 命令进行查找。

手册页包含了大量非常有用的信息，因此应该成为你遇见任何问题的时候第一个寻求解决方案的地方。命令相关的手册页介绍了命令行选项、输入输出、错误代码、配置和其它各种信息。系统调用和库函数的手册页介绍了参数和返回值、可能出现的错误代码和副作用，以及当你调用这个函数时需要包含的文件。

1.5.2 Info

Info 文档系统提供了更加详细的文档，范围涵盖了 GNU/Linux 系统的许多核心部件以及其它一些程序。Info 页面是一种与 HTML 页面类似的超文本文档。只需要在一个终端窗口输入 **info** 就可以启动文本界面的 Info 浏览器。首先你将看到的是在你的系统中已安装的所有 Info 页面的列表。（按下 Ctrl+H 键会显示用于浏览 Info 文档的键盘配置。）

其中最重要的一些文档包括了：

- **gcc**——GCC 编译器
- **libc**——GNU C 函数库，包含许多系统调用
- **gdb**——GNU 调试器
- **emacs**——Emacs 文本编辑器
- **info**——Info 系统自己的相关信息

几乎所有的标准 Linux 编程工具（包括链接器 **ld**、汇编程序 **as**、性能分析程序 **gprof**）都提供了详尽的 Info 页面。你可以通过在命令行中指点名字，直接跳到有关的 Info 页：

```
% info libc
```

如果你在 Emacs 中完成多数的编程任务，你可以使用 Emacs 内置的 Info 浏览器。它的调用命令是 **M-x info** 或 **C-h i**。

1.5.3 包含文件

你可以通过阅读系统包含文件来获取与系统函数相关的知识。这些包含文件位于 **/usr/include** 和 **/usr/include/sys** 目录下。比如当你在使用系统调用的时候出现了编译错误，你可以直接去查看相应的包含文件以确保函数的签名与手册页中所说的是一致的。

在 Linux 系统中，关于系统调用如何工作的各种细节信息都会从一些包含文件中反应出来；它们位于 **/usr/include/bits**、**/usr/include/asm** 和 **/usr/include/sys** 目录下。例如，各种信号（在第三章“进程”3.3 节“信号”中进行了介绍）分别对应的数量值均定义在包含文件 **/usr/include/bits/signum.h** 中。对于想知根知底的人来说，这些文件是非常值得阅读的。不过，不要在你的程序中直接包含这些文件；应该使用使用 **/usr/include** 中的包含文件，除非是你所使用的函数的手册页中特别指明的。

1.5.4 源码

它是开源的，对吧？对于解释系统如何运行这种问题，最终的仲裁者必然是系统本身的代码。对于 Linux 程序员来说，非常幸运的是，这些代码是可以自由获取的。通常，你的 Linux 发行版中可能已经包含了整个系统和各种程序的完整源码。如果没有，根据 GNU General Public License，你有权向系统发行者要求获取这些源码。（源码也可能没有被安装在你的硬盘上。请参阅发行版文档以寻找安装方法。）

通常 Linux 内核的源码被安放在 **/usr/src/linux** 目录中。如果本书使你对进程、共享内存和系统设备的工作方式产生了兴趣，你完全可以从这些源码中找到答案。本书中介绍的多数系统函数都包含在 GNU C 函数库中；请查看发行版文档以获取 C 库源码的安装位置。

第二章：编写优质 GNU/Linux 软件

本章介绍了多数 GNU/Linux 程序员会使用的一些基本技巧。如果在你的程序中遵循这些约定，你的程序就可以在 GNU/Linux 系统环境下很好地工作，并且能符合用户关于程序间协作的习惯与期望。

2.1 与运行环境交互

当你学习C或C++的时候，你会被告知main函数是程序的主入口点。当操作系统执行你的程序的时候，这个函数会自动帮助程序与操作系统和用户进行沟通。你也许知道**main**的两个参数，通常被命名为**argc**和**argv**。这两个参数帮助程序获取用户输入。你或许学过为程序提供终端输入输出的**stdout**和**stdin**（或C++程序中的**cout**和**cin**流）。这些功能分别由C和C++语言提供，且以某种方式与GNU/Linux系统交互。GNU/Linux系统也提供了其它的一些方式供程序与操作环境交互。

2.1.1 参数列表

你可以通过在命令 shell 的提示符后输入一个程序的名字来运行一个程序。你也可以选择在程序名后跟一个或多个用空格分隔的单词。这部分输入被称为*命令行参数（command-line arguments）*。你可以通过将一个参数用引号保护起来，使某个参数内可以包含空格。更加常见的说法是将它称为参数列表，因为这些参数未必是来自 shell 程序。在第三章“进程”中你将看到另外调用程序的方法。在这种方法中，一个程序将可以为被调用的程序直接指定参数列表。

当从 shell 调用一个程序的时候，参数列表中包含了整个命令行，包含了程序的名字和全部被指定的命令行参数。例如，假设你在你的 shell 中这样执行 **ls** 显示根文件夹的对应的大小：

```
% ls -s /
```

则 **ls** 程序得到的参数列表将包含三个参数。第一个参数是命令行中指定的程序名 **ls**。参数列表中的第二和第三个参数则是命令行中指定的 **-s** 和 **/** 这两个参数。

程序的 **main** 函数可以通过参数 **argc** 和 **argv** 来访问程序的参数列表（如果你不需要访问参数列表，你可以直接忽略它们）。第一个参数 **argc** 指示了命令行中参数的数量。第二个参数 **argv** 是一个字符串数组。数组的大小由 **argc** 指定，而数组的元素则为各个命令行参数的元素，表示以 NULL 结束的字符串形式。

使用命令行参数的过程因此被简化为检查 **argc** 和 **argv** 的内容。如果你对程序自己的名字没有兴趣，记得跳过第一个参数。

列表 2.1 展示了使用 **argc** 和 **argv** 的方法。

代码列表 2.1 （arglist.c）使用 argc 和 argv

```
#include <stdio.h>

int main (int argc, char* argv[])
```

```
{
  printf ("The name of this program is '%s'.\n", argv[0]);
  printf ("This program was invoked with %d arguments.\n", argc - 1);

  /* 指定了命令行参数么？*/
  if (argc > 1) {
    /* 有，那么输出这些参数。*/
    int i;
    printf ("The arguments are:\n");
    for (i = 1; i < argc; ++i)
      printf (" %s\n", argv[i]);
  }

  return 0;
}
```

2.1.2 GNU/Linux 系统命令行使用习惯

几乎所有 GNU/Linux 程序都遵循一些处理命令行参数的习惯。程序期望得到的参数可以被分为两种：*选项（options，又作 flags）*和其它（非选项）参数。选项用于调整程序的行为方式，而其它参数提供了程序的输入（例如，输入文件的名字）。

选项通常有两种格式：

- *短选项（short options）*由一个短杠（*hyphen*）和一个字符（通常为一个大写或小写字母）组成。短选项可以方便用户的输入。
- *长选项（long options）*由两个短杠开始，之后跟随一个由大小写字母和短杠组成的名字。长选项方便记忆和阅读（尤其在脚本中使用的时候）。

通常程序会为它支持的选项提供长短两种形式，前者为便于用户理解，而后者为简化输入。例如，多数程序都能理解 **–h** 和 **–help** 这两个参数，并以相同方法进行处理。一般而言，当从 shell 中调用一个程序的时候，程序名后紧跟的就是选项参数。如果一些选项需要一个参数，则参数紧跟在选项之后。例如，许多程序将选项 **–output foo** 解释为“将输出文件设置为 **foo**”。在选项之后可能出现其它命令行参数，通常用于指明输入文件或输入数据。

例如，命令行 **ls –s /** 列举根目录的内容。选项 **–s** 改变了 **ls** 的默认行为方式，通知它为每个条目显示文件大小（以 KB 为单位）。参数 **/** 向 **ls** 指明了被列举的目录。选项 **–size** 与 **–s** 选项具有相同的含义，因此调用 **ls –size /** 会得到完全相同的结果。

在 *GNU Coding Standards（《GNU 编码标准》）*中列举了一些常用的命令行选项的名称。如果你准备提供与它们相似的选项，你应该选择使用编码标准中指定的名字。这样你的程序会与其它程序更相类似且更易于用户学习使用。在多数 GNU/Linux 系统中，你可以通过输入以下命令阅读《GNU 编码标准》中关于命令行选项的指导方针：

```
% info "(standards)User Interfaces"
```

2.1.3 使用 *getopt_long*

解析命令行参数是无聊而繁琐的。幸运的是，GNU C 库提供了函数以简化 C/C++程序中的解析工作（虽然还是有点麻烦）。函数 **getopt_long** 能够处理长短两种格式的选项。要使用这个函数，请包含头文件**<getopt.h>**。

例如，假设一个程序接受表格 2.1 中列举的三个参数：

表格 2.1 程序选项示例

短格式	长格式	作用
-h	--help	显示使用方法，然后退出。
-o filename	--output filename	指定输出文件名。
-v	--verbose	输出冗余信息。

此外，程序还会接受零个或多个命令行参数作为输入文件。

你需要提供两个数据结构以使用 **getopt_long**。第一个数据结构是一个字符串，其中包含了所有有效的短格式选项，每个字母表示一个。如果一个选项要求一个参数标记为在名称后加一个冒号。对于这个程序而言，字符串 **ho:v** 指明了程序的可用选项包括 **–h**，**-o** 和 **–v**，其中第二个选项要求一个参数。

要指明程序接受的长选项，你需要建立一个 **struct option** 类型的数组。数组的每个元素都针对一个长选项且每个元素都具有四个域。一般情况下，第一个域是长选项的名字（表示为一个字符串；不包含选项开始的两个短杠）；第二个参数如果为 1 则表示该选项接受一个参数，否则为 0；第三个域指定为 NULL；第四个参数则为一个字符常量，保存了相同含义的短选项名。数组中最后一个元素的所有域都应为 0。你可以这样初始化这个数组：

```
const struct option long_options[] = {
  { “help”,      0,  NULL,   ‘h’ },
  { “output”,    1,  NULL,   ‘o’ },
  { “verbose”,   0,  NULL,   ‘v’ },
  { NULL,        0,  NULL,   0   }
};
```

当你调用 **getopt_long** 的时候，并且传递给它以下这些参数：

1. **main** 函数的参数 **argc** 和 **argv**；
2. 描述短选项的字符串；
3. 描述长选项的 **struct options** 数组。

当使用 **getopt_long** 的时候：

- 每次调用 **getopt_long** 的时候，它解析一个选项，返回这个选项对应的短格式字母。如果没有其它选项则返回 -1。
- 这个函数的典型用法是在一个循环中不断调用以处理用户指明的所有选项，且程序在一个 **switch** 语句中分别处理每个选项。
- 如果 **getopt_long** 检测到一个无效选项（一个没有被指定为任何长短选项的选项），它会输出一条错误消息并返回字符’?’（一个英文问号）。多数程序通常会在这种情况下显示使用帮助并退出。
- 当处理一个带参数的选项时，全局变量 **optarg** 将指向参数字符串的开始。
- 当 **getopt_long** 结束处理所有选项之后，全局变量 **optarg** 包含了第一个非选项参数在 **argv** 中的索引。

列表 2.2 中展示了一个使用 **getopt_long** 处理参数的示例程序。

代码列表 2.2 （getopt_long.c）使用 getopt_long

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

/* 程序的名称。*/
const char* program_name;

/* 将程序使用方法输出到 STREAM 中（通常为 stdout 或 stderr），并以 EXIT_CODE 为返回值退出程序。函数调用不会返回。*/
void print_usage (FILE* stream, int exit_code)
{
  fprintf (stream, “Usage: %s options [ inputfile ... ]\n”, program_name);
  fprintf (stream,
           “   -h --help Display this usage information.\n”
           “   -o --output filename Write output to file.\n”
           “   -v --verbose Print verbose messages.\n”);
  exit (exit_code);
}


/* 程序主入口点。ARGC 包含了参数列表中元素的数量；ARGV 是指向这些参数的指针数组。*/
int main (int argc, char* argv[])
{
  int next_option;

  /* 包含所有有效短选项字母的字符串。*/
  const char* const short_options = “ho:v”;

  /* 描述了长选项的 struct option 数组。*/
  const struct option long_options[] = {
    { “help”, 0, NULL, ‘h’ },
    { “output”, 1, NULL, ‘o’ },
    { “verbose”, 0, NULL, ‘v’ },
    { NULL, 0, NULL, 0 } /* 数组末要求这样一个元素。*/
  };

  /* 用于接受程序输出的文件名，如果为 NULL 则表示标准输出。*/
  const char* output_filename = NULL;

  /* 是否显示冗余信息？*/
  int verbose = 0;
```

```
    /* 记住程序的名字，可以用于输出的信息。名称保存在 argv[0]中。*/
    program_name = argv[0];

    do {
      next_option = getopt_long (argc, argv, short_options,
                                 long_options, NULL);
      switch (next_option)
      {
      case 'h': /* -h 或 --help */
        /* 用户要求查看使用帮助。输出到标准输出，退出程序并返回 0（正常结束）。*/
        print_usage (stdout, 0);
      case 'o': /* -o 或 --output */
        /* 此函数接受一个参数，表示输出文件名。*/
        output_filename = optarg;
        break;
      case 'v': /* -v 或 --verbose */
        verbose = 1;
        break;
      case '?': /* The user specified an invalid option. */
        /* 向标准错误输出帮助信息，结束程序并返回 1（表示非正常退出）。*/
        print_usage (stderr, 1);
      case -1: /* 结束处理选项的过程。*/
        break;
      default: /* 别的什么：非预料中的。*/
        abort ();
      }
    }
    while (next_option != -1);

    /* 选项处理完毕。OPTIND 指向第一个非选项参数。
       出于演示目的，如果指定了冗余输出选项，则输出这些参数。*/
    if (verbose) {
      int i;
      for (i = optind; i < argc; ++i)
        printf ("Argument: %s\n", argv[i]);
    }
  /* 主程序到这里结束。*/
  return 0;
}
```

使用 **getopt_long** 看起来需要不少的工作量，但是如果你尝试自己写代码解析这些参数，你就会发现这会浪费更多的代码。函数 **getopt_long** 已经过反复的考验，而且它在程序选择接受何种参数方面提供了出众的灵活性。不过，最好还是不要轻易去使用那些高级技巧而斤

两使用上面介绍的基本用法。

2.1.4 标准 I/O

标准 C 库提供了标准输入和输出流（分别为 **stdin** 和 **stdout**）。它们被用于 **scanf**、**printf** 和其它库函数中。在 UNIX 传统中，程序习惯使用标准输入输出进行沟通。这种习惯允许多个程序通过管道和重定向进行串连。（参考你使用的 shell 的手册页以学习相关语法。）

C 库还提供了标准错误流 stderr。程序应该将警告和错误信息输出到标准错误流而不是标准输出流。这样方便了用户区别对待普通程序输出和错误输出，比如将标准输出重定向到一个文件而让标准错误显示在终端里。可以通过 **fprintf** 函数向标准错误流输出信息：

```
fprintf (stderr, (“Error: …”));
```

这三个流也可以文件描述符的形式通过底层 UNIX I/O 命令（**read**、**write** 等）进行操作。文件描述符 0 代表标准输入，1 为标准输出而 2 为标准错误。

当调用一个程序的时候，有时候需要将标准输出和错误同时重定向到一个文件或管道。不同 shell 为这个操作提供了不同的语法；对于类似 Bourne shell 的 shell 程序（包括多数 GNU/Linux 系统的默认 shell 程序 **bash**）语法是这样的：

```
% program > output_file.txt 2>&1
% program 2>&1 | filter
```

这里，**2>&1** 的语法表示文件描述符 2（**stderr**）应并入文件描述符 1（**stdout**）。注意，**2>&1** 这个语法必须出现在文件重定向之前（如第一个例子所示）或者管道重定向之前（如第二个例子所示）。

需要注意的是，**stdout** 是经过缓冲处理的。写入 **stdout** 的数据不会立刻被写入终端（或其它设备，如果程序输出被重定向）除非缓冲区满、程序正常退出或 **stdout** 被关闭。你可以这样显式地刷新输出流：

```
fflush (stdout);
```

与**stdout**不同的是，**stderr**没有经过缓冲处理；输出到**stderr**的数据会直接被发送到终端。[1]

这可能导致令人惊奇的结果。例如下面这个程序，运行时并不会每一秒钟输出一个句点，而是会在缓冲被填满的时候一起输出一堆。

```
while (1) {
  printf (“.”);
  sleep (1);
}
```

在这个循环中句点则会每秒钟输出一个。

```
while (1) {
  fprintf (stderr, “.”);
  sleep (1);
}
```

[1] 在C++中，cout和cerr之间也有这样的区别。注意endl操作符除了输出换行符，还会执行刷新操作；如果你不希望执行刷新操作（例如出于运行效率的考虑）则应该使用常量'\n'表示换行。

2.1.5 程序退出代码

当一个程序结束的时候，它会通过一个退出代码表示自己的运行结果。退出代码是一个小整数值。一般的习惯是，返回 0 表示正常，而非 0 表示错误的出现。一些程序通过不同的非 0 值表示不同的错误情况。

在许多 shell 中，可以通过特殊环境变量**$?**得到最近执行的一个程序的退出代码。下面这个例子中，**ls** 命令被执行了两次，每次执行完毕之后我们都输出了命令的退出代码。第一次调用中，**ls** 成功执行且返回 0。第二此运行的时候 **ls** 在运行中出现了错误（因为在命令行中指定的文件不存在），并因此返回了非 0 值作为退出代码。

```
% ls /
bin   coda  etc   lib         misc  nfs   proc  sbin  usr
boot  dev   home  lost+found  mnt   opt   root  tmp   var
% echo $?
0
$ ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

C 或 C++程序通过从 **main** 函数返回来指定程序的退出代码。还可以通过其它的方法指定程序的退出代码，且特殊的退出代码被分配用于标识特殊的程序退出原因（被信号终止等）我们将在第三章中对这些情况进行深入的讨论。

2.1.6 环境

GNU/Linux 为每个运行程序提供了一个*环境（environment）*。环境是一组“键－值”对的集合。环境变量名和它们的值都是字符串。环境变量名通常由大写字母组成。

你可能已经对一些常见的环境变量有所熟悉。例如：

- **USER** 包含了你的用户名。
- **HOME** 包含了你的个人目录（*home directory*）的位置。
- **PATH** 包含了一些文件夹路径，之间由冒号进行分隔。Linux 系统在这些文件夹中搜索可执行程序。
- **DISPLAY** 包含了 X 窗口服务器的名称和显示器编号。这里指定的 X 服务器和显示器编号将是基于 X 的图形程序运行时将会出现的地方。

Shell 和其它所有程序一样，都有一个环境。Shell 提供了直接查看和修改环境的方法。可以使用 **printenv** 程序输出完整的当前环境。不同的 shell 程序通过不同的内建语法使用环境变量的值；以下示例使用的是 Bourne 式的 shell。

- Shell 会自动为每个检测到的环境变量设置一个 Shell 变量，因此你可以通过**$变量名**的语法访问环境变量。例如：

  ```
  %echo $USR
  samuel
  % echo $HOME
  /home/samuel
  ```

- 可以通过 **export** 命令将一个 shell 变量加入环境中。例如，可以这样设置环境变

量 **EDITOR** 的值：

```
% EDITOR=emacs
% export EDITOR
```

或

```
% export EDITOR=emacs
```

程序中可以使用**<stdlib.h>**中提供的 **getenv** 函数访问环境变量。这个函数接受一个包含变量名的字符串作为参数，并返回包含了相应的环境变量值的字符串。如果参数中指定的环境变量不存在，**getenv** 将返回 NULL。而 **setenv** 和 **unsetenv** 函数则分别可用于设置和清除环境变量。

列举所有环境变量需要一点技巧。你需要通过访问一个叫做 **environ** 的全局变量来列举所有环境变量。这个变量是由 GNU C 库定义的。它是一个 **char ****类型的变量，包含了一个以 NULL 指针结束的字符串数组。每个字符串都包含了一个环境变量。这个环境变量被表示为“**变量=值**”的形式。

请看下面的例子。列表 2.3 中的程序通过一个循环遍历整个 **environ** 数组并输出所有环境变量。

代码列表 2.3 （print-env.c）输出运行环境

```
#include <stdio.h>

/* ENVIRON 变量包含了整个环境。*/
extern char** environ;

int main ()
{
  char** var;
  for (var = environ; *var != NULL; ++var)
    printf (“%s\n”, *var);
  return 0;
}
```

不要直接修改 **environ** 变量；如果需要修改环境变量，则应通过 **setenv** 和 **unsetenv** 函数完成。

通常，当启动一个新程序的时候，这个程序会从调用者那里继承一份运行环境（在交互运行的情况下，通常调用者是 shell 程序）。因此，你从 shell 中运行的程序可以使用你通过 shell 设置的环境变量。

环境变量常被用于向程序提供配置信息。假设你正在写一个程序，它需要连接到一台 Internet 服务器并获取一些信息。程序可以利用命令行参数获取服务器地址。但是，如果用户不会需要经常改变服务器地址，那么你可以选择将服务器地址存储在一个特殊的环境变量中（譬如 **SERVER_NAME**）。如果这个环境变量不存在则使用一个默认值。这个程序的部分可能像这个样子：

代码列表 2.4 （client.c）一个网络客户程序的片断

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

int main ()
{
  char* server_name = getenv (“SERVER_NAME”);
  if (server_name == NULL)
    /* 环境变量 SERVER_NAME 不存在。使用默认值。*/
    server_name = “server.my-company.com”;

  printf (“accessing server %s\n”, server_name);
  /* 在这里访问服务器。*/

  return 0
}
```

假设这个程序叫 **client**。假设你还没有设置 **SERVER_NAME** 变量，则程序会使用默认值进行连接：

```
% client
accessing server server.my-company.com
```

但修改要连接到的服务器也很容易：

```
% export SERVER_NAME=backup-server.elsewhere.net
% client
accessing server backup-server.elsewhere.net
```

2.1.7 使用临时文件

有时候程序需要使用临时文件，用来缓存或者向别的程序传递大量的数据。在 GNU/Linux 系统中，临时文件被存储在 **/tmp** 文件夹下。当使用临时文件的时候，你需要注意以下的问题：

- 同一个程序的多个副本可能正在（由同一个用户或不同的用户）并行运行。每个副本都应该使用不同的临时文件以避免冲突。
- 文件权限的设置应当保证临时文件不会被未被授权的用户修改或替换，从而导致程序行为被改变。
- 生成的临时文件名应该不可被外界预料；否则，攻击者可能会在程序检测一个文件名是否被占用与实际打开临时文件进行读写之间的间隔进行攻击。

GNU/Linux 提供了 **mkstemp** 和 **tmpfile** 两个函数以帮助你解决这些问题（以辅助使用其它一些仍需要面临这些问题的函数）。两者之间的选择取决于你对文件操作的要求：是否准备将临时文件转交给其它程序？使用 UNIX I/O（**open**、**write** 之类）还是标准 C 库的 I/O（**fopen**、**fprintf** 之类）？

使用 *mkstemp*

函数 **mkstemp** 从一个文件名模板生成临时文件名，创建这个临时文件，将模式设置为仅当前用户可以访问，并且以读写权限打开这个文件。文件名模板是一个字符串，其结尾应为“XXXXXX”（六个大写字母 X）；**mkstemp** 函数用其它字符替换这些 X 以得到一个不重复的文件名。函数返回已打开的文件描述符；可以通过 **write** 族的函数对它执行写入操作。

由 **mkstemp** 创建的临时文件是不会被自动删除的。是否删除、以及在何时删除这些临时文件完全取决于你。（程序员应该时刻记住及时清理临时文件，否则一旦 **/tmp** 文件夹被填满，系统将不可用。）如果这个临时文件只是程序内部使用而不会移交给其它程序，在创建之后立刻调用 **unlink** 是个不错的主意。这个函数会从目录中移除对应的文件项，但是文件系统中的文件是有引用计数的，因此只有当所有指向该文件的描述符都被关闭的时候，它才会被文件系统真正删除。通过这个方法，你的程序可以继续使用这个临时文件，而这个临时文件会在你关闭文件描述符的时候被清除出系统。因为 Linux 系统会在程序结束的时候关闭所有文件描述符，即使你的程序被异常终止，临时文件仍然会被删除。

列表 2.5 中的两个函数展示了 **mkstemp** 的使用。这两个函数一起使用可以简化将一块内存缓冲区写入临时文件（以便释放内存或将内存挪作它用）以及从临时文件读回内容的过程。

代码列表 2.5 （temp_file.c）使用 mkstemp

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

/* 用于保存 write_temp_file 创建的临时文件的句柄类型。
   在这个实现中它是一个文件文件描述符。*/
typedef int temp_file_handle;

/* 将 BUFFER 中的 LENGTH 字节内容写入临时文件。
   临时文件会立刻被执行 unlink 操作。
   返回指向临时文件的句柄。*/
temp_file_handle write_temp_file (char* buffer, size_t length)
{
  /* 创建文件名和文件。XXXXXX 会被替换以生成不重复的文件名。*/
  char temp_filename[] = “/tmp/temp_file.XXXXXX”;
  int fd = mkstemp (tmep_filename);
  /* 立刻进行 unlink，从而使这个文件在我们关闭描述符的时候被删除。*/
  unlink (temp_filename);
  /* 将数据的长度写入文件。*/
  write (fd, &length, sizeof (length));
  /* 现在再写入数据本身。*/
  write (fd, buffer, length);
  /* 将文件描述符作为指向文件的句柄。*/
  return fd;
}
```

```
/* 将 write_temp_file 建立的临时文件 TEMP_FILE 中的内容读回。
   函数返回一块新分配的缓冲区，其中包含了这些内容。调用者必须用 free 释放这个缓冲区。
   *LENGTH 被设置为数据的长度，以字节计。临时文件最后被删除。
*/
char* read_temp_file (temp_file_handle temp_file, size_t* length)
{
  char* buffer;
  /* 句柄 TEMP_FILE 是一个指向临时文件的描述符。*/
  int fd = temp_file;
  /* 回滚到文件的开始。*/
  lseek (fd, 0, SEEK_SET);
  /* 读取数据的长度。*/
  read (fd, length, sizeof(*length));
  /* 分配缓冲区，然后读取数据。*/
  buffer = (char*) malloc (*length);
  read (fd, buffer, *length);
  /* 关闭文件描述符。这会导致临时文件被从系统中删除。*/
  close (fd);
  return buffer;
}
```

使用 *tmpfile*

如果你使用的是 C 库 I/O 函数，且你不需要向其它程序传递这个临时文件，你可以使用 **tmpfile** 函数。这个函数创建并打开一个临时文件并返回一个对应的文件指针。如前例中所示，这个临时文件已经被 unlink，因此当文件指针被关闭（通过 **fclose**）或程序结束的时候临时文件将被自动删除。

GNU/Linux 提供了其它一些用于生成临时文件或文件名的函数，包括 **mktemp**、**tmpnam** 和 **tempnam** 等。不要使用这些函数，因为这些函数在可靠性和安全性方面存在不足。

2.2 防御性编码

写一个程序在“普通”状态下正常运行不是一件容易的事情；要让程序在运行出错的情况下表现得优雅就更难了。本节中介绍了一些技巧和方法能够帮助程序员在编码阶段尽早发现错误，以及使程序在运行中检测和恢复错误状况的出现。

本书后面章节中的代码都有意省略了全面的错误检测与恢复代码，因为这些代码可能掩盖所介绍的特性。不过在十一章“一个 GNU/Linux 样板应用程序”的最后的示例中，我们回过头展示了如何利用这些技巧建立强壮的程序。

2.2.1 使用 assert

当构造一个应用程序的时候，应该始终记住：应该让程序在出现 bug 或非预期的错误的时候，应该让程序尽可能早地突然死亡。这样做可以帮助你在开发——测试循环中尽早地发现错误。不导致突然死亡的错误将很难被发现；它们通常会被忽略，直到程序在客户系统中

运行以后才被注意到。

检查非预期状态的最简单的方式是通过标准 C 库的 **assert** 宏。这个宏的参数是一个布尔表达式（*Boolean expression*）。当表达式的值为假的时候，**assert** 会输出源文件名、出错行数和表达式的字面内容，然后导致程序退出。**Assert** 宏可用于大量程序内部需要一致性检查的场合。例如，可以用 **assert** 检查程序参数的合法性、检查函数（或 C++中的类方法）的前提条件和最终状态（*postcondition*）、检查非预期的函数返回值，等等。

每次使用 **assert** 宏，不仅可以作为一项运行期的检查，还可以被当作是嵌入代码中的文档，用于指明程序的行为。如果你的程序中包含了 **assert(*condition*)**，它就是在告诉阅读代码的人：***condition*** 在这里应该始终成立；否则很可能是程序中的 bug。

对于效率至上的代码，**assert** 这样的运行时检查可能引入严重的效率损失。在这种情况下，你可以定义 **NDEBUG** 宏并重新编译源码（可以通过在编译器参数中添加 **–DNDEBUG** 参数做到）。在这种情况下，**assert** 宏的内容将被预处理器清除掉。应该只在当效率必须优先考虑的情况下，对包含效率至上的代码的文件设置 **NDEBUG** 宏进行编译。

因为 **assert** 可能被预处理过程清除，当使用这个宏的时候必须确信条件表达式不存在副作用。特别的，不应该在 **assert** 的条件表达式中使用这些语句：函数调用、对变量赋值、使用修改变量的操作符（如 ++ 等）。

例如，假设你在一个循环中重复调用函数 **do_something**。这个函数在成功的情况下返回 0，失败则返回非 0 值。但是你完全不期望它在程序中出现失败的情况。你可能会想这样写：

```
for (i = 0; i < 100; ++i)
  assert (do_something () == 0);
```

不过，你可能发现这个运行时检查引入了不可承受的性能损失，并因此决定稍候指定 **NDEBUG** 以禁用运行时检测。这样做的结果是整个对 **assert** 的调用会被完全删除，也就是说，**assert** 宏的条件表达式将永远不会被执行，**do_something** 一次也不会被调用。因此，这样写才是正确的：

```
for (i = 0; i < 100; ++i) {
  int status = do_something ();
  assert (status == 0);
}
```

另外一个需要记住的是，不应该使用 **assert** 去检测不合法的用户输入。用户即使在输入不合适的信息后也不希望看到程序仅在输出一些含义模糊的错误信息后崩溃。你应该检查用户的非法输入并向用户返回可以理解的错误信息。只当进行程序内部的运行时检查时才应使用 **assert** 宏。

一些建议使用 **assert** 宏的地方：

- 检查函数参数的合法性，例如判断是否为 NULL 指针。由 **{ assert (pointer != NULL)}** 得到的错误输出

  ```
  Assertion ‘pointer != ((void *)0)’ failed.
  ```

  与当程序因对 NULL 指针解引用得到的错误信息

  ```
  Segmentation fault (core dumped)
  ```

  相比而言要清晰得多。
- 检查函数参数的值。例如，当一个函数的 foo 参数必须为正数的时候我们可以在函数开始处进行这样的检查：

```
assert (foo > 0);
```

这会帮助你发现错误的调用；同时它很清楚地告诉了读代码的人：这个函数对参数的值有特殊的要求。

不要就此退缩；在你的程序中适当地时候使用 **assert** 宏吧。

2.2.2 系统调用失败

通常我们都学会了写一个程序并期望它沿着一条预知的路线顺利执行完毕。我们将一个程序分解成不同的任务和子任务；每个函数通过调用其它函数，逐级完成各种任务。通过提供合适的输入，我们期望程序产生合理的输出及副作用。

可惜，现实中的电脑硬件和软件将这种期待彻底地粉碎。系统资源的有限、硬件故障、许多程序的并发执行、用户和程序员制造的错误……这些残酷的现实往往在应用程序与操作系统交界的地方暴露出来。因此，当通过系统调用访问系统资源、执行 I/O 或其它命令的时候，除了理解执行成功的意义，更应该明白出现错误可能出现的时机和原因。

系统调用可能由于各种原因而失败。例如：

- 系统可能出现资源短缺（或程序用尽了系统为单个程序设置的资源使用限制）。例如，程序正在尝试分配过多的内存，向磁盘写入过多信息或同时打开过多的文件，等等。
- 当程序尝试执行某些自身权限不允许的操作的时候，系统可能会致使执行失败。例如，程序尝试向一个设置为只读权限的文件写入信息、访问属于另外一个进程的内存或杀死其它用户运行的程序等。
- 系统调用可能由于程序之外的原因而失败。这种情况常见于通过系统调用访问硬件设备的时候。这个设备可能不能正常工作、可能不支持特定的操作，也可能是处于类似“没有将磁碟插入驱动器”这样的情况中。
- 系统调用可能因为外部的事件（如信号等）而被中断。这可能不代表彻底的执行失败，但是如果需要，程序应当重新尝试执行系统调用。

在一个好的、大量使用系统调用的程序中，很多情况下，多数的代码将被用于检测、处理错误和其它意外情况，而不是用于完成程序的主要任务。

2.2.3 系统调用返回的错误码

多数系统调用在成功执行之后返回 0，而在失败的时候返回非 0 值。（不过，也有许多函数通过返回值表示不同的意义；例如，**malloc** 在执行失败后返回空指针。在使用系统调用之前，务请仔细阅读相关的手册页。）（译者注：前文提到的 **malloc** 函数并不属于一般意义上的系统调用，而是 Glibc 提供的库函数。）尽管这些信息足够用于判定程序是否应该继续执行，它并不足以让程序清楚地了解如何对已经出现的错误进行恢复和处理。

多数系统调用在执行失败的时候会通过一个特殊的全局变量**errno**保存错误相关的信息[2]。当执行失败的时候，系统调用会将**errno**设置为特定的值以指示错误原因。因为所有系统调用均会使用**errno**指示出现的错误，你应该在出现错误之后立刻将**errno**的值保存到其它的变量中。在此之后的系统调用过程可能在出错的情况下覆盖现有的值。

错误代码是整型的；可能值均通过预处理宏指定了。这些宏的命名均为错误的约定名字

[2] 实际上，出于线程安全的考虑，**errno**会被实现为一个宏；但是使用方式与一个普通的全局变量并无二致。

的全部大写形式加上前缀 **E** 构成——例如“**EACCES**”和“**EINVAL**”等。始终应该通过这些宏表示 **errno** 的值，而不是直接使用整数。如果要使用这些错误代码，则需要在程序中包含**<errno.h>**。

GNU/Linux 提供了 **strerror** 函数以简化错误处理。这个函数用于返回一个 **errno** 错误代码对应的说明性字符串。这个字符串可以用于程序输出的错误信息中。使用 **strerror** 需要在程序中包含**<string.h>**。

GNU/Linux 还提供了 **perror** 函数。这个函数会直接将错误信息输出到 **stderr** 流。传递给 **perror** 的参数是一个前缀字符串；当函数执行的时候，这个字符串将作为错误信息的前缀输出，因此通常可以在这个字符串中说明出错的函数。使用 **perror** 需要在程序中包含**<stdio.h>**。

在这个代码片断中我们尝试打开一个文件；如果打开文件操作失败，它会输出错误信息并退出程序。注意系统调用 **open** 在成功时返回一个文件描述符，而在失败时返回 -1。

```
fd = open (“inputfile.txt”, O_RDONLY);
if (fd == -1) {
  /* 对 open 的调用失败。输出错误信息并退出。*/
  fprintf (stderr, “error opening file: %s\n”, strerror (errno));
  exit (1);
}
```

取决于你的程序和系统调用本身的特性，当出错的时候你可以选择输出错误信息、取消一个操作、强制退出程序、重新尝试调用甚至是直接忽略这个错误。不过，包含一段通过某种方式处理所有错误情况的程序逻辑是非常必要的。

有一个你始终应该注意（尤其在执行 I/O 操作的时候）的错误代码是 **EINTR**。某些函数，如 **read**、**write** 和 **sleep** 等，可能需要很长时间才能执行完毕。这些函数被成为*阻塞（blocking）*函数，因为程序的执行会被阻塞直到这个函数调用结束。但是，当程序在阻塞过程中收到一个信号，这些函数可能不等执行完毕就直接返回。在这种情况下，**errno** 被设置为 **EINTR**。通常在这种时候你应该尝试重新执行这个操作。

下面的代码片断中，我们利用 **chown** 函数将 **path** 指定的文件的属主改为 **user_id** 指明的用户。如果调用失败，程序将根据 **errno** 的值作出不同的响应。注意，当我们检测到一个可能是程序 bug 的错误情况时，我们通过 **abort** 或 **assert** 退出程序；这样会导致生成一个核心转储文件（core file）。这对于事后调试（post-mortem debugging）将很有用。对于其它不可恢复的错误，我们通过 **exit** 与一个非 0 返回值表明错误的出现；因为这种情况下核心转储文件并不会带来更多的好处。

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  /* 保存 errno 的值，因为下次系统调用将可能抹去这个值。*/
  int error_code = errno;
  /* 操作失败；chown 会在执行出错时返回 -1。*/
  assert (rval == -1);
  /* 检查 errno 的值，并作出相应的处理。*/
  switch (error_code) {
  case EPERM:       /* 权限不足。*/
  case EROFS:       /* PATH 位于一个只读文件系统中。*/
```

```
    case ENAMETOOLONG: /* PATH 长度超过限度。*/
    case ENOENT:       /* PATH 指定的路径不存在。*/
    case ENOTDIR:      /* PATH 的某些部分不是文件夹。*/
    case EACCES:       /* PATH 的某些部分无法访问。*/
      /* 文件有些问题。输出错误信息。*/
      fprintf (stderr, "error changing ownership of %s: %s\n",
               path, strerror (error_code));
      /* 不要结束程序；可能可以给用户提供一个重新选择文件的机会……*/
      break;

    case EFAULT:
      /* PATH 包含了非法的内存地址。这可能是一个程序 bug。*/
      abort ();

    case ENOMEM:
      /* 内核内存不足。*/
      fprintf (stderr, "%s\n", strerror (error_code));
      exit (1);

    default:
      /* 其它一些非预期的错误代码。我们已经尝试处理所有可能的错误代码；
         如果我们忽略了某个代码，这将是一个 bug! */
      abort ();

    };
  }
```

你可以选择使用下面这段代码；当运行成功的时候它的行为方式与上面的程序相同。

```
rval = chown (path, user_id, -1);
assert (rval == 0);
```

但如果调用失败，这段代码无法对错误进行汇报、处理及恢复。

选择第一种、第二种或是两者之间的某种错误处理方式，完全取决于你的程序对错误检测和处理方面的要求。

2.2.4 错误与资源分配

很多情况下，当一个系统调用失败的时候程序可以选择取消当前的操作而不终止运行，因为这些错误是可以恢复的。一种方式是从当前函数中返回，通过将错误代码传回给调用者表示错误的出现。

如果你决定从函数当中返回，必须确保函数中已经成功分配的资源必须首先被释放。这些资源可能包括内存、文件描述符、文件指针、临时文件、同步对象等。否则，如果你的程序继续保持运行，这些资源将被泄漏。

考虑这个例子，一个函数将一个文件的内容读入缓冲区。这个函数可能会按照这个步骤执行：

1. 分配缓冲区。
2. 打开文件。
3. 将文件读入缓冲区。
4. 关闭文件。
5. 返回缓冲区。

如果文件不存在，第二步将失败。从这个函数中返回 NULL 是一种可选的对策。但是，如果缓冲区已经在第一步中完成分配，直接返回可能导致内存的泄漏。你必须记得在返回之前的流程中释放这个缓冲区。如果程序在第三步出错，不仅是缓冲区，你还必须记住关闭文件。

列表 2.6 展示了这个函数的一种实现。

代码列表 2.6 （readfile.c）在异常情况下释放资源

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

char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  ssize_t bytes_read;

  /* 分配缓冲区。*/
  buffer = (char*) malloc (length);
  if (buffer == NULL)
    return NULL;
  /* 打开文件。*/
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    /* open 失败。在返回之前释放缓冲区。*/
    free (buffer);
    return NULL;
  }

  /* 读取数据。*/
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    /* read 失败。释放缓冲区，关闭文件，然后返回。*/
    free (buffer);
    close (fd);
```

```
    return NULL;
  }
}
```

Linux 会在程序退出的时候清理分配的内存、打开的文件和其它绝大多数资源，因此在调用 **exit** 之前释放缓冲区并关闭文件。不过，你可能需要手工释放其它资源，如临时文件和共享内存等——这些资源在进程结束后仍然存在。

2.3 编写并使用程序库

差不多可以认为，每个程序都链接到一个或几个库上。任何一个使用了 C 函数（诸如 **printf** 等）都须链接到 C 运行时库。如果你的程序具有图形界面（GUI），它将被链接到窗口系统的库。如果你的程序使用了数据库，数据库供应商会提供给你一些简化访问数据库的库。

在这些情况中，你必须作出选择：*静态（statically）*还是*动态（dynamically）*地将程序链接到库上。如果你选择了静态链接，程序体积可能会更大，程序也会比较难以升级，但是可能相对而言比较易于部署。如果你选择动态链接，则程序体积会比较小、易于升级，但是部署的难度将会有所提高。本节中我们将介绍静态和动态两种链接方法，仔细比较它们的优劣，并提出一些在两者之间选择的简单的规则。

2.3.1 存档文件

存档文件（archive），也被称为静态库（static library），是一个存储了多个对象文件（object file）的单一文件。（与 Windows 系统的 **.LIB** 文件基本相当。）编译器得到一个存档文件后，会在这个存档文件中寻找需要的对象文件，将其提取出来，然后与链接一个单独的对象文件一样地将其链接到你的程序中。

你可以使用 **ar** 命令创建存档文件。传统上，存档文件使用 **.a** 后缀名，以便与 **.o** 的对象文件区分开。下面的命令可以将 **test1.o** 和 **test2.o** 合并成一个 **libtest.a** 存档：

```
% ar cr libtest.a test1.o test2.o
```

上面命令中的**cr**选项通知**ar**创建这个存档文件[3]。现在你可以通过为**gcc**或**g++**指定 **–ltest** 参数将程序链接到这个库。这个过程在第一章“起步”1.2.2 节“链接对象文件”中进行了说明。

当链接器在命令行参数中获取到一个存档文件时，它将在其中搜索所有之前已经被引用而没有被定义的符号（函数或变量）的定义。定义了这些符号的对象文件将从存档中被提取出来，链接到新程序执行文件中。因为链接器会在读取命令行参数的过程中一遇见存档文件就进行解析，通常将存档文件放在命令行参数的最后最有意义。例如，假设 **test.c** 包含了列表 2.7 中的代码而 **app.c** 包含了列表 2.8 中的代码。

代码列表 2.7 （test.c）库内容

```
int f ()
```

[3] 还有其它一些选项，包括从存档中删除一个文件或者执行其它操作。这些操作很少用，不过在**ar**手册页中都有说明。

```
{
  return 3;
}
```

代码列表 2.8 （app.c）一个使用库函数的程序

```
int main ()
{
  return f ();
}
```

现在假设我们将 **test.o** 与其它一些对象文件合并生成了 **libtest.a** 存档文件。下面的命令行不会正常工作：

```
% gcc -o app -L. -ltest app.o
app.o: In function ‘main’:
app.o(.text+0x4): undefined reference to ‘f’
collect2: ld returned 1 exit status
```

错误信息指出虽然 **libtest.a** 中包含了一个 **f** 的定义，链接器并没有找到它。这是因为 **libtest.a** 在第一次出现在命令行的时候就被搜索了，而这个时候链接器并没有发现对 **f** 的任何引用。

而如果我们稍微更改一下命令行，则不会再有错误消息出现：

```
% gcc -o app app.o -L. -ltest
```

这是因为 **app.o** 中对 **f** 的引用导致连接器将 **libtest.a** 中的 **test.o** 包含在生成的执行文件中。

2.3.2 共享库

*共享库（shared library，也被称为共享对象 shared object 或动态链接库 dynamically linked library）*在某种程度上与由一组对象文件生成的打包文件相当类似。不过，两者之间的区别也是非常明显的。最本质的区别在于，当一个共享库被链接到程序中的时候，程序本身并不会包含共享库中出现的代码。程序仅包含一个对共享库的引用。当系统中有多个程序链接到同一个共享库的时候，它们都将引用这个共享库而不是将代码直接包含在自身程序中——正因为如此，我们说这个库被所有这些程序“共享”。

第二个重要的区别在于，共享库不仅仅是对象文件的简单组合。当使用的时候，链接器会从中寻找需要的部分进行链接，以匹配未定义的符号引用。而当生成共享库的时候，所有对象文件被合成为一个单独的对象文件，从而使链接到这个库的程序总能包含库中的全部代码，而不仅仅是所需要的部分。

要创建一个共享库，你必须在编译那些用于生成共享库的对象时为编译器指定 **–fPIC** 选项。

```
% gcc -c -fPIC test1.c
```

这里的 **–fPIC** 选项会通知编译器你要将得到的 **test1.o** 作为共享库的一部分。

> **位置无关代码（Position-Independent Code）**
> 共享库中的函数在不同程序中可能被加载在不同的地址，因此共享库中的代码不能依赖特定的加载地址（或位置）。作为程序员，这并不需要你自己操心；你只需要在编译这些用于共享库的对象文件的时候，在编译器参数中指明 **–fPIC**。

然后你将得到的对象文件合并成一个共享库：

```
% gcc –shared –fPIC –o libtest.so test1.o test2.o
```

这里 **–shared** 选项通知链接器生成共享库，而不是生成普通的可执行文件。共享库文件通常使用 **.so** 作为后缀名，这里 **so** 表示共享对象（shared object）。与静态库文件相同，文件名以 **lib** 开头，表示这是一个程序库文件。

将程序链接到共享库与链接到静态库的方法并无二致。例如，当 **libtest.so** 位于当前目录或者某个系统默认搜索目录时，下面这条命令可以将程序与它进行链接：

```
% gcc –o app app-o –L. –ltest
```

假设系统中同时有 **libtest.a** 和 **libtest.so**。这时链接器必须从两者中选择一个进行链接。链接器会依次搜索每个文件夹（首先搜索 **–L** 选项指定的路径，然后是系统默认搜索路径）。不论链接器发现了哪一个，它都会停止搜索过程。如果当时只找到了两者中的一个，链接器会选择找到的那个进行链接。如果两个版本同时存在，除非你明确指定链接静态版本，链接器会选择共享库版本进行链接。对链接器指定 **–static** 选项表示你希望使用静态版本。例如，当使用下面的命令进行链接的时候，即使 **libtest.so** 同时存在，链接器仍将选择 **libtest.a** 进行链接：

```
% gcc –static –o app app.o –L. –ltest
```

可以用 **ldd** 命令显示与一个程序建立了动态链接的库的列表。当程序运行的时候，这些库必须存在系统中。注意 **ldd** 命令会输出一个特殊的、叫做 **ld-linux.so** 的库。它是 GNU/Linux 系统动态链接机制的组成部分。

使用 LD_LIBRARY_PATH

当你将一个程序与共享库进行动态链接的时候，链接器并不会将共享库的完整路径加入得到的执行文件中，而是只记录共享库的名字。当程序实际运行的时候，系统会搜索并加载这个共享库。默认情况下，系统只搜索 **/lib** 和 **/usr/lib**。如果某个链接到程序中的共享库被安装在这些目录之外的地方，系统将无法找到这个共享库，并因此拒绝执行你的程序。

一种解决方法是在链接的时候指明 **–Wl,-rpath** 参数。假设你用下面的命令进行链接：

```
% gcc –o app app.o –L. –ltest –Wl,-rpath,/usr/local/lib
```

当运行 **app** 的时候，系统会在 **/usr/local/lib** 中寻找所需的库文件。

另外一个解决方案是在运行程序的时候设置**LD_LIBRARY_PATH**环境变量。与**PATH**变量类似，**LD_LIBRARY_PATH**包含的是一组以冒号分割的目录列表。例如，假设我们将**LD_LIBRARY_PATH**设为 **/usr/local/lib:/opt/lib**，则系统会在搜索默认路径 **/lib**和**/usr/lib**之前搜索 **/usr/local/lib**和 **/opt/lib**目录。需要注意的是，如果在编译程序的时候设定了**LD_LIBRARY_PATH**环境变量，链接器会在搜索 **–L**参数指定的路径之前搜索这个环境变

量中指定的路径以寻找库文件。[4]

2.3.3 标准库

即使你在程序链接阶段并没有指明任何库，几乎可以确信程序总会链接到某些共享库中。这是因为 GCC 会自动将程序链接到标准 C 库 **libc**。标准 C 库的数学函数并未被包含在 **libc** 中；它们位于 **libm** 中，而这个库要求你明确指定才会链接到程序中。例如，要编译一个使用了诸如 **sin** 和 **cos** 之类的三角函数的程序 **compute.c**，你需要执行这个命令：

```
% gcc –o compute compute.c –lm
```

如果你写的是一个 C++程序，并用 **c++** 或 **g++** 命令完成链接过程，你还将自动获得对标准 C++库 **libstdc++**的链接。

2.3.4 库依赖性

经常出现这样的情况：一个库依赖另一个库。例如，许多 GNU/Linux 系统提供了 **libtiff**，一个包含了读写 TIFF 格式图片的函数的库。这个库依次依赖 **libjpeg**（JPEG 图像函数库）和 **libz**（压缩函数库）。

列表 2.9 展示了一个非常简单的程序。它通过 **libtiff** 打开一个 TIFF 格式的图片。

代码列表 2.9 （tifftest.c）使用 libtiff

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

int main ( int argc, char** argv)
{
  TIFF* tiff;
  tiff = TIFFOpen (argv[1], “r”);
  TIFFClose (tiff);
  return 0;
}
```

将这份源码保存为 **tifftest.c**。要在编译时将这个程序链接到 **libtiff** 则应在链接程序命令行中指定 **–ltiff**：

```
% gcc –o tifftest tifftest.c –ltiff
```

默认情况下，链接器会选择共享库版本的 **libtiff**。它通常位于 **/usr/lib/libtiff.so**。因为 **libtiff** 会引用 **libjpeg** 和 **libz**，这两个库的共享库版本也会被引入（共享库可以指向自己依赖的其它共享库）。可以用 **ldd** 命令验证这一点：

```
% ldd tifftest
        libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
        libc.so.6 => /lib/libc.so.6 (0x40060000)
```

[4] 在一些在线文档中可能你会看见对 **LD_RUN_PATH**环境变量的引用。不要相信它们；这个变量在 GNU/Linux系统中不起任何作用。

```
libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

而另一方面，静态库无法指向其它的库。如果你决定通过指定 **–static** 参数，将程序与静态版本的 **libtiff** 链接时，你会得到这些关于“无法解析的符号”的错误信息：

```
% gcc -static -o tifftest tifftest.c -ltiff
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function
‘TIFFjpeg_error_exit’:
tif_jpeg.o(.text+0x2a): undefined reference to ‘jpeg_abort’
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function
‘TIFFjpeg_create_compress’:
tif_jpeg.o(.text+0x8d): undefined reference to ‘jpeg_std_error’
tif_jpeg.o(.text+0xcf): undefined reference to ‘jpeg_CreateCompress’
...
```

要想将这个程序静态链接，你必须手工指定另外两个库：

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz -lm
```

有时候，两个库可能互相依赖。也就是说，第一个库可能引用了第二个库中定义的符号，反之亦然。通常这种情况都是由于不良设计导致的；但是这种情况确实可能出现。在这种情况下，你可以在命令行中多次指定同一个库。链接器会在每次读取到这个库的时候重新查找库中的符号。例如，下面的命令会导致 **libfoo.a** 被多次扫描：

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

因此，即使 **libfooo.a** 引用了 **libbar.a** 中定义的符号，且反之亦然，程序仍将被成功链接。

2.3.5 优点与缺陷

当你了解了两种类型的库的时候，你可能开始考虑实际使用哪一种。这里有一些你在选择时必须记住的注意点。

动态库的重要优点之一在于，为安装程序的系统节省了空间。假设你安装了 10 个程序，而它们同时会利用同一个库，则使用共享库较之使用静态库将为系统节省大量的空间。如果你选用静态库，则你将会在系统中随这十个程序保存十份静态库的副本。因此，使用共享库可以节省磁盘空间。而且如果你的程序是从网络上下载的，使用共享库可以同时节省下载时间。

共享库与此相关的一个优势在于，程序员可以选择升级这个库而不必强令用户同时升级所有依赖这个库的程序。例如，假设你写了一个用于处理 HTTP 连接的库。可能有许多程序依赖这个库。如果你在库的代码中发现了 bug，你可以选择升级你的库。与此同时，所有使用这个库的程序中的 bug 都会被修复；你不必像使用静态库那样重新链接所有这些程序。

这些优点也许会让你认为应该尽量使用共享库。但是，仍然存在一些现实的理由让程序选择链接到静态库。升级共享库同时会升级所有依赖程序的特点很可能成为一个缺陷。假设你开发了一个用于处理关键性任务的程序，你可能应该选择静态链接你的程序以防止对系统的升级影响到你的程序的运行。（否则，也许用户会升级系统中的共享库，由此影响到你程序的运行，然后打电话到你的技术支持热线并责怪你的程序的错误。）

如果你可能没有将库安装到 **/lib** 或 **/usr/lib** 的权限，你绝对应该重新考虑是否将你的库

作为共享库发布。（除非你要求你的用户具有管理员权限，你的库将无法被安装到 **/lib** 或 **/usr/lib** 目录。）而且，如果你不确定库最终被安装的位置，**-Wl,rpath** 的办法也无法起作用。让你的用户去设置 **LD_LIBRARY_PATH** 对他们而言意味着额外的步骤。因为每个用户都必须为自己设置这个环境变量，这将着实成为一个负担。

每当你尝试发布一个程序的时候，你都不得不对这些有缺点进行权重并选择合适的形式发布你的程序。

2.3.6 动态加载与卸载

有时，你可能希望在运行时加载一些代码，而不是将这些代码直接链接进程序。例如，设想一个支持“插件”模块的程序，如一个网页浏览器。浏览器允许第三方开发者制作插件以提供额外的功能。这些开发者制作共享库，并将它放在指定的位置。浏览器在运行的时候将自动加载这些库中的代码。

在 Linux 系统中，这种功能可以通过使用 **dlopen** 函数获取。你可以这样通过 **dlopen** 加载一个名为 **dlopen** 的函数：

```
dlopen (“libtest.so”, RTLD_LAZY);
```

（第二个参数是一个标志，它指明了绑定库中符号的方法。你可以参考 **dlopen** 的手册页以获取更详细的信息，不过 **RTLD_LAZY** 通常就是你所需要的。）如果使用动态加载函数，你需要在程序文件中包含 **<dlfcn.h>** 头文件，并将程序链接到 **libdl** 库（通过为编译器指定 **–ldl** 参数）。

这个函数会返回一个 **void *** 指针；这个指针将被用作一个操作被加载的共享库的句柄。你可以将这个指针传递给 **dlsym** 函数以获取被加载的库中特定函数的地址。假设 **libtest.so** 中定义了一个函数 **my_function**，则你可以这样调用这个函数：

```
void* handle = dlopen (“libtest.so”, RTLD_LAZY);
void (*test)() = dlsym (handle, “my_function”);
(*test)();
dlclose (handle);
```

这里，系统调用 **dlsym** 还可以用于从共享库中获取静态变量的地址。

前面提到的两个函数，**dlopen** 和 **dlsym**，均会在执行失败的时候返回 NULL。这时你可以调用 **dlerror**（不需指定任何参数）获取一个可读的信息对出现的错误进行解释。

函数 **dlclose** 可以从内存中卸载已经加载的库。技术上来说，**dlopen** 只在库并未被加载的情况下将共享库载入内存；如果这个库已被加载，则 **dlopen** 仅增加指向这个库的引用计数。同样的，**dlclose** 只是将库的引用计数减一；只有当引用计数到达 0 的时候这个函数才会真正地将库卸载。

如果你的共享库是用 C++ 语言写成，则你需要将那些用于提供外界访问的函数和变量用 **extern “C”** 链接修饰符进行修饰。假设你的共享库中有一个 C++ 函数 **foo**，而你希望通过 **dlsym** 访问这个函数，你需要这样对它进行声明：

```
extern “C” void foo ();
```

这样就可以防止 C++ 编译器对函数名称进行修饰。否则，C++ 编译器可能将函数名从 **foo** 变为另外一个看起来很可笑的名字；这个名字中包含了其它一些与这个函数相关的信息。C 编译器不会对标识符进行修饰；它会直接使用任何你指定的函数或变量名。

第三章：进程

一个程序的一份运行中的实例叫做一个进程。如果你屏幕上显示了两个终端窗口，你很可能同时将一个终端程序运行了两次——你有两个终端窗口进程。每个窗口可能都运行着一个 shell；每个运行中的 shell 都是一个单独的进程。当你从一个 shell 里面调用一个程序的时候，对应的程序在一个新进程中运行；运行结束后 shell 继续工作。

高级程序员经常在一个应用程序中同时启用多个协作的进程以使程序可以并行更多任务、使程序更健壮，或者可以直接利用已有的其它程序

本章中将要介绍的各种进程操作函数与其它 UNIX 操作系统中的进程操作函数非常相似。多数函数都在**<unistd.h>**这个包含文件中声明了原型；检查对应的手册页以确保无误。

3.1 查看进程

就算你只是坐在你的电脑前面，进程也在电脑内运行着。每个运行着的程序都会运行着一个或几个进程。让我们从观察那些正在系统中运行的进程开始。

3.1.1 进程 ID

Linux 系统中的每个进程都由一个独一无二的进程 ID（通常也被称为 *pid*）标识。进程 ID 是一个 16 位的数字，由 Linux 在创建新进程的时候自动依次分配。

每个进程都有一个父进程（除了将在 3.4.3 节“僵尸进程”中介绍的特殊的 **init** 进程）。因此，你可以把 Linux 中的进程结构想象成一个树状结构，其中 **init** 进程就是树的“根”。*父进程 ID（ppid）*就是当前进程的父进程的 ID。

当需要从 C 或 C++程序中使用进程 ID 的时候，应该始终使用**<sys/types.h>**中定义的 **pid_t** 类型。程序可以通过 **getpid()**系统调用获取自身所运行的进程的 ID，也可以通过 **getppid()**系统调用获取父进程 ID。例如下面一段程序（列表 3.1）输出了程序运行时的进程 ID 和父进程 ID。

代码列表 3.1 (print-pid.c) 输出进程 ID

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

int main ()
{
     printf ("The proces ID is %d\n", (int) getpid ());
     printf ("The parent process ID is %d\n", (int) getppid ());
     return 0;
   }
```

把这个程序运行几次并观察每次的结果，会发现每次都会输出一个不同的进程 ID，因为每次运行这个程序都建立了一个新进程。但是，如果你每次都从同一个 shell 里面调用，父进程 ID（也就是 shell 进程的 ID）并不会改变。

3.1.2 查看活动进程

运行 **ps** 命令可以显示当前系统中运行的进程。GNU/Linux 版本的 **ps** 有很多选项，因为它试图与很多不同 UNIX 版本的 **ps** 命令兼容。这些选项决定显示哪些进程以及要显示的信息。

默认情况下，调用 **ps** 会显示当前调用 **ps** 命令的终端或终端窗口所控制的所有进程的相关信息。例如：

```
% ps
  PID TTY          TIME CMD
21693 pts/8    00:00:00 bash
21694 pts/8    00:00:00 ps
```

这次调用 **ps** 显示了两个进程。第一个 **bash** 就是在这个终端上运行的 shell 程序。第二个是正在运行的 **ps** 实例自身。第一列被标记为 PID，分别显示了每个进程的 ID。

可以利用下面这个命令来仔细的研究你的 GNU/Linux 系统中运行了什么：

```
% ps -e -o pid,ppid,command
```

这里**-e** 选项让 **ps** 命令显示系统中运行的所有进程的信息。而**-o pid,ppid,command** 选项告诉 **ps** 要显示每个进程的哪些信息——这里，我们让 **ps** 显示进程 ID、父进程 ID 以及进程运行的命令行。

> ps 输出格式
>
> 当在调用 **ps** 时附加了**-o** 选项，你可以用一个逗号分割的列表指定你需要显示的进程信息。例如，`ps -o pid,user,start_time,command` 会显示进程 ID，运行该进程的用户名，进程开始的时间（*wall clock time*，墙面钟时间），以及进程运行的命令。参考 **ps** 手册页中列出的完整字段代码列表。你也可以指定**-f**（完整列表），**-l**（长列表）或**-j**（任务列表）选项以得到三种预定的列表格式。

这里是我在自己的系统上运行上面这条命令后，得到结果的开始和最后几行。你可能得到不同的结果；这取决于你系统中运行着的程序。

```
% ps -e -o pid,ppid,command
  PID  PPID COMMAND
    1     0 init [5]
    2     1 [kflushd]
    3     1 [kupdate]
...
21725 21693 xterm
21727 21725 bash
21728 21727 ps -e -o pid,ppid,command
```

注意 **ps** 命令的父进程 ID，21727，就是我调用 **ps** 命令的 shell，也就是 **bash** 的进程 ID。**Bash** 的父进程 ID 是 21725，也就是运行着 shell 的 **xterm** 程序的进程 ID。

3.1.3 中止一个进程

你可以用 **`kill`** 命令中止一个正在运行的进程。只要将需要中止的进程 ID 作为命令行

参数调用 **kill** 就可以。

kill命令通过对目标进程发送**SIGTERM**（中止）[1]信号来中止目标进程。在这个程序没有显式处理或忽略了**SIGTERM**信号的情况下，这会导致目标进程被终止。3.3 节“信号”详细介绍了信号。

3.2 创建进程

通常有两种方法可以创建进程。第一种方法相对简单，但是在使用之前应慎重考虑，因为它效率低下，而且具有不容忽视的安全风险。第二种方法相对复杂了很多，但是提供了更好的弹性、效率和安全性。

3.2.1 使用 system

C 标准库中的 **system** 函数提供了一种调用其它程序的简单方法。利用 **system** 函数调用程序结果与从 shell 中执行这个程序基本相似。事实上，**system** 建立了一个运行着标准 Bourne shell（/bin/sh）的子进程，然后将命令交由它执行。例如，列表 3.2 节的程序调用 ls 命令显示根目录的内容，正如你在 shell 中输入 `ls -l /`一样。

代码列表 3.2 （system.c）使用 system 函数

```
#include <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /");
return return_value;
  }
```

调用**system** 函数的返回值就是被调用的 shell 命令的返回值。如果shell自身无法运行，**system** 函数返回 127；如果出现了其它错误，**system** 返回 -1。

因为 **system** 函数使用 shell 调用命令，它受到系统 shell 自身的功能特性和安全缺陷的限制。你不应该试图依赖于任何特定版本的 Bourne shell。许多 UNIX 系统中，`/bin/sh` 是一个指向其它 shell 的符号链接。例如，在绝大多数 GNU/Linux 系统中，`/bin/sh` 指向 **bash**（Bourne-Again SHell），并且不同的 Linux 系统使用不同版本的 bash。例如，以 root 权限通过 **system** 调用一个程序，在不同的 Linux 系统中可能得到不同结果。因此，**fork** 和 **exec** 才是推荐用于创建进程的方法。

3.2.2 使用 fork 和 exec

DOS 和 Windows API 都包含了 **spawn** 系列函数。这些函数接收一个要运行的程序名作为参数，启动一个新进程中运行它。Linux 没有这样一个系统调用可以在一个步骤中完成这些。相应的，Linux 提供了一个 **fork** 函数，创建一个调用进程的精确拷贝。Linux 同时提供

[1] `kill` 命令还可以用于对进程发送其它的信号。`3.4` 节“进程中止”介绍了这些内容。

了另外一系列函数，被称为 **exec** 族函数，使一个进程由运行一个程序的实例转换到运行另外一个程序的实例。要产生一个新进程，应首先用 **fork** 创建一个当前进程的副本，然后使用 **exec** 将其中一个进程转为运行新的程序。

调用 *fork*

一个进程通过调用 **fork** 会创建一个被称为 *子进程* 的副本进程。父进程从调用 **fork** 的地方继续执行；子进程也一样。

那么，如何区分两个进程？首先，子进程是一个新建立的进程，因此有一个与父进程不同的进程 ID。因此可以通过调用 **getpid** 检测自身运行在子进程还是父进程。不过，**fork** 函数对父子进程提供了不同的返回值——一个进程“进入” **fork** 调用，而另外一个则从调用中“出来”。父进程得到的 **fork** 调用返回值是子进程的 ID。子进程得到的返回值是 0。因为任何进程的 ID 均不为 0，程序可以藉此很轻松的判断自身运行在哪个进程中。

列表 3.3 是一个使用 **fork** 复制进程的例子。需要注意的是，**if** 语句的第一段将仅在父进程中运行，而 **else** 部分则在子进程中运行。

代码列表 3.3 （fork.c）用 fork 复制程序进程

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

int main ()
{
  pid_t child_pid;

  printf ("the main program process ID is %d\n", (int) getpid
());

  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int)
getpid ());
    printf ("the child's process ID is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int)
getpid ());

  return 0;
}
```

调用 exec 族函数

Exec 族函数用一个程序替换当前进程中正在运行的程序。当某个 **exec** 族的函数被调用

时，如果没有出现错误的话，调用程序会被立刻中止，而新的程序则从头开始运行。

Exec 族函数在名字和作用方面有细微的差别。

- 名称包含 *p* 字母的函数（**execvp** 和 **execlp**）接受一个程序名作为参数，然后在当前的执行路径（译者注：环境变量 PATH 指明的路径）中搜索并执行这个程序；名字不包含 *p* 字母的函数在调用时必须指定程序的完整路径。
- 名称包含 *l* 字母的函数（**execl**、**execlp** 和 **execle**）接收一个字符串数组作为调用程序的参数；这个数组必须以一个 NULL 指针作为结束的标志。名字包含 *v* 字母的函数（**execv**，**execvp** 和 **execve**）以 C 语言中的 vargs（译者注：原文为 varargs，疑为笔误）形式接受参数列表。（译者注：原文中 v 和 l 的部分颠倒，疑为笔误。已参考 **execl(3)**手册页进行了更正。）
- 名称包含 *e* 字母的函数（**execve** 和 **execle**）比其它版本多接收一个指明了环境变量列表的参数。这个参数的格式应为一个以 NULL 指针作为结束标记的字符串数组。每个字符串应该表示为“变量=值”的形式。

因为 **exec** 会用新程序代替当前程序，除非出现错误，否则它不会返回。

传递给程序的参数列表和当你从 shell 运行时传递给程序的命令行参数相似。新程序可以从 main 函数的 **argc** 和 **argv** 参数中获取它们。请记住，当一个程序是从 shell 中被调用的时候，shell 程序会将第一个参数（**argv[0]**）设为程序的名称，第二个参数（**argv[1]**）为第一个命令行参数，依此类推。当你在自己的程序中使用 exec 函数的时候，也应该将程序名称作为第一个参数传递进去。

将 *fork* 和 *exec* 结合使用

运行一个子程序的最常见办法是先用 **fork** 创建现有进程的副本，然后在得到的子进程中用 **exec** 运行新程序。这样在保持原程序继续运行的同时，在子进程中开始运行新的程序。

列表 3.4 中的程序与 3.2 中的作用相似，也是用 **ls** 命令显示系统根目录下的内容。与前面例子不同的是，它直接传递**-l** 和 / 作为参数并调用了 **ls** 命令，而不是通过运行一个 shell 再调用命令来完成这一操作。

代码列表 3.4 （fork-exec.c）将 fork 和 exec 结合使用

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

/* 产生一个新进程运行新的程序。PAORGAM 是要运行的程序的名字；系统会在
执行路径中搜索这个程序运行。ARG_LIST 是一个以 NULL 指针结束的字符串列表，
用作程序的参数列表。返回新进程的 ID。 */

int spawn (char* program, char** arg_list)
{
  pid_t child_pid;

  /* 复制当前进程。*/
```

```
  child_pid = fork ();
  if (child_pid != 0)
    /* 这里是父进程。*/
    return child_pid;
  else {
    /* 现在从执行路径中查找并执行 PROGRAM。*/
    execvp (program, arg_list);
    /* execvp 函数仅当出现错误的时候才返回。*/
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}

int main ()
{
  /* 准备传递给 ls 命令的参数列表 */
  char* arg_list[] = {
    "ls",     /* argv[0], 程序的名称 */
    "-l",
    "/",
    NULL      /* 参数列表必须以 NULL 指针结束 */
  };

  /* 建立一个新进程运行 ls 命令。忽略返回的进程 ID */
  spawn ("ls", arg_list);

  printf ("done with main program\n");

  return 0;
}
```

3.2.3 进程调度

Linux 会分别独立地调度父子进程；不保证进程被调度的先后顺序，也不保证被调度的进程在被另外一个（或系统中其它进程）打断之前会运行多久。具体来说，**ls**命令也许在父进程结束之前根本没有被调度运行，也可能是**ls**命令运行了一部分或者全部完成之后主进程才结束执行[2]。Linux保证每个程序都会得到运行——不会有某个进程因为缺乏资源而无法运行。

你可以将一个进程标记为次要的；给进程指定一个较高的 *niceness* 值会给这个进程分配较低的优先级。默认情况下，每个进程的 *niceness* 均为 0。较高的 *niceness* 值代表了较低的进程优先级；相应的，较低的 *niceness* 值（负值）表示较高的进程优先级。

[2] 3.4.1 节（“等待进程终止”）展示了一种序列化两个进程顺序的方法。

使用 **nice** 命令的**-n** 参数允许用户以一个指定的 *niceness* 值运行特定程序。例如，要执行“`sort input.txt > output.txt`”这个会执行较长时间的排序命令，可以通过下面的命令降低它的优先级，使它不会过度地影响系统的运行：

```
% nice -n 10 sort input.txt > output.txt
```

你可以使用 **renice** 命令从命令行调整一个正在运行的程序的 *niceness* 值。

可以通过调用 **nice** 函数以编程的方法调整一个运行中的进程的优先级。它的参数会被加在调用进程的 *niceness* 值上。记住，大于 0 的值会提高进程的 *niceness* 值，从而降低进程优先级。

需要注意的是，只有当一个进程以 root 权限运行的时候才能以负的 *niceness* 值运行其它程序，或降低一个进程的 *niceness* 值。这表明，只有当你以 root 身份登陆的时候，你才可以用负数做参数调用 **nice** 和 **renice** 命令，而只有当一个进程以 `root` 身份运行的时候才可以以负值作参数调用 **nice** 函数。这种限制可以防止普通用户从其它用户的手中夺取程序运行优先级。

3.3 信号

*信号（Signal）*是 Linux 系统中用于进程之间相互通信或操作的一种机制。信号是一个相当广泛的课题；在这里，我们仅仅探讨几种最重要的信号以及利用信号控制进程的技术。

信号是一个发送到进程的特殊信息。信号机制是异步的；当一个进程接收到一个信号时，它会立刻处理这个信号，而不会等待当前函数甚至当前一行代码结束运行。信号有几十种，分别代表着不同的意义。信号之间依靠它们的值来区分，但是通常在程序中使用信号的名字来表示一个信号。在 Linux 系统中，这些信号和以它们的名称命名的常量均定义在 **`/usr/include/bits/signum.h`** 文件中。（通常程序中不需要直接包含这个头文件，而应该包含**<signal.h>**。）

当一个进程接收到信号，基于不同的*处理方式（disposition）*，该进程可能执行几种不同操作中的一种。每个信号都有一个*默认处理方式(default disposition)*，当进程没有指定自己对于某个信号的处理方式的时候，默认处理方式将被用于对对应信号作出响应。对于多数种类的信号，程序都可以自由指定一个处理方式——程序可以选择忽略这个信号，或者调用一个特定的信号处理函数。如果指定了一个信号处理函数，当前程序会暂停当前的执行过程，同时开始执行信号处理函数，并且当信号处理函数返回之后再从被暂停处继续执行。

Linux 系统在运行中出现特殊状况的时候也会向进程发送信号通知。例如，当一个进程执行非法操作的时候可能会收到 **SIGBUS**（主线错误），**SIGSEGV**（段溢出错误）及 **SIGFPE**（浮点异常）这些信号。这些信号的默认处理方式都是终止程序并且产生一个核心转储文件（core file）。

一个进程除了响应系统发来的信号，还可以向其它进程发送信号。对于这种机制的一个最常见的应用就是通过发送**SIGTERM**或**SIGKILL**信号来结束其它进程。[3]#3 除此之外，它还常见于向运行中的进程发送命令。两个“用户自定义”的信号**SIGUSR1** 和**SIGUSR2** 就是专门作此用途的。**SIGHUP**信号有时也用于这个目的——通常用于唤醒一个处于等待状态的进程或者使进程重新读取配置文件。

系统调用 **sigaction** 用于指定信号的处理方式。函数的第一个参数是信号的值。之后两个参数是两个指向 **sigaction** 结构的指针；第一个指向了将被设置的处理方式，第二个用于

[3] 有什么区别？`SIGTERM` 信号要求进程中止；进程可以修改请求或者直接忽略这个信号。`SIGKILL` 信号会立即中止进程，因为进程无法忽略或修改对 `SIGKILL` 信号的响应。

保存先前的处理方式。这两个 **sigaction** 结构中最重要的都是 **sa_handler** 域。它可以是下面三个值：

- SIG_DFL，指定默认的信号处理方式
- SIG_IGN，指定该信号将被忽略
- 一个指向信号处理函数的指针。这个函数应该接受信号值作为唯一参数，且没有返回值。

因为信号处理是异步进行的，当信号处理函数被调用的时候，主程序可能处在非常脆弱的状态，并且这个状态会一直保持到信号处理函数结束。因此，应该尽量避免在信号处理函数中使用输入输出功能、绝大多数库函数和系统调用。

信号处理函数应该做尽可能少的工作以响应信号的到达，然后返回到主程序中继续运行（或者结束进程）。多数情况下，所进行的工作只是记录信号的到达。而主程序则定期检查是否有信号到达，并且针对当时情况作出相应的处理。

信号处理函数也可能被其它信号的到达所打断。虽然这种情况听起来非常罕见，一旦出现，程序将非常难以确定问题并进行调试。（这是竞争状态的一个例子。在第四章“线程”第 4.4 节“同步及临界段”中进行了讨论。）因此，对于你的信号处理函数进行哪些工作一定要进行慎重的考虑。

甚至于对全局变量赋值可能也是不安全的，因为一个赋值操作可能由两个或更多机器指令完成，而在这些指令执行期间可能会有第二个信号到达，致使被修改的全局变量处于不完整的状态。如果你需要从信号处理函数中设置全局标志以记录信号的到达，这个标志必须是特殊类型 **sig_atomic_t** 的实例。Linux 保证对于这个类型变量的赋值操作只需要一条机器指令，因此不用担心可能在中途被打断。在 Linux 系统中，**sig_atomic_t** 就是基本的 **int** 类型；事实上，对 int 或者更小的整型变量以及指针赋值的操作都是原子操作。不过，如果你希望所写的程序可以向任何标准 UNIX 系统移植，则应将所有全局变量设为 **sig_atomic_t** 类型。

如下所示，代码列表 3.5 中的简单程序中，我们利用信号处理函数统计程序在运行期接收到 **SIGUSR1** 信号的次数。**SIGUSR1** 信号是一个为应用程序保留的信号。

代码列表 3.5 （sigusr1.c）使用信号处理函数

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handle (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
```

```
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* 这里可以执行一些长时间的工作。*/
  /* ... */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

3.4 进程终止

通常，进程会以两种情况的之一结束：调用 **exit** 函数退出或从 **main** 函数返回。每个进程都有退出值（*exit code*）：一个返回给父进程的数字。一个进程退出值就是程序调用 **exit** 函数的参数，或者 **main** 函数的返回值。

进程也可能由于信号的出现而异常结束。例如，之前提到的 **SIGBUS**，**SIGSEGV** 和 **SIGFPE** 信号的出现会导致进程结束。其它信号也可能显式结束进程。当用户在终端按下 Ctrl+C 时会发送一个 **SIGINT** 信号给进程。**SIGTERM** 信号由 **kill** 命令发送。这两个信号的默认处理方式都是结束进程。进程通过调用 **abort** 函数给自己发送一个 **SIGABRT** 信号，导致自身中止运行并且产生一个 core file。最强有力的终止信号是 **SIGKILL**，它会导致进程立刻终止，而且这个信号无法被阻止或被程序自主处理。

这里任何一个信号都可以通过指定一个特殊选项，由 **kill** 命令发送；例如，要通过发送 **SIGKILL** 中止一个出问题的进程，只需要执行下面的命令（这里 **pid** 是目标进程号）：

```
% kill -KILL pid
```

要从程序中发送信号，使用 **kill** 函数。第一个参数是目标进程号。第二个参数是要发送的信号；传递 **SIGTERM** 可以模拟 **kill** 命令的默认行为。例如，你可以利用 **kill** 函数，像这样从父进程中结束子进程的运行（这里 **child_pid** 包含的是子进程的进程号）：

```
% kill (child_pid, SIGTERM);
```

需要包含**<sys/types.h>**和**<signal.h>**头文件才能在程序中调用 **kill** 函数。

根据习惯，程序的退出代码可用来确认程序是否正常运行。返回值为 0 表示程序正确运行，而非零的返回值表示运行过程出现错误。在后一种情况下，返回值可能表示了特定的错误含义。通常应该遵守这个约定，因为 GNU/Linux 系统的其它组件会假设程序遵循这个行为模式。例如，当使用 &&（逻辑与）或 ||（逻辑或）连接多个程序的时候，shell 根据这个假定判断逻辑运算的结果。因此，除非有错误发生，你都应该在 **main** 结束的时候明确地返回 0。

对于多数 shell 程序，最后运行的程序的返回值都保存在特殊环境变量**$?**中。在下面这个例子中 **ls** 命令被执行了两次，并且每次运行之后会输出返回值。第一次，**ls** 运行正常并且返回 0。第二次，**ls** 运行出现一个错误（因为命令行参数指定的文件不存在）且返回了一个非零值。

```
% ls
```

```
bin   coda  etc   lib        misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

需要注意的是，尽管 **exit** 函数的参数类型是 **int**，而 **main** 函数返回值也是 **int** 类型，Linux 不会为返回值保留 32 位长度。实际上，你应该只使用 0 到 127 之间的数值作为退出代码。大于 128 的退出代码有特殊的含义——当一个进程由于一个信号而结束运行，它的退出值就是 128 加上信号的值。

3.4.1 等待进程结束

如果你输入并且运行了代码列表 3.4 的 **fork** 和 **exec** 示例程序，你可能已经发现，**ls** 程序的输入很多时候出现在“主程序”结束之后。这是因为子进程，也就是运行 **ls** 命令的进程，是相独立于主进程被调度的。因为 Linux 是一个多任务操作系统，两个进程看起来是并行执行的，而且你无法猜测 **ls** 程序会在主程序运行之前还是之后获取运行的机会。

不过，在某些情况下，主程序可能希望暂停运行以等待子进程完成任务。可以通过 **wait** 族系统调用实现这一功能。这些函数允许你等待一个进程结束运行，并且允许父进程得到子进程结束的信息。**Wait** 族系统调用一共有四个函数；通过选择不同的版本，你可以选择从退出进程得到信息的多少，也可以选择关注某个特定子进程的退出。

3.4.2 wait 系统调用

这一族函数中，最简单的是 **wait**。它会阻塞调用进程，直到某一个子进程退出（或者出现一个错误）。它通过一个传入的整型指针参数返回一个状态码，从而可以得到子进程的退出信息。例如，**WEXITSTATUS** 宏可以提取子进程的退出值。

你可以用 **WIFEXITED** 宏从一个子进程的返回状态中检测该进程是正常结束（利用 **exit** 函数或者从 **main** 函数返回）还是被没有处理的信号异常终止。对于后一种情况，可以用 **WTERMSIG** 宏从中得到结束该进程的信号。

这里还是 **fork** 和 **exec** 示例代码中的 **main** 函数。这一次，父进程通过调用 **wait** 等待子进程（也就是运行 **ls** 命令的进程）退出。

```
int main ()
{
  int child_status;

  /* 传递给 ls 命令的参数列表 */
  char* arg_list[] = {
    "ls",  /* argv[0], 程序的名称 */
    "-l",
    "/",
    NULL   /* 参数列表必须以 NULL 结束 */
```

```
        };

        /* 产生一个子进程运行 ls 命令。忽略返回的子进程 ID。*/
        spawn ("ls", arg_list);

        /* 等待子进程结束。*/
        wait (&child_status);
        if (WIFEXITED (child_status))
          printf ("the child proces exited normally, with exit code
%d\n",
                  WEXITSTATUS (child_status));
        else
          printf ("the child process exited abnormally\n");

        return 0;
      }
```

Linux 还提供了一些相似的系统调用；其中一些更具弹性，而一些提供了与退出的子进程相关的更多的信息。**Wait3** 函数可以获取退出进程的 CPU 占用情况，而 **wait4** 函数允许你通过更多参数指定等待的进程。

3.4.3 僵尸进程

如果一个子进程结束的时候，它的父进程正在调用 **wait** 函数，子进程会直接消失，而退出代码则通过 **wait** 函数传递给父进程。但是，如果子进程结束的时候，父进程并没有调用 **wait**，则又会发生什么？它是不是简单地就消失了呢？不，因为如果这样，它退出时返回的相关信息——譬如它是否正常结束，以及它的退出值——会直接丢失掉。在这种情况下，子进程死亡的时候会转化为一个僵尸进程。

一个僵尸进程是一个已经中止而没有被清理的进程。清理僵尸子进程是父进程的责任。**Wait** 函数会负责这个清理过程，所以你不必在等待一个子进程之前检测它是否正在运行。假设，一个进程创建了一个子进程，进行了另外一些计算，然后调用了 **wait**。如果子进程还没有结束，这个进程会在 **wait** 调用中阻塞，直到子进程结束。如果子进程在父进程调用 **wait** 之前结束，子进程会变成一个僵尸进程。当父进程调用 **wait**，僵尸子进程的结束状态被提取出来，子进程被删除，并且 **wait** 函数立刻返回。

如果父进程不清理子进程会如何？它们会作为僵尸进程，一直被保留在系统中。代码列表 3.6 里的程序在产生一个立刻结束的子进程之后然后休眠一分钟退出而不清理子进程。

代码列表 3.7 （zombie.c）制作一个僵尸进程

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

int main ()
{
```

```
  pid_t child_pid;

  /* 创建一个子进程 */
  child_pid = fork ();
  if (child_pid > 0) {
    /*这是父进程。休眠一分钟。 */
    sleep (60);
  }
  else {
    /*这是子进程。立刻退出。 */
    exit (0);
  }
  return 0;
}
```

试着把这个程序编译成一个名为 **make-zombie** 的程序。运行这个程序；在它还在运行的同时，用下列命令在另外一个窗口列出系统中的进程：

```
% ps -e -o pid,ppid,stat,cmd
```

该命令会列出进程 ID、父进程 ID、进程状态和进程命令行。观察结果，发现除了父进程 **make-zombie** 之外，还有一个 **make-zombie** 出现在列表中。这是子进程；注意它的父进程 ID 就是第一个 **make-zombie** 进程的 ID。子进程被标记为**<defunct>**而且它的状态代码为 Z，表示僵尸（Zombie）。

如果 **make-zombie** 进程退出而没有调用 **wait** 会出现什么情况？僵尸进程会停留在系统中吗？不——试着再次运行 **ps**，你会发现两个 **make-zombie** 进程都消失了。当一个程序退出，它的子进程被一个特殊进程继承，这就是 **init** 进程。**Init** 进程总以进程 ID 1 运行（它是 Linux 启动后运行的第一个进程）。**Init** 进程会自动清理所有它继承的僵尸进程。

3.4.4 异步清理子进程

如果你创建一个子进程只是简单的调用 **exec** 运行其它程序，在父进程中立刻调用 **wait** 进行等待并没有什么问题，只是会导致父进程阻塞等待子进程结束。但是，很多时候你希望在子进程运行的同时，父进程继续并行运行。怎么才能保证能清理已经结束运行的子进程而不留下任何僵尸进程在系统中浪费资源呢？

一种解决方法是让父进程定期调用 **wait3** 或 **wait4** 以清理僵尸子进程。在这种情况调用 **wait** 并不合适，因为如果没有子进程结束，这个调用会阻塞直到子进程结束为止。然而，你可以传递 **WNOHANG** 标志给 **wait3** 或 **wait4** 函数作为一个额外的参数。如果设定了这个标志，这两个函数将会以非阻塞模式运行——如果有结束的子进程，它们会进行清理；否则会立刻返回。第一种情况下返回值是结束的子进程 ID，否则返回 0。

另外一种更漂亮的解决方法是当一个子进程结束的时候通知父进程。有很多途径可以做到这一点；在第五章“进程间通信”介绍了这些方法，不过幸运的是 Linux 利用信号机制替你完成了这些。当一个子进程结束的时候，Linux 给父进程发送 **SIGCHLD** 信号。这个信号的默认处理方式是什么都不做；这也许是为什么之前你忽略了它的原因。

因此，一个简单的清理结束运行的子进程的方法是响应 **SIGCHLD** 信号。当然，当清理子进程的时候，如果需要相关信息，一个很重要的工作就是保存进程退出状态，因为一旦用 **wait** 清理了进程，就再也无法得到这些信息了。列表 3.7 中就是一个利用 **SIGCHLD** 信号处理函数清理子进程的程序代码。

代码列表 3.7 （sigchld.c）利用 SIGCHLD 处理函数清理子进程

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void clean_up_child_process (int signal_number)
{
  /* 清理子进程。*/
  int status;
  wait (&status);
  /* 在全局变量中存储子进程的退出代码。*/
  child_exit_status = status;
}

int main ()
{
  /* 用 clean_up_child_process 函数处理 SIGCHLD。*/
  struct sigaction sigcihld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigcihld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);

  /* 现在进行其它工作，包括创建一个子进程。*/
  /* ...  */

  return 0;
}
```

注意信号处理函数中将进程退出代码保存到了全局变量中，从而可以在主程序中访问它。因为这个变量在信号处理函数中被赋值，我们将它声明为 **sig_atomic_t** 类型。

第四章：线程

线程，不同于进程，是一种允许一个程序同时执行不止一个任务的机制。于进程相似，不同线程看起来是并行运行的；Linux 核心对它们进行异步调度，不断中断它们的执行以给其它线程执行的机会。

概念上，线程出现在进程中。相比进程，线程是一种更细粒度的执行单元。当你调用一个程序，Linux 创建一个新进程，并且在那个新进程中创建一个线程；这个线程依序执行程序。这个线程可以创建更多的线程；所有这些线程在同一个进程中执行同一个程序，但是每个线程在特定时间点上可能分别执行这个程序的不同部分。

我们已经看到一个进程如何创建新进程。子进程开始时候运行父进程的程序，并且从父进程处复制了虚拟内存、文件描述符和其它信息。子进程可以修改自己的内存、关闭文件描述符、执行其它各种操作，但是这些操作不会影响父进程；反之亦然。不过，当一个程序创建了一个线程时并不会复制任何东西。创建和被创建的线程同先前一样共享内存空间、文件描述符和其它各种系统资源。例如，当一个线程修改了一个变量的值，随后其它线程就会看到这个修改过的值。相似的，如果一个线程关闭了一个文件描述符，其它线程也无法从这个文件描述符进行读或写操作。因为一个进程中所有线程只能执行同一个程序，如果任何一个线程调用了一个 exec 函数，所有其它线程就此终止（当然，新的程序也可以创建线程）。

GNU/Linux 实现了 POSIX 标准线程 API（所谓 *pthreads*）。所有线程函数和数据类型都在 **<pthread.h>** 头文件中声明。这些线程相关的函数没有被包含在 C 标准库中，而是在 **libpthread** 中，所以当链接程序的时候需在命令行中加入 **-lpthread** 以确保能正确链接。

4.1 创建线程

进程中的每个线程都以*线程 ID* 标识。在 C 或 C++ 程序中，线程 ID 被表示为 **pthread_t** 类型的值。

创建线程时，每个线程都开始执行一个*线程函数*。这只是一个普通的函数，包含了线程应执行的代码；当函数返回的时候，线程也随之结束。在 GNU/Linux 系统中，线程函数接受一个 **void*** 类型的参数，并且返回 **void*** 类型。这个参数（parameter）被称为*线程参数（thread argument）*：GNU/Linux 系统不经查看直接将它传递给线程。你的程序可以利用这个参数给新线程传递数据。相似的，你的线程可以利用返回值给它的创建者线程返回数据。

函数 **pthread_create** 负责创建新线程。你需要给它提供如下信息：

1、一个指向 **pthread_t** 类型变量的指针；新线程的线程 ID 将存储在这里。

2、一个指向*线程属性（thread attribute）*对象的指针。这个对象控制着新线程与程序其它部分交互的具体细节。如果传递 *NULL* 作为线程属性，新线程将被赋予一组默认线程属性。线程属性在 4.1.5 节“线程属性”中有更多讨论。

3、一个指向线程函数的指针。这是一个普通的函数指针，类型如下：

```
void* (*) (void*)
```

4、一个线程参数，类型 **void***。不论你传递什么值作为这个参数，当线程开始执行的时候，它都会被直接传递给新的线程。

函数 **pthread_create** 会在调用后立刻返回，原线程会继续执行之后的指令。同时，新线程开始执行线程函数。Linux 异步调度这两个线程，因此你的程序不能依赖两个线程得到执行的特定先后顺序。

列表 4.1 中的程序创建一个不断输出 x 到标准错误输出的线程。在执行 **pthread_create** 之后，主线程不断输出 o 到标准错误输出。

列表 4.1 (thread-create.c) 创建线程

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

/* 打印 x 到错误输出。没有使用参数。不返回数据。*/

void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* 主程序 */

int main ()
{
  pthread_t thread_id;
  /* 传教新线程。新线程将执行 print_xs 函数。*/
  pthread_create (&thread_id, NULL, *print_xs, NULL);
  /* 不断输出 o 到标准错误输出。*/
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

使用以下命令编译链接这个程序：

```
% cc -o thread-create thread-create.c -lpthread
```

试着执行看看会发生什么。注意这个没有规律的 x 和 o 的交替输出，这表示 Linux 不断调度两个线程。

在一般状况下，一个线程有两种退出方式。一种方式，如先前所示，是从线程函数中返回以退出线程。线程函数的返回值也被作为线程的返回值。另一种方式则是线程显式调用 **pthread_exit**。这个函数可以直接在线程函数中调用，也可以在其它直接、间接被线程函数调用的函数中调用。调用 **pthread_exit** 的参数就是线程的返回值。

4.1.1 给线程传递数据

线程参数提供了一种为新创建的线程传递数据的简便方式。因为参数是 **void***，你无法

通过参数本身直接传递大量数据，而应使用线程参数传递一个指向某个数据结构或数组的指针。一个常用的技巧是给线程函数定义一个结构以包含线程函数所期待的实际参数序列。

利用线程参数可以很轻易地重用一个线程函数创建许多线程。所有这些线程可以针对不同的数据执行相同的操作。

列表 4.2 中的程序与前一个例子非常相似。这个程序会创建两个新线程，一个输出 x 而另一个输出 o。不同于之前的不停输出，每个线程输出固定的字符数之后就从线程函数中返回以退出线程。同一个函数 **char_print** 在两个线程中均被执行，但是程序为每个线程指定不同的 **struct char_print-parms** 实例作为参数。

代码列表 4.2 (thread-create2) 创建两个线程

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

/* print_function 的参数 */

struct char_print_parms
{
  /* 用于输出的字符 */
  char character;
  /* 输出的次数 */
  int count;
};

/* 按照 PARAMETERS 提供的数据，输出一定数量的字符到 stderr。
    PARAMETERS 是一个指向 struct char_print_parms 的指针 */

void* char_print (void* parameters)
{
  /* 将参数指针转换为正确的类型 */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}

/* 主程序 */

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
```

```
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_ars;

  /* 创建一个线程输出 30000 个 x */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* 创建一个线程输出 20000 个 o */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  return 0;
}
```

不过，且慢！列表 4.2 中的程序有一个严重的错误。主线程（就是执行 main 函数的线程）将线程参数结构（thread1_args 和 thread2_args）创建为局部变量，然后将指向它们的指针传递给创建的线程。如何防止 Linux 调度这三个线程，使 **main** 在另外两个线程结束之前结束？没有办法！一旦这个情况发生，包含线程参数结构的内存将在被两个线程访问的同时被释放。

4.2.2 等待线程 (原文：Joining Threads)

一个解决办法是强迫 **main** 函数等待另外两个线程的结束。我们需要一个类似 **wait** 的函数，但是等待的是线程而不是进程。这个函数是 pthread_join。它接受两个参数：线程 ID，和一个指向 **void***类型变量的指针，用于接收线程的返回值。如果你对线程的返回值不感兴趣，则将 NULL 作为第二个参数。

前面提到，列表 4.2 中的程序有错误；而现在列表 4.3 展示了正确的版本。在这个版本中，**main** 在输出 x 和 o 的两个线程完成——因此不会再引用参数——之后才会退出。

列表 4.3 主函数修订版 thread-create2.c

```
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* 创建一个输出 30000 个 x 的线程 */
  hread1_args.character = 'x';
  thread1_args.count = 30000;
  prhread_create (&thread1_id, NULL, &char_print, &thread1_args);
```

```
  /* 创建一个输出 20000 个 o 的线程 */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /* 确保第一个线程结束 */
  pthread_join (thread1_id, NULL);
  /* 确保第二个线程结束 */
  pthread_join (thread2_id, NULL);

  /* 现在我们可以安全地返回 */
  return 0;
}
```

这个故事的教训：一旦你将对某个数据变量的引用传递给某个线程，务必确保这个变量在不会被释放（甚至在其它线程中也不行!），直到你确定这个线程不会再使用它。这对于局部变量（当生命期结束的时候自动释放）和堆上分配的对象（通过 **free** 或者 C++的 **delete** 手工释放）同样适用。

4.1.3 线程返回值

如果传递给**pthread_join**的第二个参数不是 NULL，则线程返回值会被存储在这个指针指向的内存空间中。线程返回值，与线程变量一样，也是**void***类型。如果你想要返回一个**int**或者其它小数字，你可以简单地把这个数值强制转换成**void***指针并返回，并且在调用**pthread_join**之后把得到的结果转换回相应的类型[1]。

列表 4.4 中的程序在一个单独线程中计算第 n 个质数。这个线程会将得到的质数作为返回值传回主线程。与此同时，主线程可以执行其它的代码。注意，**compute_prime** 函数中使用的连续进行除法的算法是非常低效的；如果你需要在你的程序中计算很多质数，请参考有关数值算法的书。

列表 4.4 (primes.c) 在线程中计算质数

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

/*（非常低效地）计算连续的质数。返回第 N 个质数。N 是由 *ARG 指向的参数。*/

void* compute_prime (void* arg)
{
  int candidate = 2;
  int n = *((int*) arg);
```

[1] 注意，这样的代码是不可移植的，并且你必须自己保证所传递的数据类型在与 void* 之间来回转换不会导致位的丢失。

```
  while (1) {
    int factor;
    int is_prime = 1;

    /*用连续除法检测是否为质数。*/
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /*这个质数是我们寻找的么？ */
    if (is_prime) {
      if (--n == 0)
        /*将所求的质数作为线程返回值传回。*/
        return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}

int main ()
{
  pthread_t thread;
  int which_prime = 5000;
  int prime;

  /*开始计算线程，求取第 5000 个质数。*/
  pthread_create (&thread, NULL, &compute_prime, &which_prime);
  /*在这里做其它的工作……*/
  /*等待计算线程的结束，并且取得结果。*/
  pthread_join (thread, (void*) &prime);
  /*输出所求得的最大质数。*/
  printf("The %dth prime number is %d.\n", which_prime, prime);
  return 0;
}
```

4.1.4 关于线程 ID 的更多信息

有时候，一段代码需要确定是哪个线程正在执行到这里。可以通过 **pthread_self** 函数获取调用线程 ID。所得到的线程 ID 可以用 **pthread_equal** 函数与其它线程 ID 进行比较。

这些函数可以用于检测当前线程 ID 是否为一特定线程 ID。例如，一个线程利用 **pthread_join** 等待自身是错误的。（在这种情况下，**pthread_join** 会返回错误码 **EDEADLK**。）

要事前发现这个情况，可以用这样的代码进行判断：

```
if (!pthread_equal (pthread_self (), other_thread))
  pthread_join (other_thread, NULL);
```

4.1.5 线程属性

线程属性提供了一种可以用于在细粒度调整线程行为方式的机制。试着回忆一下，**pthread_create** 函数接受一个指向线程属性对象的指针。如果你传递 NULL 指针，默认线程属性被用于配置新线程。同时，你也可以通过创建并且传递一个线程属性对象来指明属性中的一些值。

要指明自定义的线程属性，你必须参照以下步骤：

1. 创建一个 **pthread_attr_t** 对象。最简单的方法是声明一个该类型的自动变量。
2. 调用 **pthread_attr_init**，传递一个指向新创建对象的指针。这个步骤将各个属性置为默认值。
3. 修改这个对象，使各个属性包含期望的值。
4. 在调用 **pthread_create** 的时候，传递一个指向该对象的指针。
5. 调用 **pthread_attr_destroy** 释放这个属性对象。这个 **pthread_attr_t** 对象本身不会被释放；可以通过 **pthread_attr_init** 将其重新初始化。

一个单独线程对象可以用于创建许多线程。在创建线程之后没有必要保持线程属性对象。

对于多数 GNU/Linux 应用程序而言，一般只有一个线程属性会显得有趣（其它的主要属性均针对实时程序）。这个属性就是线程的*脱离状态（detach state）*。一个线程可以创建为一个*可等待线程（joinable thread）*（默认情况）或者一个*脱离线程（detached thread）*。一个可等待线程，类似一个进程，在结束的时候不会被 GNU/Linux 系统自动清理。相反的，它的退出状态停留在系统中（某种程度来说，类似一个僵尸进程）直到另外某个线程调用 **pthread_join** 获取它的返回值。直到这时，它的资源才被释放。与此不同的是，一个脱离线程在结束的时候会被自动清理。因为脱离线程会被立刻清理，其它线程无法与它的结束事件进行同步，也无法获取其返回值。

可以使用 **pthread_attr_setdetachstate** 函数设置脱离属性。第一个参数是一个指向线程属性对象的指针，第二个参数是脱离状态。因为可等待状态是默认的，只有创建脱离线程的时候才需要调用这个函数；传递 **PTHREAD__CREATE__DETACHED** 作为第二个参数。

列表 4.5 中的代码通过修改线程属性创建一个脱离线程。

列表 4.5 (detached.c) 创建脱离线程的原型程序

```
#include <pthread.h>

void* thread_function (void* thread_arg)
{
  /* 这里完成工作……*/
}
```

```
int main ()
{
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);

  /* 进行其它工作……*/

  /* 不需要等待第二个线程 */
  return 0;
}
```

即使一个线程是创建成为一个可等待线程，它也可以随后转换成一个脱离线程。调用 **pthread_detach** 进行这个转换过程。一旦线程成为脱离线程，它将无法转换会可等待状态。

4.2 线程取消

一般情况下，一个线程在它正常结束（通过从线程函数返回或者调用 **pthread_exit** 退出）的时候终止。但是，一个线程可以请求另外一个线程中止。这被成为取消一个线程。

要取消一个线程，以被取消的线程 ID 作为参数调用 **pthread_cancel**。一个被取消的线程可以稍后被其它线程等待；实际上，你应该对一个被取消的线程执行 **pthread_wait** 以释放它占用的资源，除非这个线程是脱离线程（参考 4.1.5 节，“线程属性”）。一个取消线程的返回值由特殊值 **PTHREAD_CANCELED** 指定。

经常，线程可能运行在一段不可分割的代码中，必须全部得到执行或者干脆不执行。例如，线程可能分配一些资源，使用并稍后释放它们。如果线程在中途被取消，它可能没有机会释放那些被分配的资源，从而导致资源的泄漏。为防止这种情况发生，一个线程可以控制自身是否可以被取消，以及何时允许取消操作。

对于线程取消而言，一个线程可能处于如下三种情况之一：

- 线程可以被*异步取消（asynchronously cancelable）*。线程可以在执行中的任意时刻被取消。
- 线程可以被*同步取消（synchronously cancelable）*。线程可以被取消，但是不是在任意时刻都可以。相反的，取消请求会被排队，而线程只有在到达特殊的执行点才会执行取消操作。
- 线程*不可被取消（uncancelable）*。尝试取消线程的请求会被直接忽略。

当一个线程刚被建立的时候，它处于可同步取消状态。

4.2.1 同步和异步线程

一个可异步取消的线程可在它执行过程中的任意时刻被取消。一个可同步取消的线程，

概念上来说，只能在执行过程中的特定位置被取消。这些位置被称为*取消点（cancellation points）*。线程会将取消请求排队，直到到达下一个取消点再由程序处理。

可以通过调用 **pthread_setcanceltype** 使一个线程进入允许被异步取消的状态。这个函数作用于调用它的线程。第一个参数可以是常量 **PTHREAD_CANCEL_ASYNCHRONOUS**，表示将线程设置为可异步取消状态；或者是 **PTHREAD_CANCEL_DEFERRED**，将线程设置回可同步取消状态。如果第二个参数不为空，则它指向的变量将保存线程的前一个取消类型。下面例子中的代码将线程设置为异步取消状态：

```
pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

取消点究竟是什么？它们应该被放置在哪里？最直接的创建一个取消点的方法是调用 **pthread_testcancel**。这个函数的唯一工作就是在一个可同步取消的线程中处理一个没有被处理的线程取消请求。如果一个线程要执行长时间的计算过程，则应该定期在线程取消不会导致资源泄露或产生其它负面影响的情况下调用 **pthread_testcancel**。

还有部分函数可以作为隐式的取消点。在 **pthread_cancel** 的手册页中有这些函数的列表。需要注意的是，其它函数可能因为调用这些函数而间接成为取消点。

4.2.2 不可取消的临界区

线程可以利用 **pthread_setcancelstate** 函数完全禁止自己被取消。类似于 **pthread_setcanceltype**，这个函数作用于调用线程。如果将 **PTHREAD_CANCEL_DISABLE** 作为第一个参数，则线程取消操作将被禁止；如果是 **PTHREAD_CANCEL_ENABLE** 则线程取消被重新允许。第二个参数如果不为空，则指向的变量将保存线程的前一个线程取消状态。下面例子中的代码将禁止线程取消：

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL);
```

程序可以利用 **pthread_setcancelstate** 实现临界区。临界区指的是一段必须完整执行或者完全不执行的代码；换言之，一旦一个线程进入临界区，在到达临界区终点之前它将无法被取消。

举个例子，假设你正在实现一个银行程序中负责在帐户之间转移款项的一部分。要实现这一点，你必须在给一个帐户的余额中加上一个值的同时从另外一个帐户中扣除相同的值。如果运行这一段过程的线程很不巧地在这两个操作之间被取消，事务的中断会错误地导致银行的总储蓄额提高。要防止这种情况的发生，应将这两个操作放在一个临界区中。

你可以实现将整个传输过程封装在一个 **process_transaction** 函数中，如列表 4.6 中这样。这个函数禁用线程取消以进入一个临界区，然后才操作帐户。

列表 4.6 (critical-section.c) 用临界区保护银行事务

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

/* 表示帐户结余的数组，以帐户号码为序列 */

float* account_balances;
```

```
/* 将 DOLLARS 从 FROM_ACCT 帐户转移到 TO_ACCT。如果成功，
   返回 0；如果 FROM_ACCT 的结余过低则返回 1。 */

int process_transaction (int from_acct, int to_acct, float dollars)
{
  int old_cancel_state;

  /* 检查 FROM_ACCT 的结余。 */
  if (account_balences[from_acct] < dollars)
    return 1;

  /* 进入临界区 */
  pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
  /* 进行余额的转移 */
  account_balances[to_acct] += dollars;
  account_balances[from_acct] -= dollars;
  /* 退出临界区 */
  pthread_setcancelstate (old_cancel_state, NULL);

  return 0;
}
```

注意在退出临界区的时候应恢复线程取消状态到进入临界区之前的状态，而不是无条件地将线程取消状态设置为 **PTHREAD_CANCEL_ENABLE**。这样可以使从一个临界区中调用 process_transaction 的情况安全不出错——在这种情况下，这个函数会将线程取消状态恢复到调用之前的状态。

4.2.3 何时使用线程取消

通常来说，除非特殊情况，不应使用线程取消结束一个线程的执行。通常情况下，当需要线程退出的情况下通知线程然后等待线程自动退出才是更好的策略。我们将在本章的随后内容中，以及第五章“进程间通信”中讨论线程通信的技术。

4.3 线程专有数据（Thread-Specific Data）

与进程不同，一个程序中的所有线程运行在同一个地址空间中。着表示如果一个线程修改了内存中的一个位置（例如，一个全局变量），则其它所有线程都会发现这个变化。因此多个线程可以同时操作同一块数据而不依赖进程间通信技术（在第五章中介绍了这些技术）。

尽管数据是共享的，每个线程都有单独的调用堆栈。因此每个线程都可以单独执行自己的代码，不加变化地调用子程序、从子程序返回。与在单线程程序中一样，每个线程每次调用子程序都会创造一组自己的局部变量；这些变量保存在线程自己的栈上。

不过有时仍然需要将一个变量复制给每个线程一个副本。GNU/Linux 系统通过为每个

线程提供一个*线程专有数据区（thread-specific data area）*。当数据被存放在这个区域时会自动为每个线程创建一个副本；当一个线程修改自己的副本的时候并不会影响其它线程的副本。因为所有的线程共享一个地址空间，线程专有数据不能通过普通的内存地址引用进行访问。GNU/Linux 系统提供了一系列函数用于读取和设置线程专有数据的值。

你想创建多少线程专有数据对象都可以；它们的类型都是 **void***。每个数据对象都通过一个键值进行映射。要创建一个新键值从而为每个线程新创建一个数据对象，调用 **pthread_key_create** 函数。第一个参数是一个指向 **pthread_key_t** 类型变量的指针。新创建的键值将被保存在这个变量中（译者注：这句在原文中缺失，根据上下文意思以及 pthread_key_create(2) 手册页补充）。随后，这个键值可以被任意线程用于访问对应数据对象的属于自己的副本。传递给 **pthread_key_create** 的第二个参数是一个清理函数，如果你在这里传递一个函数指针，则 GNU/Linux 系统将在线程退出的时候以这个键值对应的数据对象为参数自动调用这个清理函数。清理函数非常有用，因为即使当线程在任何一个非特定运行时刻被取消，这个线程函数也会被保证调用。如果对应的数据对象是一个空指针，清理函数将不会被调用。如果你不需要调用清理函数，你可以在这里传递空指针。

创建了键值之后，可以通过调用 **pthread_setspecific** 设定相应的线程专有数据值。第一个参数是键值，而第二个参数是一个指向要设置的数据的 **void***指针。以键值为参数调用 **pthread_getspecific** 可重新获取一个已经设置的线程专有数据。

假设这样一种情况，你的程序将一个任务分解以供多个线程执行。为了进行审计，每个线程都将分配一个单独的日志文件，用于记录对应线程的任务完成进度。线程专有数据区是为每个单独线程保存对应的日志文件指针的最方便的地点。

列表 4.7 展示了实现这个目标的一种方法。在这个例子中，**main** 函数创建了一个用于保存日志文件指针的线程专有数据区，随后将标识这个数据区的键值保存在 **thread_log_key** 中。因为 **thread_log_key** 是一个全局变量，所有线程共享对它的访问。每当一个线程开始执行的时候均会打开一个日志文件并由这个键值进行映射。之后，任何一个线程均可调用 **write_to_thread_log** 函数将一条信息写入线程对应的日志文件中。这个函数从线程专有数据中获取文件指针并将信息写入文件。

代码列表 4.7 （tsd.c）通过线程专有数据实现的每线程日志

```
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>
/* 用于为每个线程保存文件指针的 TSD 键值。*/
static pthread_key_t thread_log_key;

/* 将 MESSAGE 写入当前线程的日志中。*/
void write_to_thread_log (const char* message)
{
FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
fprintf (thread_log, “%s\n”, message);
}

/* 将日志文件指针 THREAD_LOG 关闭。*/
void close_thread_log (void* thread_log)
```

```
  {
fclose ((FILE*) thread_log);
  }

  void* thread_function (void* args)
  {
char thread_log_filename[20];
FILE* thread_log;

/* 生成当前线程使用的日志文件名。*/
sprintf (thread_log_filename, “thread%d.log”, (int) pthread_self ());
/* 打开日志文件。*/
thread_log = fopen (thread_log_filename, “w”);
/* 将文件指针保存在 thread_log_key 标识的 TSD 中。*/
pthread_setspecific (thread_log_key, thread_log);

write_to_thread_log (“Thread starting.”);
/* 在这里完成线程任务……*/
return NULL;
  }

  int main ()
  {
int i;
pthread_t threads[5];

/* 创建一个键值，用于将线程日志文件指针保存在 TSD 中。
   调用 close_thread_log 以关闭这些文件指针。*/
pthread_key_create (&thread_log_key, close_thread_log);
/* 创建线程以完成任务。*/
for (i = 0; i < 5; ++i)
  pthread_create (&(threads[i]), NULL, thread_function, NULL);
/* 等待所有线程结束。*/
for (i = 0; i < 5; ++i)
  pthread_join (threads[i], NULL);
return 0;
  }
```

我们看到，**thread_function** 不需要关闭日志文件。这是因为当 TSD 键值被创建的时候我们将 **close_thread_log** 指定为这个 TSD 的清理函数。当线程退出的时候，GNU/Linux 将以 **thread_log_key** 所映射的值作为参数调用这个函数。这个函数会负责关闭文件指针。

4.3.1 清理句柄

线程专有数据的清理函数可以很有效地防止在线程退出或被取消的时候出现资源泄漏的问题。不过在有些情况下，我们希望创建一个清理函数却不希望为每个线程创建一个线程专有数据对象。出于这种需求，GNU/Linux 提供了清理句柄。

清理句柄就是一个当线程退出时被自动调用的函数。清理句柄函数接受一个 void*类型的参数，且这个参数在注册清理句柄的时候被同时确定——这样就可以很方便地允许用同一个清理函数清理多份资源实例。

清理句柄是一个临时性的工具，只在当线程被取消或中途退出而不是正常结束运行的时候被调用。在一般情况下，程序应该显式释放分配的资源并清除已经设置的清理句柄。

通过提供两个参数（一个指向清理函数的函数指针和一个作为清理函数参数的 void*类型的值）调用 **pthread_cleanup_push** 可以创建一个清理句柄。对 **pthread_cleanup_push** 的调用可以通过调用 **pthread_cleanup_pop** 进行平衡：**pthread_cleanup_pop** 会取消对一个句柄的注册。为简便操作起见，**pthread_cleanup_pop** 函数接受一个 **int** 类型的参数；如果这个参数为非零值，则在取消注册这个句柄的同时，清理句柄将被执行。

列表 4.8 中的程序片断展示了通过使用清理句柄确保动态分配的缓存在线程结束的时候被释放的方法。

代码列表 4.8 （cleanup.c）用于展示清理句柄的程序片断

```
#include <malloc.h>
#include <pthread.h>

/* 分配临时缓冲区。*/
void* allocate_buffer (size_t size)
{
  return malloc (size);
}

/* 释放临时缓冲区。*/
void deallocate_buffer (void* buffer)
{
  free (buffer);
}

void do_some_work ()
{
  /* 分配临时缓冲区。*/
  void* temp_buffer = allocate_buffer (1024);
  /* 为缓冲区注册清理句柄以确保当线程退出或被取消的时候自动释放。*/
  pthread_cleanup_push (deallocate_buffer, temp_buffer);
```

```
  /* 在这里完成一些任务，其中可能出现对 pthread_exit 的调用，
     线程也可能在此期间被取消。*/

  /* 取消对清理句柄的注册。因为我们传递了一个非零值作为参数，
     清理句柄 deallocate_buffer 将被调用以释放缓存。*/
  pthread_cleanup_pop (1);
}
```

4.3.2 C++中的线程清理方法

C++程序员习惯于通过将清理代码包装在对象析构函数中以获得“免费”的资源清理（译者注：C++重要设计原则 RAII，Resource Acquisition is Initialization 即是如此）。当由于当前块的结束或者由于 C++异常的抛出导致对象的生命期结束的时候，C++确保自动对象的析构函数（如果存在）会被自动调用。这对确保无论代码块如何结束均能调用清理代码块有很大的帮助。

但是，如果一个线程运行中调用了 pthread_exit，C++并不能保证线程的栈上所有自动对象的析构函数将被调用。不过可以通过一个很聪明的方法来获得这个保证：通过抛出一个特别设计的异常，然后在顶层的栈框架内再调用 pthread_exit 退出线程。

列表 4.9 中的程序展示了这种技巧。通过利用这个技巧，函数通过抛出一个 ThreadExitException 异常而不是直接调用 pthread_exit 来尝试退出线程。因为这个程序在顶层栈框架内被捕捉，当程序捕捉到异常的时候，所有在栈上分配的自动对象均已被销毁。

代码列表 4.9 （cxx-exit.cpp）利用 C++异常，安全退出线程

```
#include <pthread.h>

class ThreadExitException
{
public:
  /* 创建一个通过异常进行通知的线程退出方式。RETURN_VALUE 为线程返回值。*/
  ThreadExitException (void* return_value)
    : thread_return_value_ (return_value)
  {
  }

  /* 实际退出线程。返回值由构造函数中指定。*/
  void* DoThreadExit ()
  {
    pthread_exit (thread_return_value_);
  }
```

```
private:
  /* 结束线程时将使用的返回值。*/
  void* thread_return_value_;
};

void do_some_work ()
{
  while (1) {
    /* Do some useful things here... */
    if (should_exit_thread_immediately ())
    throw ThreadExitException (/* thread's return value = */ NULL);
  }
}

void* thread_function (void*)
{
  try {
    do_some_work ();
  }
  catch (ThreadExitException ex) {
    /* Some function indicated that we should exit the thread. */
  ex.DoThreadExit ();
  }
  return NULL;
}
```

4.4 同步和临界代码段

使用线程编程可能需要非常高的技巧，因为多线程程序大多也是并行程序。在这种情况下程序员无从确认系统调度两个线程所采用的特定顺序。有时可能某个线程会连续运行很长时间，但系统也可能在几个线程之间飞快地来回切换。在一个多处理器系统中，几个线程可能如“并行”字面所示，在不同处理器上同时运行。

调试多线程程序可能很困难，因为你可能无法轻易重现导致 bug 出现的情况。可能你某一次运行程序的时候一切正常，而下一次运行的时候却发现程序崩溃。没有办法让系统完全按照完全相同的次序调度这些线程。

导致多线程程序出现 bug 的最根本原因是不同线程访问相同的数据。如前所示例，这是线程最强大的一个特征，但同时也是一个非常危险的特征。如果当一个线程正在更新一个数据的过程中另外一个线程访问同一个数据，很可能导致混乱的出现。很多有 bug 的多线程程序中包含一些代码要求某个线程比另外的线程更经常——或更快——被调用才能正常工作。这种 bug 被称为“竞争状态”；不同线程在更新一个数据结构的过程中出现相互竞争。

4.4.1 竞争状态

假设你的程序利用一些线程并行处理一个队列中的任务。这个队列用一个 **struct job** 对象组成的链表来表示。

每当一个线程结束操作，它都将检查队列中是否有等待处理的任务。如果 **job_queue** 不为空，这个线程将从链表中移除第一个对象，然后把 **job_queue** 指向链表中的下一个对象。

处理任务的线程函数差不多看起来像是列表 4.10 中的样子。

代码列表 4.10 (job-queue1.c) 从队列中删除任务的线程函数

```
#include <malloc.h>

struct job {
  /* 用于连接链表的域 */
  struct job* next;

  /* 其它的域，用于描述需要处理的任务 */
};

/* 一个链表的等待任务 */
struct job* job_queue;

void* thread_function (void* arg)
{
  while (job_queue != NULL) {
    /* 获取下一个任务 */
    struct job* next+job = job_queue;
    /* 从列表中删除这个任务 */
    job_queue = jhob_queue->next;
    /* 进行处理 */
    process_job (next_job);
    /* 清理 */
    free (next_job);
  }
  return NULL;
}
```

现在假设有两个线程几乎同时完成了处理工作，但队列中只剩下一个队列。第一个线程检查 **job_queue** 是否为空；发现不是，则该线程进入循环，将指向任务对象的指针存入 **next_job**。这时，Linux 正巧中断了第一个线程而开始运行第二个线程。这第二个线程也检查任务队列，发现队列中的任务，然后将这同一个任务赋予 **next_job**。在这种不幸的巧合下，两个线程将处理同一个任务。

使情况更糟糕一点，我们假设一个线程已将任务从队列中删除，使 **job_queue** 为空。当另一个线程执行 **job_queue->next** 的时候将会产生一个段错误。

这是一个竞争条件的例子。在“幸运”的情况下，刚才提到的对这两个线程的特定调度顺序不会出现，竞争条件也许永远也不会被发现。只有在其它一些情况下，譬如当程序运行在一个高负载的系统（或者，在一个重要客户的新购置的多处理器服务器系统中！）这个 bug 可能会忽然出现。

要消灭竞争状态，你需要通过某种方法使操作具有原子性。一个原子操作是不可分割不可中断的单一操作；一旦这个操作过程开始，在结束之前将无法被暂停或中断，也不会有其它的操作同时进行。在这个特定的例子中，你需要将“检查 **job_queue**；如果它不为空，删除第一个任务”整个过程作为一个原子操作。

4.4.2 互斥体

对于刚才这个任务队列竞争状态问题的解决方法就是限制在同一时间只允许一个线程访问任务队列。当一个线程开始检查任务队列的时候，其它线程应该等待直到第一个线程决定是否处理任务，并在确定要处理任务时删除了相应任务之后才能访问任务队列。

要实现等待这个操作需要操作系统的支持。GNU/Linux 提供了互斥体（*mutex，全称 MUTual EXclusion locks，互斥锁*）。互斥体是一种特殊的锁：同一时刻只有一个线程可以锁定它。当一个锁被某个线程锁定的时候，如果有另外一个线程尝试锁定这个互斥体，则这第二个线程会被阻塞，或者说被置于等待状态。只有当第一个线程释放了对互斥体的锁定，第二个线程才能从阻塞状态恢复运行。GNU/Linux 保证当多个线程同时锁定一个互斥体的时候不会产生竞争状态；只有一个线程可能成功锁定，其它线程均将被阻塞。

将互斥体想象成一个盥洗室的门锁。第一个到达门口的人进入盥洗室并且锁上门。如果盥洗室被占用期间有第二个人想要使用，他将发现门被锁住因此自己不得不在门外等待，直到里面的人离开。

要创建一个互斥体，首先需要创建一个 **pthread_mutex_t** 类型的变量，并将一个指向这个变量的指针作为参数调用 **pthread_mutex_init**。而 **pthread_mutex_init** 的第二个参数是一个指向互斥体属性对象的指针；这个对象决定了新创建的互斥体的属性。与 **pthread_create** 一样，如果属性对象指针为 NULL，则默认属性将被赋予新建的互斥体对象。这个互斥体变量只应被初始化一次。下面这段代码展示了创建和初始化互斥体的方法。

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

另外一个相对简单的方法是用特殊值 **PTHREAD_MUTEX_INITIALIZER** 对互斥体变量进行初始化。这样就不必再调用 **pthread_mutex_init** 进行初始化。这对于全局变量（及 C++ 中的静态成员变量）的初始化非常有用。因此上面那段代码也可以写成这样：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

线程可以通过调用 **pthread_mutex_lock** 尝试锁定一个互斥体。如果这个互斥体没有被锁定，则这个函数调用会锁定它然后立即返回。如果这个互斥体已经被另一个线程锁定，则 **pthread_mutex_lock** 会阻塞调用线程的运行，直到持有锁的线程解除了锁定。同一时间可以有多个线程在一个互斥体上阻塞。当这个互斥体被解锁，只有一个线程（以不可预知的方式被选定的）会恢复执行并锁定互斥体，其它线程仍将处于锁定状态。

调用 **pthread_mutex_unlock** 将解除对一个互斥体的锁定。始终应该从锁定了互斥体的线程调用这个函数进行解锁。

代码列表 4.11 展示了另外一个版本的任务队列。现在我们用一个互斥体保护了这个队

列。访问这个队列之前（不论读写）每个线程都会锁定一个互斥体。只有当检查队列并移除任务的整个过程完成，锁定才会被解除。这样可以防止前面提到的竞争状态的出现。

代码列表 4.11 (job-queue2.c) 任务队列线程函数，用互斥体保护

```
#include <malloc.h>
#include <pthread.h>

struct job {
  /* 维护链表结构用的成员。*/
  struct job* next;

  /* 其它成员，用于描述任务。*/
};

/* 等待执行的任务队列。*/
struct job* job_queue;

/* 保护任务队列的互斥体。*/
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

/* 处理队列中剩余的任务，直到所有任务都经过处理。*/
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* 锁定保护任务队列的互斥体。*/
    pthread_mutex_lock (&job_queue_mutex);
    /* 现在可以安全地检查队列中是否为空。*/
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* 获取下一个任务。*/
      next_job = job_queue;
      /* 从任务队列中删除刚刚获取的任务。*/
      job_queue = job_queue->next;
    }
    /* 我们已经完成了对任务队列的处理，因此解除对保护队列的互斥体的锁定。*/
    pthread_mutex_nlock (&job_queue_mutex);

    /* 任务队列是否已经为空？如果是，结束线程。*/
    if (next_job == NULL)
      break;
```

```
      /* 执行任务。*/
      proces_job (next_job);
      /* 释放资源。*/
      free (next_job);
    }
    return NULL;
}
```

所有对 **job_queue** 这个共享的指针的访问都在 **pthread_mutex_lock** 和 **pthread_mutex_unlock** 两个函数调用之间进行。任何一个 **next_job** 指向的任务对象，都是在从队列中移除之后才处理的；这个时候其它线程都无法继续访问这个对象。

注意当队列为空（也就是 **job_queue** 为空）的时候我们没有立刻跳出循环，因为如果立刻跳出，互斥对象将继续保持锁定状态从而导致其它线程再也无法访问整个任务队列。实际上，我们通过设定 **next_job** 为空来标识这个状态，然后在将互斥对象解锁之后再跳出循环。

用互斥对象锁定 **job_queue** 不是自动完成的；你必须自己选择是否在访问 **job_queue** 之前锁定互斥体对象以防止并发访问。如下例，向任务队列中添加一个任务的函数可以写成这个样子：

```
void enqueue_job (struct job* new_job)
{
  pthread_mutex_lock (&job_queue_mutex);
  new_job->next = job_queue;
  job_queue = new-job;
  pthread_mutex_unlock (&job_queue_mutex);
}
```

4.4.3 互斥体死锁

互斥体提供了一种由一个线程阻止另一个线程执行的机制。这个机制导致了另外一类软件错误的产生：死锁。当一个或多个线程处于等待一个不可能出现的情况的状态的时候，我们称之为死锁状态。

最简单的死锁可能出现在一个线程尝试锁定一个互斥体两次的时候。当这种情况出现的时候，程序的行为取决于所使用的互斥体的种类。共有三种互斥体：

- 锁定一个快速互斥体（*fast mutex*，默认创建的种类）会导致死锁的出现。任何对锁定互斥体的尝试都会被阻塞直到该互斥体被解锁的时候为止。但是因为锁定该互斥体的线程在同一个互斥体上被锁定，它永远无法接触互斥体上的锁定。
- 锁定一个递归互斥体（*recursive mutex*）不会导致死锁。递归互斥体可以很安全地被锁定多次。递归互斥体会记住持有锁的线程调用了多少次 **pthread_mutex_lock**；持有锁的线程必须调用同样次数的 **pthread_mutex_unlock** 以彻底释放这个互斥体上的锁而使其它线程可以锁定该互斥体。
- 当尝试第二次锁定一个纠错互斥体（*error-checking mutex*）的时候，GNU/Linux 会自动检测并标识对纠错互斥体上的双重锁定；这种双重锁定通常会导致死锁的出现。第二次尝试锁定互斥体时 pthread_mutex_lock 会返回错误码 EDEADLK。

默认情况下 GNU/Linux 系统中创建的互斥体是第一种，快速互斥体。要创建另外两种互斥体，首先应声明一个 **pthread_mutexattr_t** 类型的变量并且以它的地址作为参数调用 **pthread_mutexattr_init** 函数，以对它进行初始化。然后调用 **pthread_mutexattr_setkind_np** 函数设置互斥体的类型；该函数的第一个参数是指向互斥体属性对象的指针，第二个参数如果是 **PTHREAD_MUTEX_RECURSIVE_NP** 则创建一个递归互斥体，或者如果是 **PTHREAD_MUTEX_ERRORCHECK_NP** 则创建的将是一个纠错互斥体。当调用 **pthread_mutex_init** 的时候传递一个指向这个属性对象的指针以创建一个对应类型的互斥体，之后调用 **pthread_mutexattr_destroy** 销毁属性对象。

下面的代码片断展示了如何创建一个纠错互斥体；

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```

如“np”后缀所指明的，递归和纠错两种互斥体都是 GNU/Linux 独有的，不具有可移植性（译者注：np 为 non-portable 缩写）。因此，通常不建议在程序中使用这些类型的互斥体。（当然，纠错互斥体对查找程序中的错误可能很有帮助。）

4.4.4 非阻塞互斥体测试

有时候我们需要检测一个互斥体的状态却不希望被阻塞。例如，一个线程可能需要锁定一个互斥体，但当互斥体已经锁定的时候，这个线程还可以处理其它的任务。因为 **pthread_mutex_lock** 会阻塞直到互斥体解锁为止，所以我们需要其它的一些函数来达到我们的目的。

GNU/Linux 提供了 **pthread_mutex_trylock** 函数作此用途。当你对一个解锁状态的互斥体调用 **pthread_mutex_trylock** 时，就如调用 **pthread_mutex_lock** 一样会锁定这个互斥体；**pthread_mutex_trylock** 会返回 0。而当互斥体已经被其它线程锁定的时候，**pthread_mutex_trylock** 不会阻塞。相应的，**pthread_mutex_trylock** 会返回错误码 **EBUSY**。持有锁的其它线程不会受到影响。你可以稍后再次尝试锁定这个互斥体。

4.4.5 线程信号量

之前的例子中，我们让几个线程从一个队列中取出并处理任务，每个线程函数都会尝试从队列中取得任务并当没有任务的时候结束线程函数。如果事先给队列中添加好任务，或者至少以比处理线程提取任务更快的速度向队列中添加新任务，这个模型没有问题。但如果工作线程速度太快了，任务列表会被清空而处理线程会退出，而再有新任务到达的时候就没有线程处理任务了。因此，我们更希望有这样一种机制：让工作线程阻塞以等待新的任务的到达。

信号量可以很方便地做到这一点。信号量是一个用于协调多个线程的计数器。如互斥体一样，GNU/Linux 保证对信号量的取值和赋值操作都是安全的，不会造成竞争状态。

每个信号量都有一个非负整数作为计数。信号量支持两种基本操作：

- “等待”（*wait*）操作会将信号量的值减一。如果信号量的值已经是一，这个操作会阻塞直到（由于其它线程的一些操作）信号量的值成为正值。当信号量的值成为正值的时候，等待操作会返回，同时信号量的值减一。
- “投递”（*post*）操作会将信号量的值加一。如果信号量之前的值为零，并且有其它线程在等待过程中阻塞，其中一个线程就会解除阻塞状态并结束等待状态（同时将信号量的值重置为 0）。

需要注意的是 GNU/Linux 提供了两种有少许不同的信号量实现。一种是我们这里所说的兼容 POSIX 标准的信号量实现。当处理线程之间的通信的时候可以使用这种实现。另一种实现常用于进程间通信，在 5.2 节“进程信号量”中进行了介绍。如果要使用信号量，应包含头文件 <semaphore.h>。

信号量是用**sem_t**类型的变量表示的。在使用一个信号量之前，你需要通过**sem_init**函数对它进行初始化；**sem_init**接受一个指向这个信号量变量的指针作为第一个参数。第二个参数应为 0[2]，而第三个参数则指定了信号量的初始值。当你不再需要一个信号量之后，应该调用**sem_destory**销毁它。

我们可以用 **sem_wait** 对一个信号量执行等待操作，用 **sem_post** 对一个信号量执行投递操作。同时 GNU/Linux 还提供了一个非阻塞版本的信号量等待函数 **sem_trywait**。这个函数类似 **pthread_mutex_trylock**——如果当时的情况应该导致阻塞，这个函数会立即返回错误代码 **EAGAIN** 而不是造成线程阻塞。

GNU/Linux 同时提供了一个用于获取信号量当前值的函数 **sem_getvalue**。这个函数将信号量的值保存在第二个参数（指向一个 **int** 类型变量的指针）所指向的变量中。不过，你不应使用从这个函数得到的值作为判断应该执行等待还是投递操作的依据。因为这样做可能导致竞争状态的出现：其它线程可能在 **sem_getvalue** 和随后的其它信号量函数之间开始执行并修改信号量的值。应使用属于原子操作的等待和投递代替这种做法。

回到我们的任务队列例子中。我们可以使用一个信号量来计算在队列中等待处理的任务数量。代码列表 4.12 使用一个信号量控制队列。函数 **enqueue_job** 负责向队列中添加一个任务。

代码列表 4.12 (job-queue3.c) 用信号量控制的任务队列

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job {
  /* 维护链表结构用的成员。*/
  struct job* next;

  /* 其它成员，用于描述任务。*/
};

/* 等待执行的任务队列。*/
struct job* job_queue;
```

[2] 非 0 值表示的是可以在进程之间共享的信号量。GNU/Linux 系统中的这种信号量不支持在进程之间共享。

```
/* 用于保护 job_queue 的互斥体。*/
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

/* 用于计数队列中任务数量的信号量。*/
sem_t job_queue_count;

/* 对任务队列进行唯一的一次初始化。*/
void initialize_job_queue ()
{
  /* 队列在初始状态为空。*/
  job_queue = NULL;
  /* 初始化用于计数队列中任务数量的信号量。它的初始值应为 0。*/
  sem_init (&job_queue_count, 0, 0);
}

/* 处理队列中的任务，直到队列为空。*/

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* 等待任务队列信号量。如果值为正，则说明队列中有任务，应将信号量值减一。
       如果队列为空，阻塞等待直到新的任务加入队列。*/
    sem_wait (&job_queue_count);

    /* 锁定队列上的互斥体。*/
    pthread_mutex_lock (&job_queue_mutex);
    /* 因为检测了信号量，我们确信队列不是空的。获取下一个任务。*/
    next_job = job_queue;
    /* 将这个任务从队列中移除。*/
    job_queue = job_queue->next;
    /* 解除队列互斥体的锁定因为我们已经不再需要操作队列。*/
    pthread_mutex_unlock (&job_queue_mutex);

    /* 处理任务。*/
    process_job (next_job);
    /* 清理资源。*/
    free (next_job);
  }
  return NULL;
}
```

```
/* 向任务队列添加新的任务。*/

void enqueue_job (/* 在这里传递特定于任务的数据…… */)
{
  struct job* new_job;

  /* 分配一个新任务对象。*/
  new_job = (struct job*) malloc (sizeof (struct job));
  /* 在这里设置任务中的其它字段……*/

  /* 在访问任务队列之前锁定列表。*/
  pthread-mutex_lock (&job_queue_mutex);
  /* 将新任务加入队列的开端。*/
  new_job->next = job_queue;
  job_queue = new_job;

/* 投递到信号量通知有新任务到达。如果有线程被阻塞等待信号量，一个线程就会恢复执行并处
   理这个任务。*/
  sem_post (&job_queue_count);

  /* 将任务队列解锁。*/
  pthread_mutex_unlock (&job_queue_mutex);
}
```

在从队列前端取走任务之前，每个线程都会等待信号量。如果信号量的值是 0，则说明任务队列为空，线程会阻塞，直到信号量的值恢复正值（表示有新任务到达）为止。

函数 **enqueue_job** 将一个任务添加到队列中。就如同 **thread_function** 函数，它需要在修改队列之前锁定它。在将任务添加到队列之后，它将信号量的值加一以表示有新任务到达。在列表 4.12 中的版本中，工作线程永远不会退出；当没有任务的时候所有线程都会在 **sem_wait** 中阻塞。

4.4.6 条件变量

我们已经展示了如何在两个线程同时访问一个变量的时候利用互斥体进行保护，以及如何使用信号量实现共享的计数器。条件变量是 GNU/Linux 提供的第三种同步工具；利用它你可以在多线程环境下实现更复杂的条件控制。

假设你要写一个永久循环的线程，每次循环的时候执行一些任务。不过这个线程循环需要被一个标志控制：只有当标志被设置的时候才运行，标志被清除的时候线程暂停。

代码列表 4.13 显示了你可以通过在不断自旋（重复循环）以实现这一点。每次循环的时候，线程都检查这个标志是否被设置。因为有多个线程都要访问这个标志，我们使用一个互斥体保护它。这种实现虽然可能是正确的，但是效率不尽人意。当标志没有被设置的时候，线程会不断循环检测这个标志，同时会不断锁定、解锁互斥体，浪费 CPU 时间。你真正需要的是这样一种方法：当标志没有设置的时候让线程进入休眠状态；而当某种特定条件出现

时，标志位被设置，线程被唤醒。

代码列表 4.13 (spin-condvar.c) 一个简单的条件变量实现

```
#include <pthread.h>

int thread_flag;
pthread_mutex_t thread_flag_mutex;

void initialize_flag()
{
  pthread_mutex_init (&thread_flag_mutex, NULL);
  thread_flag = 0;
}

/* 当标志被设置的时候反复调用 do_work，否则自旋等待。*/

void* thread_function (void* thread_arg)
{
  while (1) {
    int flag_is_set;

    /* 用一个互斥体保护标志。*/
    pthread-mutex_lock (&thread_flag_mutex);
    flag_is_set = thread_flag;
    pthread_mutex_unlock (&thread_flag_mutex);

    if (flag_is_set)
      do_work ();
    /* 否则什么也不做，直接进入下一次循环。*/
  }
  return NULL;
}

/* 将线程标志的值设置为 flag_value。*/
void set_thread_flag (int flag_value)
{
  /* 用一个互斥体保护线程标志。*/
  pthread_mutex_lock (&thread_flag_mutex);
  thread_flag = flag_value;
  pthread_mutex_unlock (&thread-flag_mutex);
}
```

条件变量将允许你实现这样的目的：在一种情况下令线程继续运行，而相反情况下令线

程阻塞。只要每个可能涉及到改变状态的线程正确使用条件变量，Linux 将保证当条件改变的时候由于一个条件变量的状态被阻塞的线程均能够被激活。

如同信号量，线程可以对一个条件变量执行等待操作。如果线程 A 正在等待一个条件变量，它会被阻塞直到另外一个线程，设为线程 B，向同一个条件变量发送信号以改变其状态。不同于信号量，条件变量没有计数值，也不占据内存空间；线程 A 必须在 B 发送信号之前开始等待。如果 B 在 A 执行等待操作之前发送了信号，这个信号就丢失了，同时 A 会一直阻塞直到其它线程再次发送信号到这个条件变量。

你可以这样使用条件变量以使前面那个例子运行得更有效率：

- thread_function 中的循环检查标志。如果标志没有被设置则线程开始等待条件变量。
- set_thread_flag 函数在改变了标志的值之后向条件变量发送信号。这样，如果 thread_function 处于等待条件变量的状态，则它会恢复运行并重新检查标志。

这里有一个问题：检查状态的操作与对条件变量进行的等待或发送信号操作之间可能形成竞争状态。假设 **thread_function** 检查了标志，发现标志没有被设置。这时候，Linux 调度器暂停了这条线程而返回运行主线程。很偶然的，主线程正处于 **set_thraed_flag** 中。它设置了标志，然后向条件变量发送了信号。因为这个时候没有线程在等待这个条件变量的信号（别忘了，**thread_function** 在开始等待信号量上的事件之前就被暂停了执行），这个信号就此丢失了。现在，Linux 重新调度并回到原先的线程，这个线程开始等待信号并很可能会永远等待下去。

要解决这个问题，我们需要用一个互斥体将标志变量和条件变量绑定在一起。幸运的是，GNU/Linux 刚好提供了这个机制。每个条件变量都必须与一个互斥体共同使用，以防止这种竞争状态的发生。这种设计下，线程函数应遵循以下步骤：

1. **thread_function** 中的循环首先锁定互斥体并且读取标志变量的值。
2. 如果标志变量已经被设定，该线程将互斥体解锁然后执行工作函数
3. 如果标志没有被设置，该线程自动锁定互斥体并开始等待条件变量的信号

这里最关键的特点就在第三条。这里，GNU/Linux 系统允许你用一个原子操作完成解除互斥体锁定和等待条件变量信号的过程而不会被其它线程在中途插入执行。这就避免了在 **thread_function** 中检测标志和等待条件变量的过程中其它线程修改标志变量并对条件变量发送信号的可能性。

条件变量用 **pthread_cond_t** 类型表示。别忘了每个条件变量都必须与一个互斥体伴生。这里是可以用于操作条件变量的函数。

- 通过调用 **pthread_cond_init** 初始化一个条件变量。第一个参数是一个指向 **pthread_cond_t** 变量的指针。第二个参数是一个指向条件变量属性对象的指针；这个参数在 GNU/Linux 系统中是被忽略的。
  互斥体对象必须单独被初始化。具体请参考 4.4.2 节“互斥体”
- 调用 **pthread_cond_signal** 向一个条件变量发送信号。在该条件变量上阻塞的线程将被恢复运行。如果没有线程正在等待这个信号，则这个信号会被忽略。该函数的参数是一个指向 **pthread_cond_t** 类型变量的指针。
  相似的，**pthread_cond_broadcast** 函数会将所有等待该条件变量的线程解锁而不是仅仅解锁一个线程。
- 调用 **pthred_cond_wait** 会让调用线程阻塞直到条件变量收到信号。该函数的第一个参数是指向一个 **pthread_cond_t** 类型变量的指针，第二个参数是指向一个 **pthread_mutex_t** 类型变量的指针。
  当调用 **pthread_cond_wait** 的时候，互斥体对象必须已经被调用线程锁定。这个函

数以一个原子操作解锁互斥体并锁定条件变量等待信号。当信号到达且调用线程被解锁之后，**pthread_cond_wait** 自动申请锁定互斥体对象。

当你的程序要改变你利用条件变量所维护的程序状态的时候，始终应该遵循以上这些步骤。（在我们的例子中，我们要保护的就是标志变量的状态，所以每当试图改版标志变量的值的时候都应该遵循这些步骤。）

1. 锁定与条件变量伴生的互斥体。
2. 执行可能改变程序状态的指令（在我们的例子中，修改标志）。
3. 向条件变量投递或广播信号。这取决于我们希望的行为。
4. 将与条件变量伴生的互斥体解锁。

代码列表 4.14 再次展示了之前的那个例子，不过现在改用条件变量保护标志。注意，在 **thread_function** 中，在检测 **thread_flag** 的值之前我们锁定了互斥体。这个锁会被 **pthread_cond_wait** 在阻塞之前自动释放，并在阻塞结束后自动重新获取。同时也要注意，**set_thread_flag** 会在设定 **thread_flag** 的值之前自动锁定互斥体并向状态变量（译者注：这里原文为 mutex，疑为 condition variable 笔误）发送信号。

代码列表 4.14 (condvar.c) 用条件变量控制线程

```
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
  /* 初始化互斥体和条件变量。*/
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* 初始化标志变量。*/
  thread_flag = 0;
}

/* 如果标志被设置，则反复调用 do_work；否则阻塞。*/

void* thread_function (void* thread_arg)
{
  /* Loop infinitely. */
  while (1) {
    /* 访问标志之前锁定互斥体。*/
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
  /* 标志被清空。等待条件变量指示标志被改变的信号。信号到达的时候线程解锁，
```

```
    然后再次循环并检查标志。*/
        pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
      /* 当我们到达这里的时候，我们确信标志已经被设置。将互斥体解锁。*/
      pthread_mutex_unlock (&thread_flag_mutex);
      /* 执行任务。*/
      do_work ();
    }
    return NULL;
  }

  /* 将线程标志值设置为 flag_value。*/
  viod set_thread_flag (int flag_value)
  {
    /* 赋值之前先锁定互斥体。*/
    pthread_mutex_lock (&thread_flag_mutex);
    /* 进行赋值操作，然后对等待标志改变而被阻塞的 thread_function 发送信号。但事实
上 thread_function 必须等待互斥体被解锁才能检查标志。*/
    thread_flag = flag_value;
    pthread_cond_signal (&thread_flag_cv);
    /* 解除互斥体锁定 */
    pthread_mutex_unlock (&thread_flag_mutex);
  }
```

条件变量所保护的状态可以相当复杂。不过，在改变任何状态之前都应该首先锁定一个互斥体，并且在修改操作之后向条件变量发送信号。

条件变量也可以用于不涉及程序状态的情况，而仅用作一种让一个线程阻塞等待其它线程唤醒的机制。信号量也可用于这个目的。两者之前的主要区别是，当没有线程处于阻塞状态的时候信号量会“记住”唤醒下一个被阻塞的线程，而条件变量只是简单地丢弃这个信号。另外，信号量只能发送一个唤醒信息给一个线程，而 **pthread_cond_broadcast** 可以同时唤醒不限数量的可以被唤醒的线程。

4.4.7 两个或多个线程的死锁

死锁可能发生在这样一种情况：两个（或更多）线程都在阻塞等待一个只能被其它线程引发的事件。例如，当线程 A 等待线程 B 向一个条件变量发送信号而线程 B 也在等待线程 A 向一个条件变量发送信号的时候，因为两个线程都永远无法发送对方等待的信号，死锁就出现了。你应该尽力避免这种情况的发生，因为这种错误很难被察觉。

一个可能引发死锁的常见错误是多个线程试图锁定同一组对象。假设有这样一个程序，有两个线程运行不同的线程函数却尝试锁定相同的两个互斥体。假设线程 A 先锁定互斥体 A 而后锁定互斥体 B，而线程 B 先锁定互斥体 B 而后尝试锁定互斥体 A。在一个非常不幸的情况下，Linux 可能让线程 A 运行到成功锁定互斥体 A 之后，然后转而运行线程 B 直到锁定互斥体 B。接下来，两个线程都被阻塞在对方持有的互斥体上而再也无法继续运行。

不仅是针对互斥体等同步对象，当针对更多种类的资源，例如文件或设备上的锁定进行同步的时候，更容易造成这种死锁问题。这种问题出现的原因是一组线程以不同的顺序锁定

同一组资源。解决这个问题的方法就是确保所有线程锁定这些资源的顺序相同，这样就可以避免死锁的出现。

4.5 GNU/Linux 线程实现

GNU/Linux 平台上的 POSIX 线程实现与其它许多类 UNIX 操作系统上的实现有所不同：在 GNU/Linux 系统中，线程就是用进程实现的。每当你用 **pthread_create** 创建一个新线程的时候，Linux 创建一个新进程运行这个线程的代码。不过，这个进程与一般由 **fork** 创建的进程有所不同；具体来说，新进程与父进程共享地址空间和资源，而不是分别获得一份拷贝。

列表 4.15 中的程序 **thread-pid** 演示了这一点。这个程序首先创建一个线程，随后两个线程都调用 getpid 并打印各自的进程号，最后分别无限循环。

代码列表 4.15 (thread-pid) 打印线程的进程号

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

void* thread_function (void* arg)
{
  fprintf (stderr, "child thread pid is %d\n", (int) getpid ());
  /* 无限循环 */
  while (1);
  return NULL;
}

int main ()
{
  pthread_t thread;
  fprintf (stderr, "main thread pid is %d\n", (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* 无限循环 */
  while (1);
  return 0;
}
```

在后台运行这个程序，然后调用 ps x 显示运行中的进程。别忘了随后结束 **pthread_pid** 程序——它浪费无数 CPU 时间却什么也不做。这是一个可能的输出：

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
```

```
% ps x
  PID TTY      STAT   TIME COMMAND
14042 pts/9    S      0:00 bash
14608 pts/9    R      0:01 ./thread-pid
14609 pts/9    S      0:00 ./thread-pid
14610 pts/9    R      0:01 ./thread-pid
14611 pts/9    R      0:00 ps x
% kill 14608
[1]+  Terminated              ./thread-pid
```

Shell 程序的进程控制提示

以 [1] 开头的行是由 shell 程序输出的。当你在后台运行一个程序，shell 会分配一个任务控制代码给这个程序——在这里是 1——并打印这个程序的进程号。如果后台程序终止了，shell 会在你下次执行命令后通知你。

注意这里共有三个进程运行着 **thread-pid** 程序。第一个，进程号是 14608 的，运行的是程序的主函数；第三个，进程号是 14610 的，是我们创建来执行 **thread_function** 的线程。

那么第二个，进程号是 14609 的线程呢？它是“管理线程”，属于 GNU/Linux 线程内部实现细节。管理线程会在一个程序第一次调用 **pthread_create** 的时候自动创建。

4.5.1 信号处理

假设一个多线程程序收到了一个信号。究竟哪个线程的信号处理函数会作出响应？线程和信号之间的互操作在各个 UNIX 变种系统都可能有所不同。在 GNU/Linux 系统中，这个行为的决定因素在于：线程实际是由进程实现的。

因为每个线程都是一个单独的进程，又因为信号是发送到特定进程的，究竟由哪个线程接受信号并不会成为一个问题。一般而言，从程序外发送的信号通常都是发送到程序的主线程。例如，如果一个程序通过 **fork** 调用产生了新进程运行一个多线程程序，父进程将得到新程序主线程所在的进程号，并通过这个进程号发送信号。当你试图向一个多线程程序发送信号的时候，通常也应该遵循这个方法。

要注意的是 GNU/Linux 系统中 pthread 库的实现与 POSIX 线程标准的区别。在注重可移植性的程序中不要依赖程序的特定行为。

在一个多线程程序中，一个线程可以给另一个特定线程发送信号。函数 **pthread_kill** 可以做到这一点。该函数的第一个参数是线程号，第二个参数则是信号的值。

4.5.2 clone 系统调用

虽然同一个程序中产生的线程被实现作不同的进程，所有线程都共享虚拟内存和其它资源。而通过 **fork** 创建的子进程则得到所有这些的独立副本。前一种进程究竟是怎么创建的？

Linux 的 **clone** 系统调用是一个更通用版本的 **fork** 和 **pthread_create**。它允许调用者指定哪些资源应在新旧进程之间共享。同时，**clone** 要求你指定新进程运行所需的栈空间所在的内存区域。虽然我们在这里介绍了这个系统调用以满足读者的好奇心，**clone** 系统调用通常不应该出现在程序中。应该调用 **fork** 创建新进程而调用 **pthread_create** 创建新线程。

4.6 进程 Vs. 线程

对于一些从并发处理中受益的程序而言，多进程还是多线程可能很难被抉择。这里有一些基本方针可以帮助你判断哪种模型更适合你的程序：

- 一个程序的所有线程都必须运行同一个执行文件。而一个新进程则可以通过 **exec** 函数运行一个新的执行文件。
- 由于所有线程共享地址空间和资源，一个错误的线程可能影响所有其它线程。例如，通过未经初始化的指针非法访问内存可能破坏其它线程所使用的内存。
  而一个错误的进程则不会造成这样的破坏因为每个进程都有父进程的地址空间的完整副本。
- 为新进程复制内存会比创建新线程存在性能方面的损失。不过，由于只有当对内存进行写入操作的时候复制操作才会发生，如果新进程只对内存执行读取操作，性能损失可能微乎其微。
- 对于需要精细并行控制的程序，线程是更好的选择。例如，如果一个问题可以被分解为许多相对独立的子任务，用线程处理可能更好。进程适合只需要比较粗糙的并行程序。
- 由于线程之间共享地址空间，共享数据是一件简单的任务。（不过如前所述，必须倍加小心防范竞争状态的出现。）进程之间共享属于要求使用第五章中介绍的各种 IPC 机制。这虽然显得更麻烦而笨重，但同时避免了许多并行程序错误的出现。

第五章：进程间通信

第三章“进程”中我们讨论了进程的创建方法，也展示了一个进程如何获取子进程的退出状态。这可以算是最简单的进程间通信方法，但毋庸置疑，它绝不是是最强大的一种。第三章中所提供的通信机制，对父进程而言，除了通过设置命令行参数和环境变量之外，并没有提供任何的与子进程通信的方法，同样，对于子进程而言，也只有退出代码这唯一一种向父进程返回信息的方法。这些通信机制不允许进程与正在运行中的子进程通信，更不可能允许两个没有派生关系的进程之间自由地对话。

本章介绍的进程间通信机制则完全解除了这些限制。我们将展示供“父子”进程、“无关”进程甚至是分别运行在不同主机的进程之间进行通信的多种方式。

*进程间通信（Interprocss communication, IPC）*是在不同进程之间传递数据的方法。例如，互联网浏览器可以向服务器发送一个请求，随后服务器会传回 HTML 信息。这样的数据传递通常是通过一种功能类似电话线路连接的*套接字*来完成的。另外一个例子，你可以用 **`ls | lpr`** 这个命令将一个目录下的文件名打印出来。Shell 程序会创建一个 **`ls`** 进程和一个 **`lpr`** 进程，然后用一个“管道（用 | 符号表示）”将它们连接起来。管道为这两个进程提供了一种单向通信的渠道。这个例子中，由 **`ls`** 进程向管道写入信息，而 **`lpr`** 进程则从管道读取。

在本章中，我们将讨论五种不同的进程间通信机制：

- 共享内存允许两个进程通过对特定内存地址的简单读写来完成通信过程。
- 映射内存与共享内存的作用相同，不过它需要关联到文件系统中的一个文件上。
- 管道允许从一个进程到另一个关联进程之间的顺序数据传输。
- FIFO 与管道相似，但是因为 FIFO 对应于文件系统中的一个文件，无关的进程也可以完成通信。
- 套接字允许无关的进程、甚至是运行在不同主机的进程之间相互通信。

这些进程间通信机制（IPC）可以按以下标准进行区分：

- 通信对象是否限制为相互关联的进程（即是否有共同的父进程），或者限制为共享同一个文件系统的进程，还是可以为连接到同一个网络中的不同主机上的进程。
- 通信中的一个进程是否限制为仅能读取或者写入数据。
- 允许参加通信的进程的总数。
- 通信进程是否直接在通信机制（IPC）中得到同步——例如，读取数据的进程会等待直到有数据到达时开始读取。

本章中，我们不再讨论那些只能进行有限次数的进程间通信机制，例如通过子进程的退出代码进行通信的方式等。

5.1 共享内存

共享内存是进程间通信中最简单的方式之一。共享内存允许两个或更多进程访问同一块内存，就如同 **`malloc()`** 函数向不同进程返回了指向同一个物理内存区域的指针。当一个进程改变了这块地址中的内容的时候，其它进程都会察觉到这个更改。

5.1.1 快速本地通信

因为所有进程共享同一块内存，共享内存在各种进程间通信方式中具有最高的效率。访问共享内存区域和访问进程独有的内存区域一样快，并不需要通过系统调用或者其它需要切

入内核的过程来完成。同时它也避免了对数据的各种不必要的复制。

因为系统内核没有对访问共享内存进行同步，你必须提供自己的同步措施。例如，在数据被写入之前不允许进程从共享内存中读取信息、不允许两个进程同时向同一个共享内存地址写入数据等。解决这些问题的常用方法是通过使用信号量进行同步。信号量的使用将在下一节中介绍。不过，我们的程序中只有一个进程访问了共享内存，因此在集中展示了共享内存机制的同时，我们避免了让代码被同步逻辑搞得混乱不堪。

5.1.2 内存模型

要使用一块共享内存，进程必须首先分配它。随后需要访问这个共享内存块的每一个进程都必须将这个共享内存绑定到自己的地址空间中。当完成通信之后，所有进程都将脱离共享内存，并且由一个进程释放该共享内存块。

理解 Linux 系统内存模型可以有助于解释这个绑定的过程。在 Linux 系统中，每个进程的虚拟内存是被分为许多页面的。这些内存页面中包含了实际的数据。每个进程都会维护一个从内存地址到虚拟内存页面之间的映射关系。尽管每个进程都有自己的内存地址，不同的进程可以同时将同一个内存页面映射到自己的地址空间中，从而达到共享内存的目的。第八章“**Linux 系统调用**”中第八节“**mlock 族：锁定物理内存**”将对内存页面做深入的探讨。

分配一个新的共享内存块会创建新的内存页面。因为所有进程都希望共享对同一块内存的访问，只应由一个进程创建一块新的共享内存。再次分配一块已经存在的内存块不会创建新的页面，而只是会返回一个标识该内存块的标识符。一个进程如需使用这个共享内存块，则首先需要将它绑定到自己的地址空间中。这样会创建一个从进程本身虚拟地址到共享页面的映射关系。当对共享内存的使用结束之后，这个映射关系将被删除。当再也没有进程需要使用这个共享内存块的时候，必须有一个（且只能是一个）进程负责释放这个被共享的内存页面。

所有共享内存块的大小都必须是系统页面大小的整数倍。系统页面大小指的是系统中单个内存页面包含的字节数。在 Linux 系统中，内存页面大小是 4KB，不过你仍然应该通过调用 **`getpagesize`** 获取这个值。

5.1.3 分配

进程通过调用 **`shmget`**（SHared Memory GET，获取共享内存）来分配一个共享内存块。该函数的第一个参数是一个用来标识共享内存块的键值。彼此无关的进程可以通过指定同一个键以获取对同一个共享内存块的访问。不幸的是，其它程序也可能挑选了同样的特定值作为自己分配共享内存的键值，从而产生冲突。用特殊常量 **IPC_PRIVATE** 作为键值可以保证系统建立一个全新的共享内存块。

该函数的第二个参数指定了所申请的内存块的大小。因为这些内存块是以页面为单位进行分配的，实际分配的内存块大小将被扩大到页面大小的整数倍。

第三个参数是一组标志，通过特定常量的按位或操作来 **`shmget`**。这些特定常量包括：

- **IPC_CREAT**：这个标志表示应创建一个新的共享内存块。通过指定这个标志，我们可以创建一个具有指定键值的新共享内存块。
- **IPC_EXCL**：这个标志只能与 **IPC_CREAT** 同时使用。当指定这个标志的时候，如果已有一个具有这个键值的共享内存块存在，则 **`shmget`** 会调用失败。也就是说，这个标志将使线程获得一个“独有”的共享内存块。如果没有指定这个标志而系统中存在一个具有相通键值的共享内存块，**`shmget`** 会返回这个已经建立的共享内存块，而不是重新创建一个。
- **模式标志（Mode flags）**：这个值由 9 个位组成，分别表示属主、属组和其它用户

对该内存块的访问权限。其中表示执行权限的位将被忽略。指明访问权限的一个简单办法是利用**<sys/stat.h>**中指定，并且在手册页第二节**stat**条目中说明了的常量指定[1]。例如，**S_IRUSR**和**S_IWUSR**分别指定了该内存块属主的读写权限，而 **S_IROTH**和**S_IWOTH**则指定了其它用户的读写权限。

下面例子中 **shmget** 函数创建了一个新的共享内存块（当 **shm_key** 已被占用时则获取对一个已经存在共享内存块的访问），且只有属主对该内存块具有读写权限，其它用户不可读写。

```
int segment_id = shmget (shm_key, getpagesize (),
                         IPC_CREAT | S_IRUSR| S_IWUSR );
```

如果调用成功，**shmget** 将返回一个共享内存标识符。如果该共享内存块已经存在，系统会检查访问权限，同时会检查该内存块是否被标记为等待摧毁状态。

5.1.4 绑定和脱离

要让一个进程获取对一块共享内存的访问，这个进程必须先调用 **shmat**（SHared Memory Attach，绑定到共享内存）。将 **shmget** 返回的共享内存标识符 **SHMID** 传递给这个函数作为第一个参数。该函数的第二个参数是一个指针，指向你希望用于映射该共享内存块的进程内存地址；如果你指定 NULL 则 Linux 会自动选择一个合适的地址用于映射。第三个参数是一个标志位，包含了以下选项：

- **SHM_RND** 表示第二个参数指定的地址应被向下靠拢到内存页面大小的整数倍。如果你不指定这个标志，你将不得不在调用 **shmat** 的时候手工将共享内存块的大小按页面大小对齐。
- **SHM_RDONLY** 表示这个内存块将仅允许读取操作而禁止写入。

如果这个函数调用成功则会返回绑定的共享内存块对应的地址。通过 **fork** 函数创建的子进程同时继承这些共享内存块；如果需要，它们可以主动脱离这些共享内存块。

当一个进程不再使用一个共享内存块的时候应通过调用 **shmdt**（SHared Memory DeTach，脱离共享内存块）函数与该共享内存块脱离。将由 **shmat** 函数返回的地址传递给这个函数。如果当释放这个内存块的进程是最后一个使用该内存块的进程，则这个内存块将被删除。对 **exit** 或任何 **exec** 族函数的调用都会自动使进程脱离共享内存块。

5.1.5 控制和释放共享内存块

调用 **shmctl**（"SHared Memory ConTroL"，控制共享内存）函数会返回一个共享内存块的相关信息。同时 **shmctl** 允许程序修改这些信息。该函数的第一个参数是一个共享内存块标识。

要获取一个共享内存块的相关信息，则为该函数传递 **IPC_STAT** 作为第二个参数，同时传递一个指向一个 **struct shmid_ds** 对象的指针作为第三个参数。

要删除一个共享内存块，则应将 **IPC_RMID** 作为第二个参数，而将 NULLL 作为第三个参数。当最后一个绑定该共享内存块的进程与其脱离时，该共享内存块将被删除。

你应当在结束使用每个共享内存块的时候都使用 **shmctl** 进行释放，以防止超过系统所允许的共享内存块的总数限制。调用 **exit** 和 **exec** 会使进程脱离共享内存块，但不会删除这个内存块。

[1] 这些权限位与用于控制文件权限的相同。10.3 节“文件系统权限”对它们做了更多的介绍。

要查看其它有关共享内存块的操作的描述，请参考 **shmctl** 函数的手册页。

5.1.6 示例程序

代码列表 5.1 中的程序展示了共享内存块的使用。

代码列表 5.1 (shm.c) 尝试共享内存

```
#include <stdoi.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* 分配一个共享内存块 */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR );
  /* 绑定到共享内存块 */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n",
shared_memory);
  /* 确定共享内存的大小 */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* 在共享内存中写入一个字符串 */
  sprintf (shared_memory, "Hello, world.");
  /* 脱离该共享内存块 */
  shmdt (shared_memory);

  /* 重新绑定该内存块 */
  shared_memory = (char*) shmat (segment_id, (void*) 0x500000, 0);
  printf ("shared memory reattached at address %p\n",
shared_memory);
  /* 输出共享内存中的字符串 */
  printf ("%s\n", shared_memory);
  /* 脱离该共享内存块 */
  shmdt (shared_memory);
```

```
  /* 释放这个共享内存块 */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

5.1.7 调试

使用 **ipcs** 命令可用于查看系统中包括共享内存在内的进程间通信机制的信息。指定**-m** 参数以获取有关共享内存的信息。例如，以下的示例表示有一个编号为 **1627649** 的共享内存块正在使用中：

```
% ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 1627649    user       640        25600      0
```

如果这个共享内存块在程序结束后没有被删除而是被错误地保留下来，你可以用 **ipcrm** 命令删除它。

```
% ipcrm shm 1627649
```

5.1.8 优点和缺点

共享内存块提供了在任意数量的进程之间进行高效双向通信的机制。每个使用者都可以读取写入数据，但是所有程序之间必须达成并遵守一定的协议，以防止诸如在读取信息之前覆写内存空间等竞争状态的出现。不幸的是，Linux 无法严格保证提供对共享内存块的独占访问，甚至是在你通过使用 **IPC_PRIVATE** 创建新的共享内存块的时候也不能保证访问的独占性。

同时，多个使用共享内存块的进程之间必须协调使用同一个键值。

5.2 进程信号量

前一节中我们提到过，当访问共享内存的时候，进程之间必须相互协调以避免竞争状态的出现。正如我们在第四章“线程”中 4.4.5 节“线程信号量”里说过的，信号量是一个可用于同步多线程环境的计数器。Linux 还提供了一个另外一个用于进程间同步的信号量实现（通常它被称为进程信号量，有时也被称为 System V 信号量）。进程信号量的分配、使用和释放方法都与共享内存块相似。尽管单个信号量足以解决大多数问题，进程信号量是按组（*set*）分配的。本节中，我们将展示如何利用各种 Linux 提供的各种系统调用来实现一个具有两种状态的信号量。

5.2.1 分配和释放

与用于分配、释放共享内存的 **shmget** 和 **shmctl** 类似，系统调用 **semget** 和 **semctl** 负责分配、释放信号量。调用 **semget** 函数并传递如下参数：一个用于标识信号量组的键值，该组中包含的信号量数量和与 **shmget** 所需的相同的权限位标识。该函数返回的是信号量组的标识符。你可以通过指定正确的键值来获取一个已经存在的信号量的标识符；这种情况下，传递的信号量组的容量可以为 0。

信号量会一直保存在系统中，甚至所有使用它们的进程都退出后也不会自动被销毁。最后一个使用信号量的进程必须明确地删除所使用的信号量组，来确保系统中不会有太多闲置的信号量组，从而导致无法创建新的信号量组。可以通过调用 **semctl** 来删除信号量组。调用时的四个参数分别为信号量组的标识符，组中包含的信号量数量、常量 **IPC_RMID** 和一个 **union semun** 类型的任意值（被忽略）。调用进程的有效用户 id 必须与分配这个信号量组的用户 id 相同（或者调用进程为 root 权限亦可）。与共享内存不同，删除一个信号量组会导致 Linux 立即释放资源。

列表 5.2 展示了用于分配和释放一个二元信号量的函数。

代码列表 5.2 （sem_all_deall.c）分配和释放二元信号量

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

/* 我们必须自己定义 semun 联合类型。 */

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* 获取一个二元信号量的标识符。如果需要则创建这个信号量 */
int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}

/* 释放二元信号量。所有用户必须已经结束使用这个信号量。如果失败，返回 -1 */
int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

5.2.2 初始化信号量

分配与初始化信号量是两个相互独立的操作。以 0 为第二参数，以 **SETALL** 为第三个参数调用 **semctl** 可以对一个信号量组进行初始化。第四个参数是一个 **semun** 对象，且它的 **array** 字段指向一个 **unsigned short** 数组。数组中的每个值均用于初始化该组中的一个信号量。

列表 5.3 展示了初始化一个二元信号量的函数

代码列表 5.3 （sem_init.c） 初始化一个二元信号量

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* 我们必须自己定义 union semun。*/

union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};

/* 将一个二元信号量初始化为 1。*/

int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```

5.2.3 等待和投递操作

每个信号量都具有一个非负的值，且信号量支持等待和投递操作。系统调用 **semop** 实现了这两个操作。它的第一个参数是信号量的标识符，第二个参数是一个包含 **struct sembuf** 类型元素的数组；这些元素指明了你希望执行的操作。第三个参数是这个数组的长度。

结构体 **sembuf** 中包含如下字段：

- **sem_num** 将要执行操作的信号量组中包含的信号量数量
- **sem_op** 是一个指定了操作类型的整数

- 如果 **sem_op** 是一个正整数，则这个值会立刻被加到信号量的值上
  如果 **sem_op** 为负，则将从信号量值中减去它的绝对值。如果这将使信号量的值小于零，则这个操作会导致进程阻塞，直到信号量的值至少等于操作值的绝对值（由其它进程增加它的值）。
- 如果 **sem_op** 为 0，这个操作会导致进程阻塞，直到信号量的值为零才恢复。
- **sem_flg** 是一个符号位。指定 **IPC_NOWAIT** 以防止操作阻塞；如果该操作本应阻塞，则 **semop** 调用会失败。如果为 **sem_flg** 指定 **SEM_UNDO**，Linux 会在进程退出的时候自动撤销该次操作。

列表 5.4 展示了二元信号量的等待和投递操作。

代码列表 5.4 （sem_pv.c）二元信号量的等待和投递操作

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* 等待一个二元信号量。阻塞直到信号量的值为正，然后将其减 1 */

int binary_semaphore_wait (int semid)
{
  struct sembuf operations[1];
  /* 使用（且仅使用）第一个信号量 */
  operations[0].sem_num = 0;
  /* 减一 */
  operations[0].sem_op = -1;
  /* 允许撤销操作 */
  operations[0].sem_flg = SEM_UNDO;

  return semop (semid, operations, 1);
}

/* 对一个二元信号量执行投递操作：将其值加一。
   这个操作会立即返回。*/

int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* 使用（且仅使用）第一个信号量 */
  operations[0].sem_num = 0;
  /* 加一 */
  operations[0].sem_op = 1;
  /* 允许撤销操作 */
  operations[0].sem_flg = SEM_UNDO;
```

```
  return semop (semid, operations, 1);
}
```

指定 **SEM_UNDO** 标志解决当出现一个进程仍然持有信号量资源时被终止这种特殊情况时可能出现的资源泄漏问题。当一个进程被有意识或者无意识地结束的时候，信号量的值会被调整到“撤销”了所有该进程执行过的操作后的状态。例如，如果一个进程在被杀死之前减小了一个信号量的值，则该信号量的值会增长。

5.2.4 调试信号量

命令 **ipcs -s** 可以显示系统中现有的信号量组的相关信息。而 **ipcrm sem** 命令可以从命令行删除一个信号量组。例如，要删除标识符为 5790517 的信号量组则应运行以下命令：

```
% ipcrm sem 5790517
```

5.3 映射内存

映射内存提供了一种使多个进程通过一个共享文件进行通信的机制。尽管可以将映射内存想象为一个有名字的共享内存，你始终应当记住两者之间有技术层面的区别。映射内存既可以用于进程间通信，也可以作为一种访问文件内容的简单方法。

映射内存在一个文件和一块进程地址空间之间建立了联系。Linux 将文件分割成内存分页大小的块并复制到虚拟内存中，因此进程可以在自己的地址空间中直接访问文件内容。这样，进程就可以以读取普通内存空间的方法来访问文件的内容，也可以通过写入内存地址来修改文件的内容。这是一种方便的访问文件的方法。

你可以将映射内存想象成这样的操作：分配一个足够容纳整个文件内容的缓存，将全部文件内容读入缓存，并且（当缓存内容被修改过后）最后将缓存写回文件。Linux 替你完成文件读写的操作。

除了用于进程间通信，还有其它一些情况会使用映射内存。其中一些用途在 5.3.5 节“mmap 的其它用途”中进行了讨论。

5.3.1 映射一个普通文件

要将一个普通文件映射到内存空间，应使用 **mmap**（映射内存，“Memory MAPped”，读作“em-map”）。函数 **mmap** 的第一个参数指明了你希望 Linux 将文件映射在进程地址空间中的位置；传递 NULL 指针允许 Linux 系统自动选择起始地址。第二个参数是映射内存块的长度，以字节为单位。第三个参数指定了对被映射内存区域的保护，由 **PROT_READ**、**PROT_WRITE** 和 **PROT_EXEC** 三个标志位按位与操作得到。三个值分别标识读、写和执行权限。第四个参数是一个用于指明额外选项的标志值。第五个参数应传递一个已经打开的、指向被映射文件的句柄。最后一个参数指明了文件中被映射区域相对于文件开始位置的偏移量。通过选择适当的开始位置和偏移量，你可以选择将文件的全部内容或某个特定部分映射到内存中。

标志值可以由以下常量进行按位或操作进行组合得到：

- **MAP_FIXED**——如果你指定这个标志，Linux 会强制使用你提供的地址进行映射，而不只是将该地址作为一个对映射地址的参考进行选择。该地址必须按内存分页边界对齐。
- **MAP_PRIVATE**——对映射区域内存的写操作不会直接导致对被绑定文件的修改，

而是修改该进程持有的一份该文件的私有副本。其它进程不会发现这些写操作。这个模式不能与 **MAP_SHARED** 同时使用。

- **MAP_SHARED**——对内存的写入操作会立刻反应在被映射的文件中，而不会被系统缓冲。将映射内存作为一种 **IPC** 手段时应使用这个标志。这个模式不能与 **MAP_PRIVATE** 同时使用。

如果调用成功，**mmap** 会返回一个指向被映射内存区域的起点的指针。如果调用失败则返回常量 **MAP_FAILED**。

当你不再使用一块映射内存的时候应调用 **munmap** 进行释放。将被映射内存区域的开始地址和内存块的长度作为参数调用这个函数。Linux 会在进程结束的时候自动释放进程中映射的内存区域。

5.3.2 示例程序

让我们看两个利用映射内存对文件进行读写的程序。列表 5.5 中的第一个程序会产生一个随机数并写入一个映射到内存的文件中。列表 5.6 中的第二个程序则会从文件中读取并输出这个值，然后将该值的两倍写回到文件中。两个程序均接受一个指明被映射文件的参数。

代码列表 5.5 （mmap-write.c）将一个随即数写入内存映射文件

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* 在从 low 到 high 的范围中返回一个平均分布的随机数 */

int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand() / (RAND_MAX + 1.0));
}

int man (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

  /* 为随机数发生器提供种子 */
  srand (time (NULL));

  /* 准备一个文件使之长度足以容纳一个无符号整数 */
```

```
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* 创建映射内存 */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd,
0);
  close (fd);
  /* 将一个随机整数写入映射的内存区域 */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* 释放内存块（不是必需，因为程序即将退出而映射内存将被自动释放） */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

上面的 **mmap-write** 程序打开了一个指定的文件（如果不存在则创建它）。传递给 **open** 的第二个参数表明以读写模式创建文件。（译者注：原文为第三个参数，疑为笔误。）因为我们不知道文件的长度，我们利用 **lseek** 确保文件具有足够容纳一个整数的长度，然后将游标移动到文件的开始位置。

程序在将文件映射到内存之后随即关闭了文件描述符，因为我们不再需要通过这个描述符操作文件。随后程序将一个随机整数写入映射内存，从而也写入了文件内容本身；之后程序取消了内存映射。对 **munmap** 的调用不是必须的，因为 Linux 会在程序结束的时候自动取消全部内存映射。

代码列表 5.6 （mmap-read.c）从文件映射内存中读取一个整数，然后将其倍增

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* 打开文件 */
```

```
    fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
    /* 创建映射内存 */
    file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    close (fd);

    /* 读取整数，输出，然后将其倍增 */
    sscanf (file_memory, "%d", %integer);
    printf ("value: %d\n", integer);
    sprintf ((char*) file_memory, "%d\n", 2 * integer);
    /* 释放内存（非必须，因为程序就此结束）*/
    munmap (file_memory, FILE_LENGTH);

    return 0;
  }
```

上面的 **mmap-read** 程序从文件中读取一个整数值，将其倍增并写回到文件中。首先它以读写模式打开文件。因为我们确信文件足以容纳一个整数，我们不必像前面的程序那样使用 **lseek**。程序从内存中用 **sscanf** 读取这个值，然后用 **sprintf** 将值写回内存中。

这里是某次运行这个程序的结果。它将文件/tmp/integer-file 映射到内存。

```
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer/file
value: 42
% cat /tmp/integer-file
84
```

我们可以看到，程序并没有调用 **write** 就将数字写入了文件，同样也没有用 **read** 就将数字读回。注意，仅出于演示的考虑，我们将数字转换为字符串进行读写（通过使用 **sprintf** 和 **sscanf**）——一个内存映射文件的内容并不要求为文本格式。你可以在其中存取任意二进制数据。

5.3.3 对文件的共享访问

不同进程可以将同一个文件映射到内存中，并借此进行通信。通过指定 **MAP_SHARED** 标志，所有对映射内存的写操作都会直接作用于底层文件并且对其它进程可见。如果不指定这个标志，Linux 可能在将修改写入文件之前进行缓存。

除了使用 **MAP_SHARED** 标志，你也可以通过调用 **msync** 强制要求 Linux 将缓存的内容写入文件。它的前两个参数与 **munmap** 相同，用于指明一个映射内存块。第三个参数可以接受如下标志位：

- **MS_ASYNC**——计划一次更新，但是这次更新未必在调用返回之前完成。
- **MS_SYNC**——立刻执行更新；**msync** 调用会导致进程阻塞直到更新完成。
  MS_SYNC 和 **MS_ASYNC** 不能同时使用。
- **MS_INVALIDATE**——其它所有进程对这个文件的映射都会失效，因此它们可以

看到被修改过的值。

例如，要更新一块从 **mem_addr** 开始的、长度为 **mem_length** 的共享内存块需要使用如下调用：

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

与使用共享内存一样，使用文件映射内存的程序之间必须遵循一定的协议以避免竞争状态的发生。例如，可以通过一个信号量协调多个进程一块内存映射文件的并发访问。除此之外你还可以使用第八章 8.3 节“fcntl：锁定与其它文件操作”中介绍的方法对文件进行读写锁定。

5.3.4 私有映射

在 **mmap** 中指定 **MAP_PRIVATE** 可以创建一个写时复制（*copy-on-write*）区域。所有对映射区域内存内容的修改都仅反映在当前程序的地址空间中；其它进程即使映射了同一个文件也不会看到这些变化。与普通情况下直接写入所有进程共享的页面中的行为不同，指定 **MAP_PRIVATE** 进行映射的进程只将改变写入一份私有副本中。该进程随后执行的所有读写操作都针对这个副本进行。

5.3.5 *mmap* 的其它用途。

系统调用 **mmap** 还可以用于除进程间通信之外的其它用途。一个常见的用途就是取代 **read** 和 **write**。例如，要读取一个文件的内容，程序可以不再显式地读取文件并复制到内存中，而是将文件映射到地址空间然后通过内存读写操作来操作文件内容。对于一些程序而言这样更方便，也可能具有更高的效率。

许多程序都使用了这样一个非常强大的高级技巧：将某种数据结构（例如各种 struct 结构体的实例）直接建立在映射内存区域中。在下次调用过程中，程序将这个文件映射回内存中，此时这些数据结构都会恢复到之前的状态。不过需要注意的是，这些数据结构中的指针都会失效，除非这些指针都指向这个内存区域内部并且这个内存区域被特意映射到与之前一次映射位置完全相同的地址。

另一个相当有用的技巧是将设备文件**/dev/zero** 映射到内存中。这个文件，将在第六章“设备” 6.5.2 节“/dev/zero”中介绍到的，将自己表现为一个无限长且内容全部为 0 字节的文件。对**/dev/zero** 执行的写入操作将被丢弃，因此由它映射的内存区域可以用作任何用途。自定义的内存分配过程经常通过映射/dev/zero 以获取整块经过初始化的内存。

5.4 管道

管道是一个允许单向信息传递的通信设备。从管道“写入端”写入的数据可以从“读取端”读回。管道是一个串行设备；从管道中读取的数据总保持它们被写入时的顺序。一般来说，管道通常用于一个进程中两个线程之间的通信，或用于父子进程之间的通信。

在 shell 中，| 符号用于创建一个管道。例如，下面的程序会使 shell 创建两个子进程，一个运行 **ls** 而一个运行 **less**：

```
% ls | less
```

Shell 同时还会创建一个管道，将运行 **ls** 的子进程的标准输出连接到运行 **less** 的子进程的标准输入。由 **ls** 输出的文件名将被按照与发送到终端时完全相同的顺序发送到 **less** 的标准输入。

管道的数据容量是有限的。如果写入的进程写入数据的速度比读取进程消耗数据的速度

更快，且管道无法容纳更多数据的时候，写入端的进程将被阻塞，直到管道中出现更多的空间为止。换言之，管道可以自动同步两个进程。

5.4.1 创建管道

要创建一个管道，请调用 **pipe** 命令。提供一个包含两个 **int** 值的数组作为参数。**Pipe** 命令会将读取端文件描述符保存在数组的第 0 个元素而将写入端文件描述符保存在第 1 个元素中。举个例子，考虑如下代码：

```
int pipe_fds[2];
int read_fd;
int write_fd;

pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

对文件描述符 **write_fd** 写入的数据可以从 **read_fd** 中读回。

5.4.2 父子进程之间的通信

通过调用 **pipe** 得到的文件描述符只在调用进程及子进程中有效。一个进程中的文件描述符不能传递给另一个无关进程；不过，当这个进程调用 **fork** 的时候，文件描述符将复制给新创建的子进程。因此，管道只能用于连接相关的进程。

列表 5.7 中的程序中，**fork** 产生了一个子进程。子进程继承了指向管道的文件描述符。父进程向管道写入一个字符串，然后子进程将字符串读出。实例程序将文件描述符利用 **fdopen** 函数转换为 **FILE *** 流。因为我们使用文件流而不是文件描述符，我们可以使用包括 **printf** 和 **scanf** 在内的标准 C 库提供的高级 I/O 函数。

代码列表 5.7 （pipe.c）通过管道与子进程通信

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

/* 将 COUNT 份 MESSAGE 的副本写入 STREAM，每次写入之后暂停 1 秒钟 */

void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
    /* 将消息写入流，然后立刻发送 */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* 休息，休息一会儿 */
    sleep (1);
  }
```

```
}

/* 从流中读取尽可能多的随机字符串 */

void reader (FILE* stream)
{
  char buffer[1024];
  /* 从流中读取直到流结束。 fgets 会不断读取直到遇见换行或文件结束符。 */
  while (!feof (stream)
        && !ferror (stream)
        && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}

int main ()
{
  int fds[2];
  pid_t pid;

  /* 创建一个管道。代表管道两端的文件描述符将被放置在 fds 中。*/
  pipe (fds);
  /* 创建子进程。*/
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
    /* 这里是子进程。关闭我们得到的写入端文件描述符副本。*/
    close (fds[1]);
    /* 将读取端文件描述符转换为一个 FILE 对象然后从中读取消息 */
    stream = fdopen (fds[0], "r");
    reader (stream);
    close (fds[0]);
  }
  clse {
    /* 这是父进程。*/
    FILE* stream;
    /* 关闭我们的读取端文件描述符副本。*/
    close (fds[0]);
    /* 将写入端文件描述符转换为一个 FILE 对象然后写入数据。*/
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }

  return 0;
```

```
}
```

在 main 函数开始的时候，**fds** 被声明为一个包含两个整型变量的数组。对 **pipe** 的调用创建了一个管道，并将读写两个文件描述符存放在这个数组中。程序随后创建一个子进程。在关闭了管道的读取端之后，父进程开始向管道写入字符串。而在关闭了管道的写入端之后，子进程开始从管道读取字符串。

注意，在 **writer** 函数中，父进程在每次写入操作之后通过调用 **fflush** 刷新管道内容。否则，字符串可能不会立刻被通过管道发送出去。

当你调用 **ls | less** 这个命令的时候会出现两次 **fork** 过程：一次为 **ls** 创建子进程，一次为 **less** 创建子进程。两个进程都继承了这些指向管道的文件描述符，因此它们可以通过管道进行通信。如果希望不相关的进程互相通信，应该用 **FIFO** 代替管道。**FIFO** 将在 5.4.5 节“**FIFO**”中进行介绍。

5.4.3 重定向标准输入、输出和错误流

你可能经常希望创建一个子进程，并将一个管道的一端设置为它的标准输入或输出。利用 **dup2** 系统调用你可以使一个文件描述符等效于另外一个。例如，下面的命令可以将一个进程的标准输入重定向到文件描述符 **fd**：

```
dup2 (fd, STDIN_FILENO);
```

符号常量 **STDIN_FILENO** 代表指向标准输入的文件描述符。它的值为 0。这个函数会关闭标准输入，然后将它作为 **fd** 的副本重新打开，从而使两个标识符可以互换使用。

相互等效的文件描述符之间共享文件访问位置和相同的一组文件状态标志。因此，从 fd 中读取的字符不会再次从标志输入中被读取。

列表 5.8 中的程序利用**dup2** 系统调用将一个管道的输出发送到**sort**命令。[2] 当创建了一个管道之后，程序生成了子进程。父进程向管道中写入一些字符串，而子进程利用**dup2** 将管道的读取端描述符复制到自己的标准输入，然后执行**sort**程序。

代码列表 5.8 （dup2.c）用 dup2 重定向管道输出

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

int main ()
{
  int fds[2];
  pid_t pid;

  /* 创建管道。标识管道两端的文件描述符会被放置在 fds 中。*/
  pipe (fds);
  /* 产生子进程。*/
```

[2] sort 程序从标志输入按行读取文本信息，按照字母序进行排列，然后输出到标准输出。

```
  pid = fork ();
  if (pid == (pid_t) 0) {
    /* 这里是子进程。关闭我们的写入端描述符。*/
    close (fds[1]);
    /* 将读取端连接到标准输入*/
    dup2 (fds[0], STDIN_FILENO);
    /* 用 sort 替换子进程。*/
    execlp ("sort", "sort", 0);
  }
  else {
    /* 这是父进程。*/
    FILE* stream;
    /* 关闭我们的读取端描述符。*/
    close (fds[0]);
    /* 将写入端描述符转换为一个 FILE 对象，然后将信息写入。*/
    stream = fdopen (fds[1], "w");
     fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
    /* 等待子进程结束。*/
    waitpid (pid, NULL, 0);
  }

  return 0;
}
```

5.4.4 popen 和 pclose

管道的一个常见用途是与一个在子进程中运行的程序发送和接受数据。而 **popen** 和 **pclose** 函数简化了这个过程。它取代了对 **pipe**、**fork**、**dup2**、**exec** 和 **fdopen** 的一系列调用。

下面将使用了 **popen** 和 **pclose** 的列表 5.9 与之前一个例子（列表 5.8）进行比较。

代码列表 5.9 （popen.c）使用 popen 的示例

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

int main ()
{
```

```
    FILE* stream = popen ("sort", "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    return pclose (stream);
  }
```

通过调用 **popen** 取代 **pipe**、**fork**、**dup2** 和 **execlp** 等，一个子进程被创建以执行了 **sort** 命令，。第二个参数，”w”，指示出这个进程希望对子进程输出信息。**Popen** 的返回值是管道的一端；另外一端连接到了子进程的标准输入。在数据输出结束后，**pclose** 关闭了子进程的流，等待子进程结束，然后将子进程的返回值作为函数的返回值返回给调用进程。

传递给 **popen** 的第一个参数会作为一条 shell 命令在一个运行/bin/sh 的子进程中执行。Shell 会照常搜索 PATH 环境变量以寻找应执行的程序。如果第二个参数是"r"，函数会返回子进程的标准输出流以便父进程读取子进程的输出。如果第二个参数是"w"，函数返回子进程的标准输入流一边父进程发送数据。如果出现错误，**popen** 返回空指针。

调用 **pclose** 会关闭一个由 **popen** 返回的流。在关闭指定的流之后，**pclose** 将等待子进程退出。

5.4.5 FIFO

*先入先出（first-in, first-out, **FIFO**）*文件是一个在文件系统中有一个名字的管道。任何进程均可以打开或关闭 **FIFO**；通过 **FIFO** 连接的进程不需要是彼此关联的。**FIFO** 也被称为命名管道。

可以用 **mkfifo** 命令创建 **FIFO**；通过命令行参数指定 **FIFO** 的路径。例如，运行这个命令将在/tmp/fifo 创建一个 **FIFO**：

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-  1 samuel users  0 Jan 16 14:04 /tmp/fifo
```

ls 输出的第一个字符是 **p**，表示这个文件实际是一个 **FIFO**（命名管道）。在一个窗口中这样从 **FIFO** 中读取内容：

```
% cat < /tmp/fifo
```

在第二个窗口中这样向 **FIFO** 中写入内容：

```
% cat > /tmp/fifo
```

然后输入几行文字。每次你按下回车后，当前一行文字都会经由 **FIFO** 发送到第一个窗口。通过在第二个窗口中按 **Ctrl+D** 关闭这个 **FIFO**。运行下面的命令删除这个 **FIFO**：

```
% rm /tmp/fifo
```

创建 FIFO

通过编程方法创建一个 **FIFO** 需要调用 **mkfifo** 函数。第一个参数是要创建 **FIFO** 的路径，第二个参数是被创建的 **FIFO** 的属主、属组和其它用户权限。关于权限，第十章“安全”的 10.3 节“文件系统权限”中进行了介绍。因为管道必然有程序读取信息、有程序写入信息，因此权限中必须包含读写两种权限。如果无法成功创建管道（如当同名文件已经存在的时候），**mkfifo** 返回-1。当你调用 **mkfifo** 的时候需要包含 **<sys/types.h>** 和

<sys/stat.h>。

访问 FIFO

访问 **FIFO** 与访问普通文件完全相同。要通过 **FIFO** 通信，必须有一个程序打开这个 **FIFO** 写入信息，而另一个程序打开这个 **FIFO** 读取信息。底层 I/O 函数（**open**、**write**、**read**、**close** 等，列举在附录 B“底层 I/O”中）或 C 库 I/O 函数（**fopen**、**fprintf**、**fscanf**、**fclose** 等）均适用于访问 **FIFO**。

例如，要利用底层 I/O 将一块缓存区的数据写入 **FIFO** 可以这样完成：

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

利用 C 库 I/O 从 **FIFO** 读取一个字符串可以这样做：

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer");
fclose (fifo);
```

FIFO 可以有多个读取进程和多个写入进程。来自每个写入进程的数据当到达 **PIPE_BUF**（Linux 系统中为 4KB）的时候会自动写入 **FIFO**。并发写入可能导致数据块的互相穿插。同步读取也会出现相似的问题。

与 Windows 命名管道的区别

Win32 操作系统的管道与Linux系统中提供的相当类似。（相关技术细节可以从Win32 库文档中获得印证。）主要的区别在于，Win32 系统上的命名管道的功能更接近套接字。Win32 命名管道可以用于连接处于同一个网络中不同主机上的不同进程之间相互通信。Linux系统中，套接字被用于这种情况。同时，Win32 保证同一个命名管道上的多个读——写连接不出现数据交叉情况，而且管道可以用于双向交流。[3]

5.5 套接字

套接字是一个双向通信设备，可用于同一台主机上不同进程之间的通信，也可用于沟通位于不同主机的进程。套接字是本章中介绍的所有进程间通信方法中唯一允许跨主机通信的方式。**Internet** 程序，如 **Telnet**、**rlogin**、**FTP**、**talk** 和万维网都是基于套接字的。

例如，你可以用一个**Telnet**程序从一台网页服务器获取一个万维网网页，因为它们都使用套接字作为网络通信方式[4]。可以通过执行`telnet www.codesourcery.com 80`连接到位于`www.codesourcery.com`主机的网页服务器。魔数 80 指明了连接的目标进程是运行于`www.codesourcery.com`的网页服务器而不是其它什么进程。成功建立连接后，试着输入 **GET /**。这会通过套接字发送一条消息给网页服务器，而相应的回答则是服务器将主页的HTML代码传回然后关闭连接——例如：

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
```

[3] 注意只有 `Windows NT` 可以建立命名管道；`Windows 9x` 程序只能建立客户连接。

[4] 通常 `telnet` 程序用于连接到 `Telnet` 服务器执行远程登陆。但你也可以使用 `telnet` 程序连接到其它类型的服务器然后直接向它发送命令。

```
Escape character is '^]'.
GET /
<html>
<head>
  <meta          http-equiv="Content-Type"          content="text/html;
charset=iso-8859-1">
...
```

5.5.1 套接字概念

当你创建一个套接字的时候你需要指定三个参数：通信类型，命名空间和协议。

通信类型决定了套接字如何对待被传输的数据，同时指定了参与传输过程的进程数量。当数据通过套接字发送的时候会被分割成段落，这些段落分别被称为一个包（*packet*）。通信类型决定了处理这些包的方式，以及为这些包定位目标地址的方式。

- 连接式（*Connection style*）通信保证所有包都以发出时的顺序被送达。如果由于网络的关系出现包丢失或顺序错乱，接收端会自动要求发送端重新传输。
  连接类型的套接字可想象成电话：发送和接收端的地址在开始时连接被建立的时候都被确定下来。
- 数据报式（*Datagram style*）的通信不确保信息被送到，也不保证送到的顺序。数据可能由于网络问题或其它情况在传输过程中丢失或重新排序。每个数据包都必须标记它的目标地址，而且不会被保证送到。系统仅保证“尽力”做到，因此数据包可能消失，或以与发出时不同的顺序被送达。
  数据报类型的通信更类似邮政信件。发送者为每个单独信息标记收信人地址。

套接字的命名空间指明了套接字地址的书写方式。套接字地址用于标识一个套接字连接的一个端点。例如，在“本地命名空间”中的套接字地址是一个普通文件。而在“Internet 命名空间”中套接字地址由网络上的一台主机的 Internet 地址（也被称为 Internet 协议地址或 **IP** 地址）和端口号组成。端口号用于区分同一台主机上的不同套接字。

协议指明了数据传输的方式。常见的协议有如下几种：**TCP/IP**，Internet 上使用的最主要的通信协议；**AppleTalk** 网络协议；UNIX 本地通信协议等。通信类型、命名空间和协议三者的各种组合中，只有部分是有效的。

5.5.2 系统调用

套接字比之前介绍的任何一种进程间通信方法都更具弹性。这里列举了与套接字相关的系统调用：

`socket`——创建一个套接字
`close`——销毁一个套接字
`connect`——在两个套接字之间创建连接
`bind`——将一个服务器套接字绑定一个地址
`listen`——设置一个套接字为接受连接状态
`accept`——接受一个连接请求并为新建立的连接创建一个新的套接字

套接字通常被表示为文件描述符。

创建和销毁套接字

Socket 和 **close** 函数分别用于创建和销毁套接字。当你创建一个套接字的时候，需指明三种选项：命名空间，通信类型和协议。利用 **PF_**开头（标识协议族，*protocol families*）的常量指明命名空间类型。例如，**PF_LOCAL** 或 **PF_UNIX** 用于标识本地命名空间，而 **PF_INET** 表示 Internet 命名空间。用以 **SOCK_**开头的常量指明通信类型。**SOCK_STREAM** 表示连接类型的套接字，而 **SOCK_DGRAM** 表示数据报类型的套接字。

第三个参数，协议，指明了发送和接收数据的底层机制。每个协议仅对一种命名空间和通信类型的组合有效。因为通常来说，对于某种组合都有一个最合适的协议，为这个参数指定 0 通常是最合适的选择。如果 **socket** 调用成功则会返回一个表示这个套接字的文件描述符。与操作普通文件描述符一样，你可以通过 **read** 和 **write** 对这个套接字进行读写。当你不再需要它的时候，应调用 **close** 删除这个套接字。

调用 *connect*

要在两个套接字之间建立一个连接，客户端需指定要连接到的服务器套接字地址，然后调用 **connect**。客户端指的是初始化连接的进程，而服务端指的是等待连接的进程。客户端调用 **connect** 以在本地套接字和第二个参数指明的服务端套接字之间初始化一个连接。第三个参数是第二个参数中传递的标识地址的结构的长度，以字节计。套接字地址格式随套接字命名空间的不同而不同。

发送信息

所有用于读写文件描述符的技巧均可用于读写套接字。关于 Linux 底层 I/O 函数及一些相关使用问题的讨论请参考附录 B。而专门用于操作套接字的 **send** 函数提供了 **write** 之外的另一种选择，它提供了 **write** 所不具有的一些特殊选项；参考 **send** 的手册页以获取更多信息。

5.5.3 服务器

服务器的生命周期可以这样描述：创建一个连接类型的套接字，绑定一个地址，调用 **listen** 将套接字置为监听状态，调用 **accept** 接受连接，最后关闭套接字。数据不是直接经由服务套接字被读写的；每次当程序接受一个连接的时候，Linux 会单独创建一个套接字用于在这个连接中传输数据。在本节中，我们将介绍 **bind**、**listen** 和 **accept**。

要想让客户端找到，必须用 **bind** 将一个地址绑定到服务端套接字。**Bind** 函数的第一个参数是套接字文件描述符。第二个参数是一个指针，它指向表示套接字地址的结构。它的格式取决于地址族。第三个参数是地址结构的长度，以字节计。将一个地址绑定到一个连接类型的套接字之后，必须通过调用 **listen** 将这个套接字标识为服务端。**Listen** 的第一个参数是套接字文件描述符。第二个参数指明最多可以有多少个套接字处于排队状态。当等待连接的套接字超过这个限度的时候，新的连接将被拒绝。它不是限制一个服务器可以接受的连接总数；它限制的是被接受之前允许尝试连接服务器的客户端总数。

服务端通过调用 **accept** 接受一个客户端连接。第一个参数是套接字文件描述符。第二个参数是一个指向套接字地址结构的指针；接受连接后，客户端地址将被写入这个指针指向的结构中。第三个参数是套接字地址结构体的长度，以字节计。服务端可以通过客户端地址确定是否希望与客户端通信。调用 **accept** 会创建一个用于与客户端通信的新套接字，并返回对应的文件描述符。原先的服务端套接字继续保持监听连接的状态。用 **recv** 函数可以从套接字中读取信息而不将这些信息从输入序列中删除。它在接受与 **read** 相同的一组参数的

基础上增添了一个 **FLAGS** 参数。指定 **FLAGS** 为 **MSG_PEEK** 可以使被读取的数据仍保留在输入序列中。

5.5.4 本地套接字

要通过套接字连接同一台主机上的进程，可以使用符号常量 **PF_LOCAL** 和 **PF_UNIX** 所代表的本地命名空间。它们被称为本地套接字（*local sockets*）或者 UNIX 域套接字（*UNIX-domain sockets*）。它们的套接字地址用文件名表示，且只在建立连接的时候使用。

套接字的名字在 **struct sockaddr_un** 结构中指定。你必须将 **sun_family** 字段设置为 **AF_LOCAL** 以表明它使用本地命名空间。该结构中的 **sun_path** 字段指定了套接字使用的路径，该路径长度必须不超过 108 字节。而 **struct sockaddr_un** 的实际长度应由 **SUN_LENG** 宏计算得到。可以使用任何文件名作为套接字路径，但是进程必须对所指定的目录具有写权限，以便向目录中添加文件。如果一个进程要连接到一个本地套接字，则必须具有该套接字的读权限。尽管多台主机可能共享一个文件系统，只有同一台主机上运行的程序之间可以通过本地套接字通信。

本地命名空间的唯一可选协议是 0。

因为它存在于文件系统中，本地套接字可以作为一个文件被列举。如下面的例子，注意开头的 **s**：

```
% ls -l /tmp/socket
srwxrwx--x    1 user    group    0 Nov 13 19:18 /tmp/socket
```

当结束使用的时候，调用 **unlink** 删除本地套接字。

5.5.5 使用本地套接字的示例程序

我们用两个程序展示套接字的使用。列表 5.10 中的服务器程序建立一个本地命名空间套接字并通过它监听连接。当它连接之后，服务器程序不断从中读取文本信息并输出这些信息直到连接关闭。如果其中一条信息是“quit”，服务器程序将删除套接字，然后退出。服务器程序 **socket-server** 接受一个标识套接字路径的命令行参数。

代码列表 5.10 （socket-server.c）本地命名空间套接字服务器

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* 不断从套接字读取并输出文本信息直到套接字关闭。当客户端发送“quit”消息的
时候返回非 0 值，否则返回 0。*/
int server (int client_socket)
{
  while (1) {
    int length;
```

```
      char* text;

      /* 首先，从套接字中获取消息的长度。如果 read 返回 0 则说明客户端关闭了连
接。*/
      if (read (client_socket, &length, sizeof (length)) == 0)
        return 0;
      /* 分配用于保存信息的缓冲区。*/
      text = (char*) malloc (length);
      /* 读取并输出信息。*/
      read (client_socket, text, length);
      printf (“%s\n”, text);
      /* 如果客户消息是“quit”，我们的任务就此结束。*/
      if (!strcmp (text, “quit”)) {
        /* 释放缓冲区。*/
        free (text);
        return 1;
      }

      /* 释放缓冲区。*/
      free (text);
      /* 译者注：合并了勘误中的提示，并增加此返回语句。*/
      return 0;
    }
  }

  int main (int argc, char* const argv[])
  {
    const char* const socket_name = argv[1];
    int socket_fd;
    struct sockaddr_un name;
    int client_sent_quit_message;

    /* 创建套接字。*/
    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
    /* 指明这是服务器。*/
    name.sun_family = AF_LOCAL;
    strcpy (name.sun_path, socket_name);
    bind (socket_fd, &name, SUN_LEN (&name));
    /* 监听连接。*/
    listen (socket_fd, 5);

    /* 不断接受连接，每次都调用 server() 处理客户连接。直到客户发送“quit”消
息的时候退出。*/
    do {
```

```
      struct sockaddr_un client_name;
      socklen_t client_name_len;
      int client_socket_fd;

      /* 接受连接。*/
      client_socket_fd    =    accept    (socket_fd,    &client_name,
&client_name_len);
      /* 处理连接。*/
      client_sent_quit_message = server (client_socket_fd);
      /* 关闭服务器端连接到客户端的套接字。*/
      close (client_socket_fd);
    }
    while (!client_sent_quit_message);

    /* 删除套接字文件。*/
    close (socket_fd);
    unlink (socket_name);

    return 0;
  }
```

列表 5.11 中的客户端程序将连接到一个本地套接字并发送一条文本消息。本地套接字的路径和要发送的消息由命令行参数指定。

代码列表 5.11 （socket-client.c）本地命名空间套接字客户端

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* 将 TEXT 的内容通过 SOCKET_FD 代表的套接字发送。*/
void write_text (int socket_fd, const char* text)
{
  /* 输出字符串（包含结尾的 NUL 字符）的长度。*/
  int length = strlen (text) + 1;
  write (socket_fd, &length, sizeof (length));
  /* 输出字符串。*/
  write (socket_fd, text, length);
}

int main (int argc, char* const argv[])
```

```
{
  const char* const socket_name = argv[1];
  const char* const message = argv[2];
  int socket_fd;
  struct sockaddr_un name;

  /* 创建套接字。*/
  socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
  /* 将服务器地址写入套接字地址结构中。*/
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, socket_name);
  /* 连接套接字。*/
  connect (socket_fd, &name, SUN_LEN (&name));
  /* 将由命令行指定的文本信息写入套接字。*/
  write_text (socket_fd, message);
  close (socket_fd);
  return 0;
}
```

在客户端发送文本信息之前，客户端先通过发送整型变量 **length** 的方式将消息的长度通知服务端。类似的，服务端在读取消息之前先从套接字读取一个整型变量以获取消息的长度。这提供给服务器一个在接收信息之前分配合适大小的缓冲区保存信息的方法。

要尝试这个例子，应在一个窗口中运行服务端程序。指定一个套接字文件的路径——例如 /tmp/socket 作为参数：

```
% ./socket-server /tmp/socket
```

在另一个窗口中指明同一个套接字和消息，并多次运行客户端程序。

```
% ./socket-client /tmp/socket “Hello, world.”
% ./socket-client /tmp/socket “This is a test.”
```

服务端将接收并输出这些消息。要关闭服务端程序，从客户端发送“quit”即可：

```
% ./socket-client /tmp/socket “quit”
```

这样服务端程序就会退出。

5.5.6 Internet 域套接字

UNIX 域套接字只能用于同主机上的两个进程之间通信。Internet 域套接字则可以用来连接网络中不同主机上的进程。

用于在 Internet 范围连接不同进程的套接字属于 Internet 命名空间，使用 **PF_INET** 表示。最常用的协议是 **TCP/IP**。Internet 协议（*Internet Protocol*，***IP***）是一个低层次的协议，负责包在 Internet 中的传递，并在需要的时候负责分片和重组数据包。它只保证“尽量”地发送，因此包可能在传输过程中丢失，或者前后顺序被打乱。参与传输的每台主机都由一个独一无

二的 **IP** 数字标识。传输控制协议（*Transmission Control Protocol*，***TCP***）架构于 **IP** 协议之上，提供了可靠的面向连接的传输。它允许主机之间建立类似电话的连接且保证数据传输的可靠性和有序性。

Internet 套接字的地址包含两个部分：主机和端口号。这些信息保存在 **`sockaddr_in`** 结构中。将 **`sin_family`** 字段设置为 **AF_INET** 以表示这是一个 Internet 命名空间地址。目标主机的 Internet 地址作为一个 32 位整数保存在 **sin_addr** 字段中。端口号用于区分同一台主机上的不同套接字。因为不同主机可能将多字节的值按照不同的字节序存储，应将 **htons** 将端口号转换为网络字节序。参看 ip 的手册页以获取更多信息。

可以通过调用 **`gethostbyname`** 函数将一个可读的主机名——包括标准的以点分割的 **IP** 地址（如 10.0.0.1）或 **DNS** 名（如 `www.codesourcery.com`）——转换为 32 位 **IP** 数字。这个函数返回一个指向 **`struct hostent`** 结构的指针；其中的 `h_addr` 字段包含了主机的 **IP** 数字。参考列表 5.12 中的示例程序。

列表 5.12 展示了 Internet 域套接字的使用。这个程序会获取由命令行指定的网页服务器的首页。

代码列表 5.12 （socket-inet.c）从 WWW 服务器读取信息

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* 从服务器套接字中读取主页内容。返回成功的标记。*/

void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;
  /* 发送 HTTP GET 请求获取主页内容。*/
  sprintf (buffer, "GET /\n");
  write (socket_fd, buffer, strlen (buffer));
  /* 从套接字中读取信息。调用 read 一次可能不会返回全部信息，所以我们必须不断尝试读取直到真正结束。*/
  while (1) {
    number_characters_read = read (socket_fd, buffer, 10000);
    if (number_characters_read == 0)
      return;
    /* 将数据输出到标准输出流。*/
    fwrite (buffer, sizeof (char), number_characters_read, stdout);
  }
}
```

```
int main (int argc, char* const argv[])
{
  int socket_fd;
  struct sockaddr_in name;
  struct hostent* hostinfo;

  /* 创建套接字。*/
  socket_fd = socket (PF_INET, SOCK_STREAM, 0);
  /* 将服务器地址保存在套接字地址中。*/
  name.sin_family = AF_INET;
  /* 将包含主机名的字符串转换为数字。*/
  hostinfo = gethostbyname (argv[1]);
  if (hostinfo == NULL)
    return 1;
  else
    name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
  /* 网页服务器使用 80 端口。*/
  name.sin_port = htons (80);

  /* 连接到网页服务器。*/
  if (connect (socket_fd, &name, sizeof (struct sockaddr_in)) == -1)
{
    perror (“connect”);
    return 1;
  }
  /* 读取主页内容。*/
  get_home_page (socket_fd);

  return 0;
}
```

这个程序从命令行读取服务器的主机名（不是 **URL**——也就是说，地址中不包括“`http://`”部分）。它通过调用 **gethostbyname** 将主机名转换为 **IP** 地址，然后与主机的 80 端口建立一个流式（TCP 协议的）套接字。网页服务器通过超文本传输协议（*HyperText Transport Protocol*，**HTTP**），因此程序发送 **HTTP GET** 命令给服务器，而服务器传回主页内容作为响应。

例如，可以这样运行程序从 `www.codesourcery.com` 获取主页：

```
% ./socket-inet www.codesourcery.com
<html>
  <meta          http-equiv="Content-Type"          content="text/html;
charset=iso-8859-1">
...
```

标准端口号

根据习惯，网页服务器在 80 端口监听客户端连接。多数 Internet 网络服务都被分配了标准端口号。例如，使用 SSL 的安全网页服务器的在 443 端口监听连接，而邮件服务器（利用 SMTP 协议通信）使用端口 25。

在 GNU/Linux 系统中，协议——服务名关系列表被保存在了**/etc/services**。该文件的第一栏是协议或服务名，第二栏列举了对应的端口号和连接类型：**tcp** 代表了面向连接的协议，而 udp 代表数据报类型的。

如果你用 Internet 域套接字实现了一个自己的协议，应使用高于 1024 的端口号进行监听。

5.5.7 套接字对

如前所示，**pipe** 函数创建了两个文件描述符，分别代表管道的两端。管道有所限制因为文件描述符只能被相关进程使用且经由管道进行的通信是单向的。而 **socketpair** 函数为一台主机上的一对相连接的的套接字创建两个文件描述符。这对文件描述符允许相关进程之间进行双向通信。

它的前三个参数与 **socket** 系统调用相同：分别指明了域、通信类型（译者著：原文为 connection style 连接类型，与前文不符，特此修改）和协议。最后一个参数是一个包含两个元素的整型数组，用于保存创建的两个套接字的文件描述符，与 **pipe** 的参数类似。当调用 **socketpair** 的时候，必须指定 **PF_LOCAL** 作为域。