什么。

[3]可以在 sar 命令的第 3 个参数中输入 1 来指定周期。

```
$ `sar` -P ALL 1
( 略 )
16:29:52  CPU  %user  %nice  %system  %iowait  %steal  %idle
16:29:53  all   0.88   0.00     0.00     0.00    0.00  99.12
16:29:53    0   2.00   0.00     1.00     0.00    0.00  97.00
16:29:53    1   1.00   0.00     0.00     0.00    0.00  99.00
16:29:53    2   0.00   0.00     0.00     0.00    0.00 100.00
16:29:53    3   2.00   0.00     0.00     0.00    0.00  98.00
16:29:53    4   0.00   0.00     0.00     0.00    0.00 100.00
16:29:53    5   1.98   0.00     0.00     0.00    0.00  98.02
16:29:53    6   0.00   0.00     0.00     0.00    0.00 100.00
16:29:53    7   0.99   0.00     0.00     0.00    0.00  99.01

16:29:53  CPU  %user  %nice  %system  %iowait  %steal  %idle
16:29:54  all   0.75   0.00     0.25     0.12    0.00  98.88
16:29:54    0   2.97   0.00     0.00     0.00    0.00  97.03
16:29:54    1   0.99   0.00     0.99     0.00    0.00  98.02
16:29:54    2   0.00   0.00     0.00     0.00    0.00 100.00
16:29:54    3   0.00   0.00     0.00     0.00    0.00 100.00
16:29:54    4   0.00   0.00     0.00     1.00    0.00  99.00
16:29:54    5   1.00   0.00     0.00     0.00    0.00  99.00
16:29:54    6   0.00   0.00     0.00     0.00    0.00 100.00
16:29:54    7   1.00   0.00     0.00     0.00    0.00  99.00
( 略 )
```

如果在运行过程中按下 Ctrl+C，则 sar 命令会结束运行，并输出已采集的所有数据的平均值。

```
( 略 )
Average:  CPU  %user  %nice  %system  %iowait  %steal  %idle
Average:  all   0.71   0.00     0.08     0.04    0.00  99.17
Average:    0   1.66   0.00     0.33     0.00    0.00  98.01
Average:    1   1.00   0.00     0.33     0.00    0.00  98.67
Average:    2   0.33   0.00     0.00     0.00    0.00  99.67
Average:    3   0.67   0.00     0.00     0.00    0.00  99.33
Average:    4   0.00   0.00     0.00     0.33    0.00  99.67
Average:    5   1.00   0.00     0.00     0.00    0.00  99.00
Average:    6   0.00   0.00     0.00     0.00    0.00 100.00
Average:    7   0.99   0.00     0.00     0.00    0.00  99.01
$
```

在每一行中，从 %user 字段到 %idle 字段表示在 CPU 核心上运行的处理的类型。同一行内全部字段的值的总和是 100%。通过上面展示的数据可以看出，一行数据对应一个 CPU 核心。这里输出的是笔者的计算机上搭载的 8 个 CPU 核心的数据（CPU 字段中值为 all 的那一行数据是全部 CPU 核心的平均值）。

将 %user 字段与 %nice 字段的值相加得到的值是进程在用户模式下运行的时间比例（第 4 章将说明 %user 与 %nice 的区别），而 CPU 核心在内核模式下执行系统调用等处理所占的时间比例可以通过 %system 字段得到。在采集数据时，所有 CPU 核心的 %idle 字段的值都几乎接近 100%。这里的 %idle 指的是 CPU 核心完全没有运行任何处理时的**空闲**（idle）状态（详见第 4 章）。关于剩余的字段，我们将在以后用到时再说明。

另外，也可以通过 sar 命令的第 4 个参数来指定采集信息的次数，如下所示[4]。

[4]在本例中是每秒采集 1 次。

```
$ sar -P ALL 1 1
( 略 )
16:32:50  CPU  %user  %nice  %system  %iowait  %steal  %idle
16:32:51  all   0.13   0.00     0.00     0.00    0.00  99.87
16:32:51    0   0.00   0.00     0.00     0.00    0.00 100.00
16:32:51    1   0.00   0.00     0.00     0.00    0.00 100.00
16:32:51    2   0.00   0.00     0.00     0.00    0.00 100.00
16:32:51    3   0.00   0.00     0.00     0.00    0.00 100.00
16:32:51    4   0.99   0.00     0.00     0.00    0.00  99.01
16:32:51    5   1.00   0.00     0.00     0.00    0.00  99.00
16:32:51    6   0.00   0.00     0.00     0.00    0.00 100.00
16:32:51    7   0.00   0.00     0.00     0.00    0.00 100.00

Average:  CPU  %user  %nice  %system  %iowait  %steal  %idle
Average:  all   0.13   0.00     0.00     0.00    0.00  99.87
Average:    0   0.00   0.00     0.00     0.00    0.00 100.00
Average:    1   0.00   0.00     0.00     0.00    0.00 100.00
Average:    2   0.00   0.00     0.00     0.00    0.00 100.00
Average:    3   0.00   0.00     0.00     0.00    0.00 100.00
Average:    4   0.99   0.00     0.00     0.00    0.00  99.01
Average:    5   1.00   0.00     0.00     0.00    0.00  99.00
Average:    6   0.00   0.00     0.00     0.00    0.00 100.00
Average:    7   0.00   0.00     0.00     0.00    0.00 100.00
$
```

下面，我们来尝试运行一个不发起任何系统调用，只是单纯地执行循环的程序，并通过 sar 命令查看它在各模式下的运行时间（代码清单 2-3）。

代码清单 2-3　loop 程序（loop.c）

```
int main(void)
{
    for(;;)
        ;
}
```

编译并运行这段代码，将出现以下结果。

```
$ cc -o loop loop.c
$ ./loop &
[1] 13093
$ sar -P ALL 1 1
( 略 )
16:45:45 CPU  %user %nice %system %iowait %steal  %idle
16:45:46 all  12.86  0.00    0.12    0.00   0.00  87.02
16:45:46   0 100.00  0.00    0.00    0.00   0.00   0.00 ←①
16:45:46   1   0.00  0.00    0.00    0.00   0.00 100.00
16:45:46   2   0.00  0.00    0.00    0.00   0.00 100.00
16:45:46   3   1.00  0.00    0.00    0.00   0.00  99.00
16:45:46   4   0.99  0.00    0.00    0.00   0.00  99.01
16:45:46   5   1.01  0.00    0.00    0.00   0.00  98.99
16:45:46   6   0.00  0.00    0.00    0.00   0.00 100.00
16:45:46   7   0.00  0.00    0.00    0.00   0.00 100.00
( 略 )
```

参照①指向的那一行数据，可以看出在采集信息的这 1 秒内，用户进程（即 loop 程序）始终运行在 CPU 核心 0 上（图 2-3）。

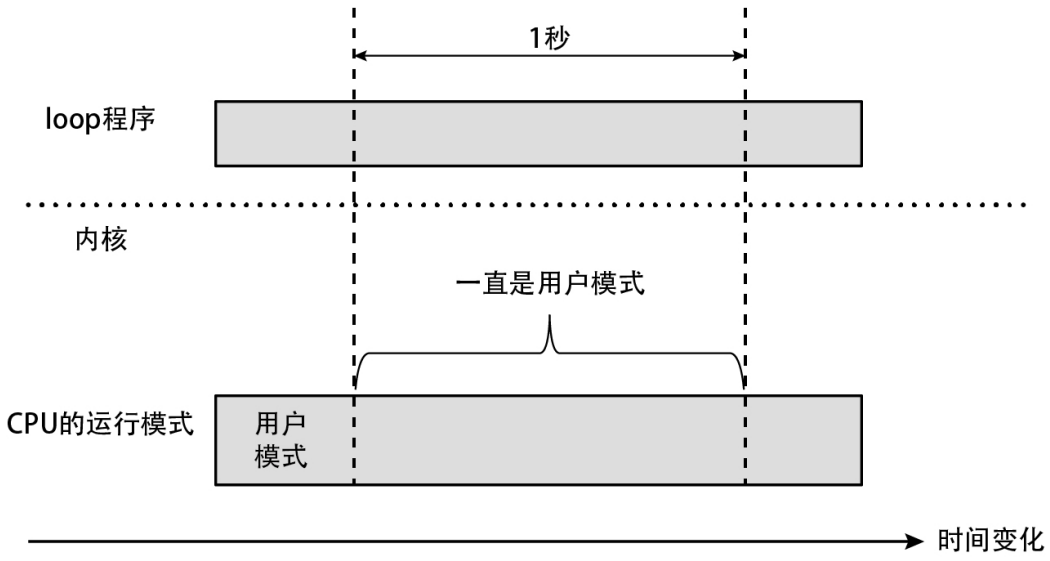

图 2-3　loop 程序的运行

在测试完成后，记得结束正在运行的 loop 程序[5]。

[5]把想结束的程序的进程 ID 指定为 kill 命令的参数即可。在输入运行命令时附加一个 &，即可获取 loop 程序的进程 ID。

```
$ kill 13093
$
```

接着，让我们对循环执行 getppid() 这个用于获取父进程的进程 ID 的系统调用的程序进行相同的操作（代码清单 2-4）。

代码清单 2-4　ppidloop 程序（ppidloop.c）

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

int main(void)
{
    for(;;)
        getppid();
}
```

编译并运行这段代码，将出现以下结果。

```
$ cc -o ppidloop ppidloop.c
$ ./ppidloop &
[1] 13389
$ sar -P ALL 1 1
( 略 )
16:49:11  CPU  %user %nice %system %iowait %steal  %idle
16:49:12  all   3.51  0.00    9.02    0.00   0.00  87.47
16:49:12    0   0.00  0.00    0.00    0.00   0.00 100.00
16:49:12    1  28.00  0.00   72.00    0.00   0.00   0.00 ←②
16:49:12    2   0.00  0.00    0.00    0.00   0.00 100.00
16:49:12    3   0.00  0.00    0.00    0.00   0.00 100.00
16:49:12    4   0.99  0.00    0.99    0.00   0.00  98.02
16:49:12    5   0.00  0.00    0.00    0.00   0.00 100.00
16:49:12    6   0.00  0.00    0.00    0.00   0.00 100.00
16:49:12    7   0.00  0.00    0.00    0.00   0.00 100.00
( 略 )
$
```

参照②指向的这一行数据，可以看出在采集信息的这 1 秒内，发生了以下情况（图 2-4）。

- 在 CPU 核心 1 上，运行 **ppidloop** 程序占用了 28% 的运行时间
- 根据 **ppidloop** 程序发出的请求来获取父进程的进程 ID 这一内核处理占用了 72% 的运行时间

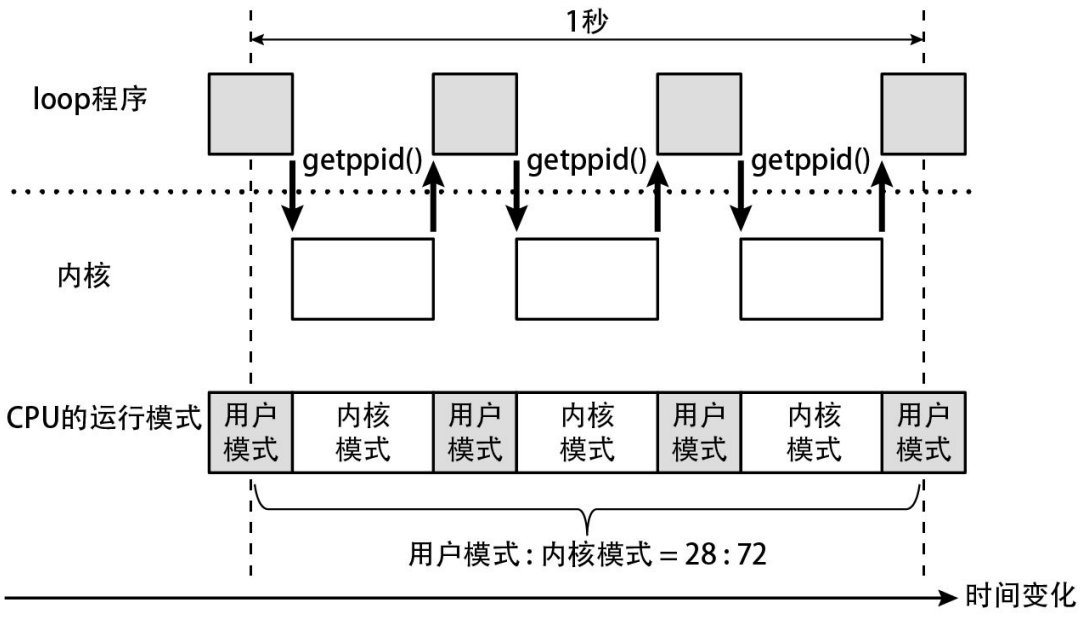

图 2-4　ppidloop 程序的运行

为什么 %system 的值不是 100% 呢？这是因为，用于循环执行 main() 函数内的 getppid() 的循环处理，是属于进程自身的处理。

在测试完成后，也不要忘记结束正在运行的程序。

```
$ kill 13389
$
```

虽然不能一概而论，但当 %system 的值高达几十时，大多是陷入了系统调用发起过多，或者系统负载过高等糟糕的状态。

● 执行系统调用所需的时间

在 strace 命令后加上 -T 选项，就能以微秒级的精度来采集各种系统调用所消耗的实际时间。在发现 %system 的值过高时，可以通过这个功能来确认到底是哪个系统调用占用了过多的系统资源。下面是对 hello world 程序使用 strace -T 后的输出结果。

```
$ strace -T -o hello.log ./hello
hello world
$ cat hello.log
execve("./hello", ["./hello"], [/* 28 vars */]) ↲
                                    = 0 <0.000225>
brk(NULL)                           = 0x6c6000 <0.000012>
access("etcld.so.nohwcap", F_OK)  = -1 ENOENT ↲
(No such file or directory) <0.000016>
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE| ↲
MAP_ANONYMOUS, -1, 0)   = 0x7ff02b49a000 <0.000013>
access("etcld.so.preload", R_OK)  = -1 ENOENT ↲
(No such file or directory) <0.000014>
( 略 )
brk(0x6e7000)                       = 0x6e7000 <0.000008>
write(1, "hello world\n", 12)       = 12 <0.000014>
exit_group(0)                       = ?
+++ exited with 0 +++
$
```

通过这些信息可以看出，输出 hello world\n 这一字符串总共花费了 14 微秒。

此外，strace 命令还存在其他选项，例如，使用 -tt 选项能以微秒为单位来显示处理发生的时刻。在使用 strace 命令时，可以根据实际需求来选择不同的选项。

2.2 系统调用的包装函数

Linux 提供了所有或者说绝大多数进程所依赖的库函数，以为编写程序提供方便。

需要注意的是，与常规的函数调用不同，系统调用并不能被 C 语言之类的高级编程语言直接发起，只能通过与系统架构紧密相连的汇编语言代码来发起。例如，在 x86_64 架构中，是如下发起 getppid() 这个系统调用的[6]。

> [6]第 1 行的意思是，将 getppid 的系统调用编号 0x6e 传递给 eax 寄存器。这里的系统调用编号是由 Linux 预先定义好的。在第 2 行中，通过 syscall 命令发起系统调用，并切换到内核模式。然后，开始执行负责处理 getppid 的内核代码。

```
mov    $0x6e.%eax
syscall
```

平时没机会接触汇编语言的读者无须了解这些源代码的意义，抱着“这显然与平常见到的源代码完全不一样”这样的想法继续往下看就可以了。

如果没有 OS 的帮助，程序员就不得不根据系统架构为每一个系统调用编写相应的汇编语言代码，然后再从高级编程语言中调用这些代码（图 2-5）。

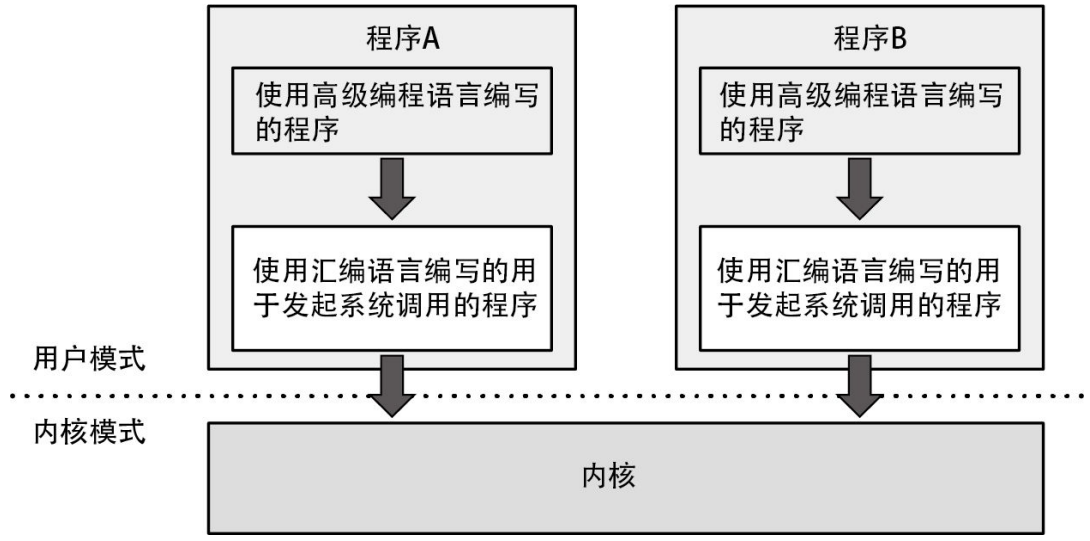

图 2-5　如果没有 OS 的帮助

这样一来，不但编写程序的时间增加了，程序也将无法移植到别的架构上。

为了解决这样的问题，OS 提供了一系列被称为**系统调用的包装函数**的函数，用于在系统内部发起系统调用。各种架构上都存在着对应的包装函数。因此，使用高级编程语言编写的用户程序，只需调用由高级编程语言提供的包装函数即可（图 2-6）。

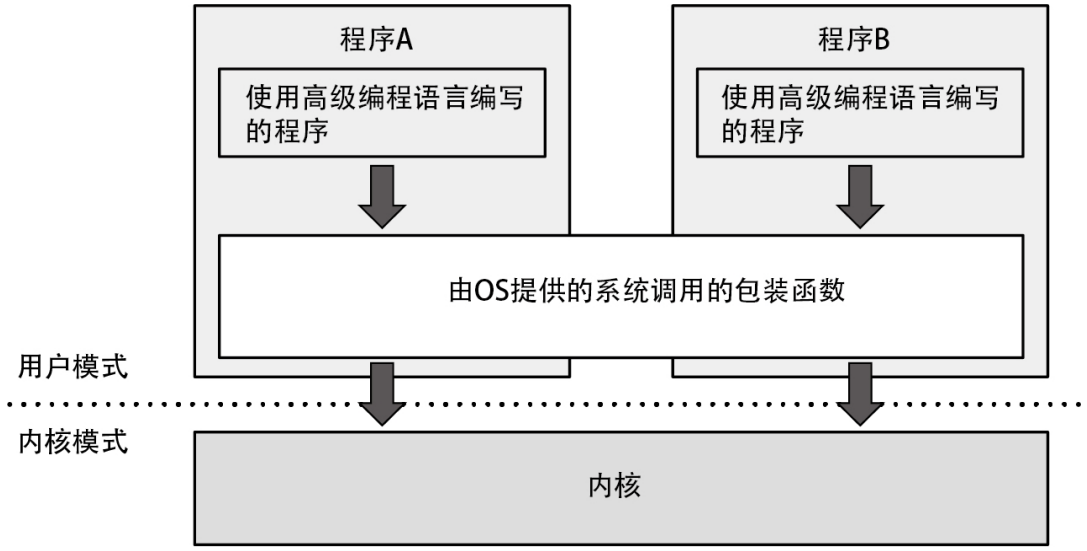

图 2-6　用户程序只需调用包装函数即可

2.3 C 标准库

C 语言拥有 ISO（International Standard Organization，国际标准化组织）定义的标准库，Linux 也提供了这些 C 语言标准库。不过，通常会以 GNU 项目提供的 glibc 作为 C 标准库使用。用 C 语言编写的几乎所有程序都依赖于 glibc 库。

glibc 不仅包括系统调用的包装函数，还提供了 POSIX 标准[7]中定义的函数。

[7]这个标准定义了基于 UNIX 的 OS 应该具备的各种功能。

Linux 提供了 ldd 命令，用于查看程序所依赖的库。

我们来尝试对 echo 命令使用 ldd 命令。

```
$ ldd bin/echo
    linux-vdso.so.1 => (0x00007fffed1a2000)
    libc.so.6 => lib/x86_64-linux-gnu/libc.so.6 (0x00007fddf1101000)
    lib64/ld-linux-x86-64.so.2 (0x000055605066a000)
$
```

在上面的结果中，libc 指的就是 C 标准库。

下面来看一下 ppidloop 命令依赖于哪些库。

```
$ ldd ppidloop
    linux-vdso.so.1 => (0x00007fff43dd7000)
    libc.so.6 => lib/x86_64-linux-gnu/libc.so.6 (0x00007f5e3884b000)
    lib64/ld-linux-x86-64.so.2 (0x000055c4b2926000)
$
```

可以看出，ppidloop 命令同样也依赖于 libc。

接下来，确认一下 Python 3 提供的 python3 命令依赖于哪些库。

```
$ ldd /usrbinpython3
    linux-vdso.so.1 => (0x00007ffe629d0000)
    libpthread.so.0 => libx86_64-linux-gnu/libpthread.so.0 ↲
(0x00007fab961a9000)
    libc.so.6 => libx86_64-linux-gnu/libc.so.6             ↲
(0x00007fab95ddf000)
    libdl.so.2 => libx86_64-linux-gnu/libdl.so.2           ↲
(0x00007fab95bda000)
    libutil.so.1 => libx86_64-linux-gnu/libutil.so.1       ↲
(0x00007fab959d7000)
    libexpat.so.1 => libx86_64-linux-gnu/libexpat.so.1     ↲
(0x00007fab957ae000)
    libz.so.1 => libx86_64-linux-gnu/libz.so.1             ↲
(0x00007fab95593000)
    libm.so.6 => libx86_64-linux-gnu/libm.so.6             ↲
(0x00007fab9528a000)
    lib64ld-linux-x86-64.so.2 (0x0000565208bd3000)
$
```

python3 命令同样也依赖于 libc。用 Python 3 编写的脚本虽然可以通过 python3 命令直接运行，但是通过上面展示的结果可以看出，python3 命令本身的实现也依赖于 C 标准库。可以说，在 OS 层面上，C 语言依然在默默地发挥着很大的作用，是一种不可或缺的编程语言。

对系统上的其他程序使用 ldd 命令，会发现它们大部分也依赖于 libc。请大家在各自的计算机上尝试一下。

除了 C 标准库之外，Linux 还提供了 C++ 等各种编程语言的标准库，以及大部分程序有可能用得上的各种库。

2.4 OS 提供的程序

OS 提供的程序与 OS 提供的库一样，对绝大多数程序来说是不可或缺的。OS 同时还提供了作为自身一部分的、用于更改 OS 运行方式的程序。

下面列举了一些 OS 提供的程序。

- 初始化系统：**init**
- 变更系统的运行方式：**sysctl**、**nice**、**sync**
- 文件操作：**touch**、**mkdir**
- 文本数据处理：**grep**、**sort**、**uniq**
- 性能测试：**sar**、**iostat**
- 编译：**gcc**
- 脚本语言运行环境：**perl**、**python**、**ruby**
- shell：**bash**
- 视窗系统：**X**

大家应该都直接或间接地使用过这些程序。本书后面将对其中一部分程序进行介绍。

第 3 章　进程管理

本章将介绍内核提供的创建与删除进程的功能。但是，不了解第 5 章中关于虚拟内存的内容，就无法理解 Linux 创建与删除进程的机制。因此，本章将抛开虚拟内存，单纯地讲述进程的创建与删除，第 5 章再详细介绍完整的运行机制。

3.1 创建进程

在 Linux 中，创建进程有如下两个目的。

- 将同一个程序分成多个进程进行处理（例如，使用 Web 服务器接收多个请求）
- 创建另一个程序（例如，从 **bash** 启动一个新的程序）

为了达成这两个目的，Linux 提供了 `fork()` 函数与 `execve()` 函数（其底层分别请求名为 `clone()` 与 `execve()` 的系统调用）。接下来，我们将介绍如何使用这两个函数。

3.2 fork() 函数

要想将同一个程序分成多个进程进行处理，只需使用 fork() 函数即可。在调用 fork() 函数后，就会基于发起调用的进程，创建一个新的进程。发出请求的进程称为**父进程**，新创建的进程称为**子进程**。

创建新进程的流程如下所示（图 3-1）。

① 为子进程申请内存空间，并复制父进程的内存到子进程的内存空间。

② 父进程与子进程分裂成两个进程，以执行不同的代码。这一点的实现依赖于 **fork()** 函数分别返回不同的值给父进程与子进程。

为了对 fork() 函数一探究竟，我们来编写一个实现下述要求的程序。

① 创建一个新进程。

② 父进程输出自身与子进程的进程 ID，而子进程只输出自身的进程 ID。

实现上述要求的 fork 程序如代码清单 3-1 所示。