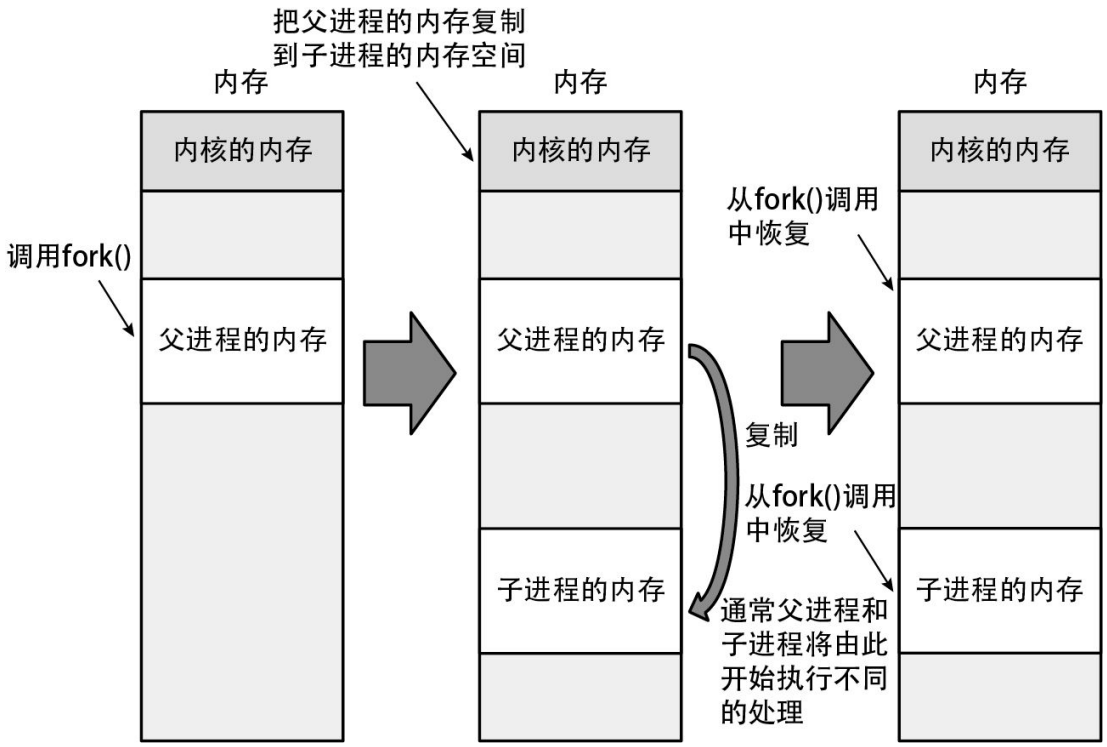

图 3-1　创建进程的流程

代码清单 3-1　fork 程序（fork.c）

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

static void child()
{
    printf("I'm child! my pid is %d.\n", getpid());
    exit(EXIT_SUCCESS);
}

static void parent(pid_t pid_c)
{
    printf("I'm parent! my pid is %d and the pid of my child
           is %d.\n", getpid(), pid_c);
    exit(EXIT_SUCCESS);
}

int main(void)
{
    pid_t ret;
    ret = fork();
```

```
    if(ret == -1)
        err(EXIT_FAILURE, "fork() failed");
    if(ret == 0) {
        //fork()会返回0给子进程，因此这里调用child()
        child();
    } else {
        //fork()会返回新创建的子进程的进程ID（大于1）给父进程，因此这里调用
parent()
        parent(ret);
    }
    // 在正常运行时，不可能运行到这里
    err(EXIT_FAILURE, "shouldn't reach here");
}
```

当父进程和子进程从 fork() 函数的调用中恢复时，会获得 fork() 函数的返回值。fork() 函数对父进程返回子进程的进程 ID，而对子进程返回 0。代码清单 3-1 的实验程序正是利用这一点，令父进程和子进程执行了不同的处理。

编译并运行代码，结果如下。

```
$ cc -o fork fork.c
$ ./fork
I'm parent! my pid is 4193 and the pid of my child is 4194.
I'm child! my pid is 4194.
$
```

进程 ID 为 4193[1]的进程分裂，创建了一个进程 ID 为 4194 的新进程。同时也能看到，在调用 fork() 函数后，两个进程执行的处理也不同了。

[1]在不同的计算机上会出现不同的进程 ID。

刚开始我们可能难以理解 fork() 函数到底做了什么处理，但在理解其原理后，就会发现其实它做的事情非常简单。

3.3 execve() 函数

在打算启动另一个程序时，需要调用 execve() 函数。首先，我们来看一下内核在运行进程时的流程。

① 读取可执行文件，并读取创建进程的内存映像所需的信息。

② 用新进程的数据覆盖当前进程的内存。

③ 从最初的命令开始运行新的进程。

也就是说，在启动另一个程序时，并非新增一个进程，而是替换了当前进程（图 3-2）。

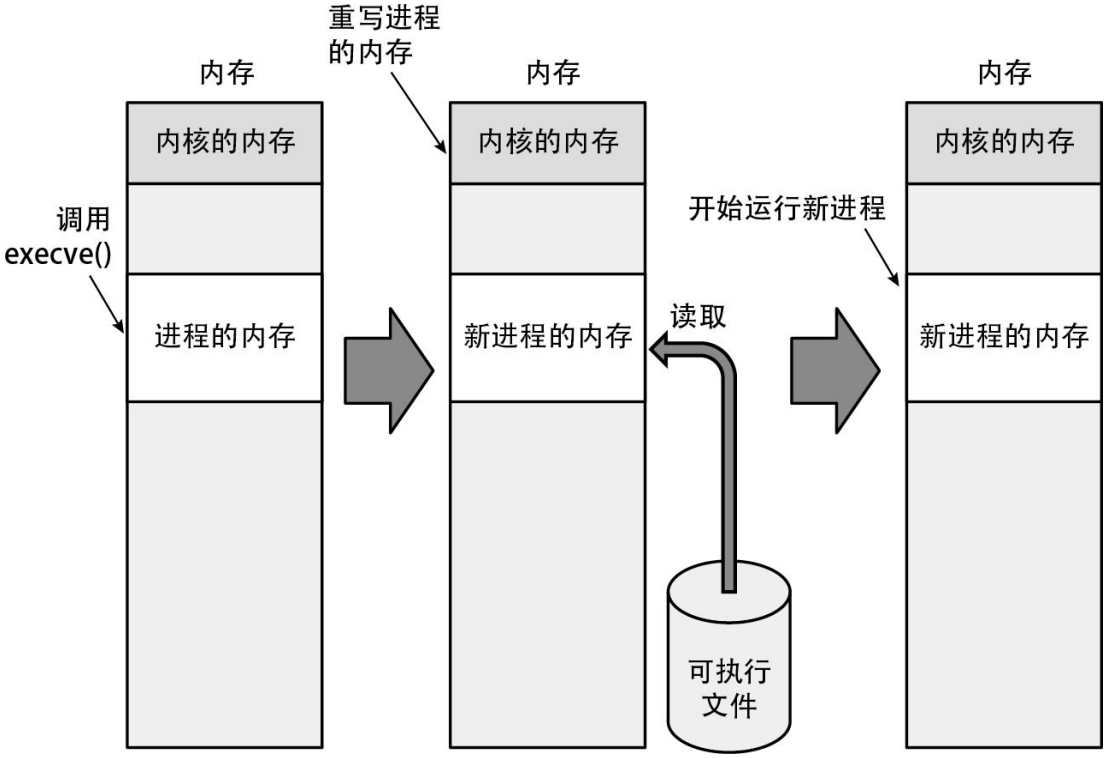

图 3-2 启动另一个程序的流程

下面来详细说明这一流程。

首先，读取可执行文件，以及创建进程的内存映像所需的信息。可执行文件中不仅包含进程在运行过程中使用的代码与数据，还包含开始运行程序时所需的数据。

- 包含代码的代码段在文件中的偏移量、大小，以及内存映像的起始地址
- 包含代码以外的变量等数据的数据段在文件中的偏移量、大小，以及内存映像的起始地址
- 程序执行的第一条指令的内存地址（入口点）

假设将要运行的程序的可执行文件的结构如图 3-3 所示。

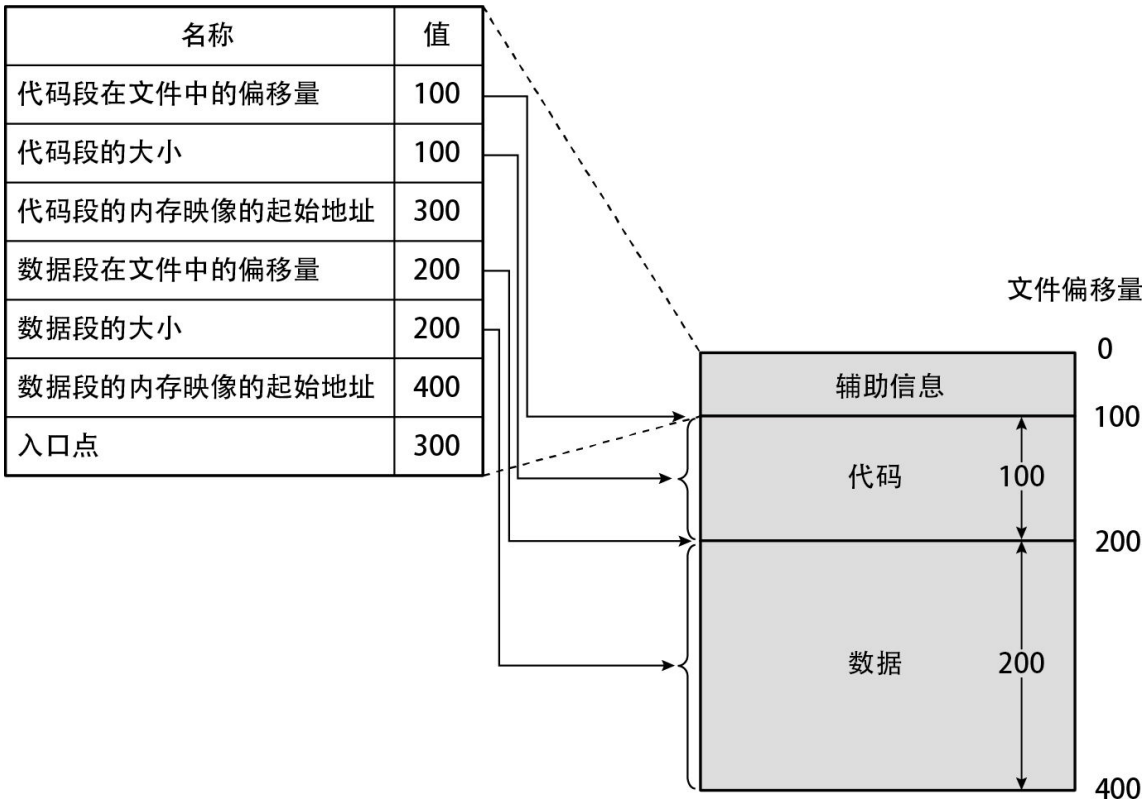

图 3-3　可执行文件的结构（例子）

与使用高级编程语言编写的源代码不同，在 CPU 上执行机器语言指令时，必须提供需要操作的内存地址，因此在代码段和数据段中必须包含内存映像的起始地址。比如，使用一种虚构的高级编程语言编写了如下一段源代码。

```
c = a + b
```

在机器语言层面，这段源代码将转变成下面这样的直接对内存地址进行操作的指令。

```
load m100 r0          ← 将内存地址100（变量a）的值读取到名为r0的寄存器中
load m200 r1          ← 将内存地址200（变量b）的值读取到名为r1的寄存器中
add r0 r1 r2          ← 将r0与r1相加，并将结果储存到名为r2的寄存器中
store r2 m300         ← 将r2的值储存到内存地址300（变量c）
```

接下来，基于读取的信息，将程序映射到内存上，如图 3-4 所示。

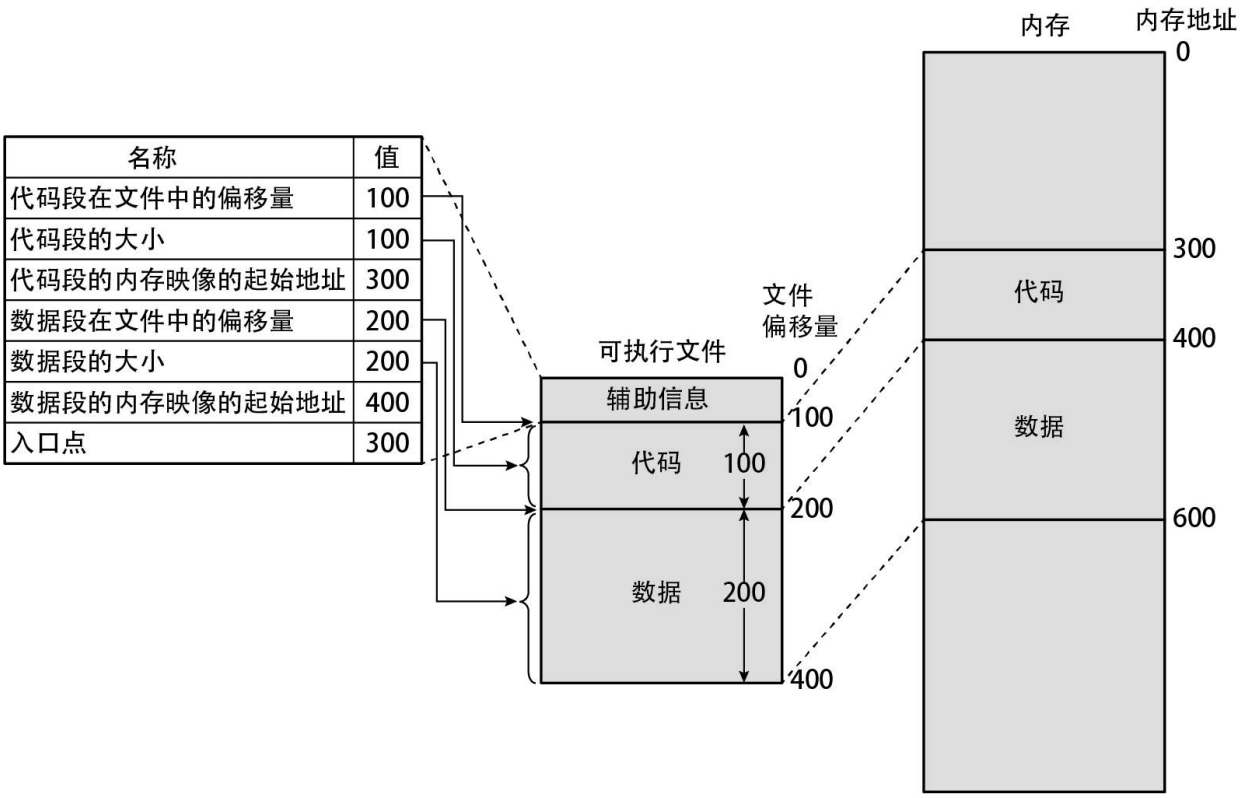

图 3-4　基于可执行文件的信息，将程序映射到内存上

最后，从入口点开始运行程序（图 3-5）。

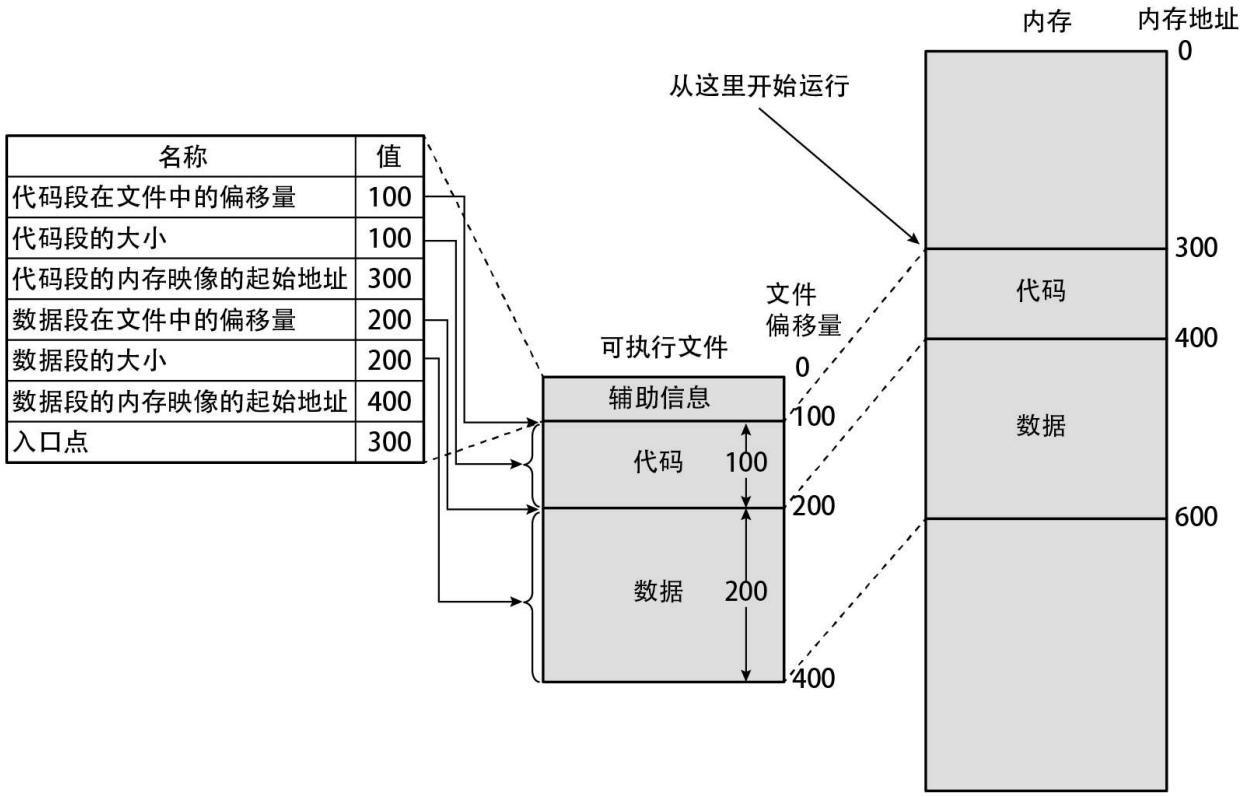

图 3-5　从入口点开始运行程序

然而，Linux 的可执行文件的结构遵循的是名为 ELF（Executable and Linkable Format，可执行与可链接格式）的格式，并非之前描述的那样简单的结构。ELF 的相关信息可以通过 `readelf` 命令来获取。

现在我们来尝试获取 *bin*`sleep` 的 ELF 信息。

通过附加 `-h` 选项，可以获取起始地址。

```
$ readelf -h binsleep
ELF Header:
( 略 )
    Entry point address:                    0x401760
( 略 )
$
```

`Entry point address` 这一行中的 `0x401760` 就是这个程序的入口点。

通过附加 -S 选项，可以获取代码段与数据段在文件中的偏移量、大小和起始地址。

```
$ readelf -S binsleep
Section Headers:
  [Nr] Name     Type          Address              Offset
       Size              EntSize       Flags Link Info  Align
( 略 )
  [14] .text    PROGBITS      00000000004014e0     000014e0
       0000000000003319 0000000000000000 AX    0    0    16
( 略 )
  [25] .data    PROGBITS      00000000006071c0     000071c0
       0000000000000074 0000000000000000 WA    0    0    32
( 略 )
$
```

运行后得到了大量输出信息，不过现在只需要理解以下内容即可。

- 输出的数据每两行为一组
- 全部数值皆为十六进制数
- 在每组的第 1 行的第 2 个字段中，**.text** 对应的是代码段的信息，**.data** 对应的是数据段的信息
- 在这些信息中，对于每组输出，只需要关注以下内容即可
  - 每组的第 1 行的第 4 个字段：内存映像的起始地址
  - 每组的第 1 行的第 5 个字段：在文件中的偏移量
  - 每组的第 2 行的第 1 个字段：大小

可以看出，*bin*sleep 的信息如下所示。

名称	值
代码段在文件中的偏移量	0x14e0
代码段的大小	0x3319
代码段的内存映像的起始地址	0x4014e0

名称	值
数据段的大小	0x74
数据段的内存映像的起始地址	0x6071c0
入口点	0x401760

在程序运行时创建的进程的内存映像信息，可以从 *proc*pid/maps 这一文件中找到。sleep 命令的相关信息如下所示。

```
$ binsleep 10000 &
[1] 3967
$ cat proc3967/maps
00400000-00407000 r-xp 00000000 08:01 23994     binsleep  ←①
( 略 )
00607000-00608000 rw-p 00007000 08:01 23994     binsleep  ←②
( 略 )
$
```

①所指的是代码段，②所指的是数据段。可以看到，它们都在上表所示的内存映像的范围内。

在查看完后，记得结束正在运行的程序。

```
$ kill 3967
$
```

在打算新建一个别的进程时，通常采用被称为 fork and exec 的方式，即由父进程调用 fork() 创建子进程，再由子进程调用 exec()。图 3-6 所示为由 bash 进程创建 echo 进程的流程。

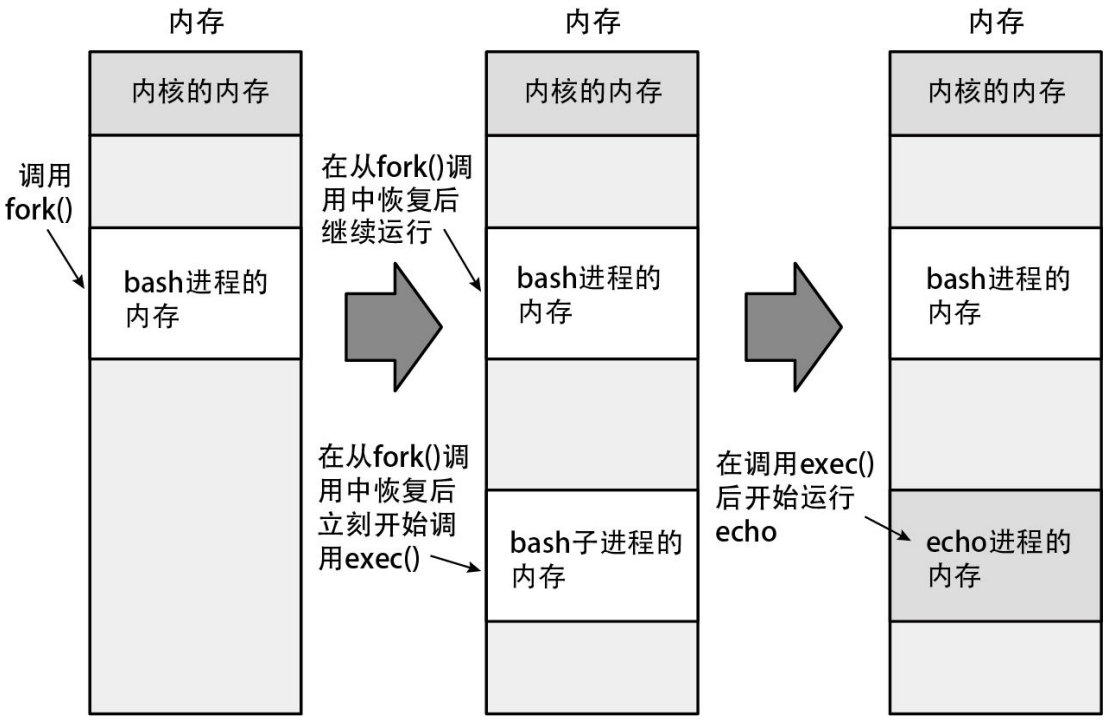

图 3-6　由 bash 进程创建 echo 进程的流程

然后，我们编写一个实现下述要求的程序，来了解一下 fork and exec 方式。

① 创建一个新的进程。

② 在创建 **echo hello** 程序后，父进程输出自身与子进程的进程 ID，并结束运行，子进程输出自身的进程 ID，然后结束运行。

代码清单 3-2 所示为实现了上述要求的源代码。

代码清单 3-2　fork-and-exec 程序（fork-and-exec.c）

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

static void child()
{
    char *args[] = {"bin/echo", "hello", NULL};
    printf("I'm child! my pid is %d.\n", getpid());
    fflush(stdout);
```

```
    execve("/bin/echo", args, NULL);
    err(EXIT_FAILURE, "exec() failed");
}
static void parent(pid_t pid_c)
{
    printf("I'm parent! my pid is %d and the pid of my child
           is %d.\n", getpid(), pid_c);
    exit(EXIT_SUCCESS);
}

int main(void)
{
    pid_t ret;
    ret = fork();
    if (ret == -1)
        err(EXIT_FAILURE, "fork() failed");
    if (ret == 0) {
        //fork()会返回0给子进程，因此这里调用child()
        child();
    } else {
        //fork()会返回新创建的子进程的进程ID（大于1）给父进程，因此这里调用
parent()
        parent(ret);
    }
    // 在正常运行时，不可能运行到这里
    err(EXIT_FAILURE, "shoudln't reach here");
}
```

编译并运行这段代码，结果如下。

```
$ cc -o fork-and-exec fork-and-exec.c
$ ./fork-and-exec
I'm parent! my pid is 4203 and the pid of my child is 4204.
I'm child! my pid is 4204.
$ hello
```

从上面的运行结果可以看出，这段代码能正常运行。

在 C 语言之外的其他编程语言中，例如在 Python 中，可以通过 `os.exec()` 函数来请求 `execve()` 系统调用。

3.4 结束进程

可以使用 _exit() 函数（底层发起 exit_group() 系统调用）来结束进程。在进程运行结束后，如图 3-7 所示，所有分配给进程的内存将被回收。

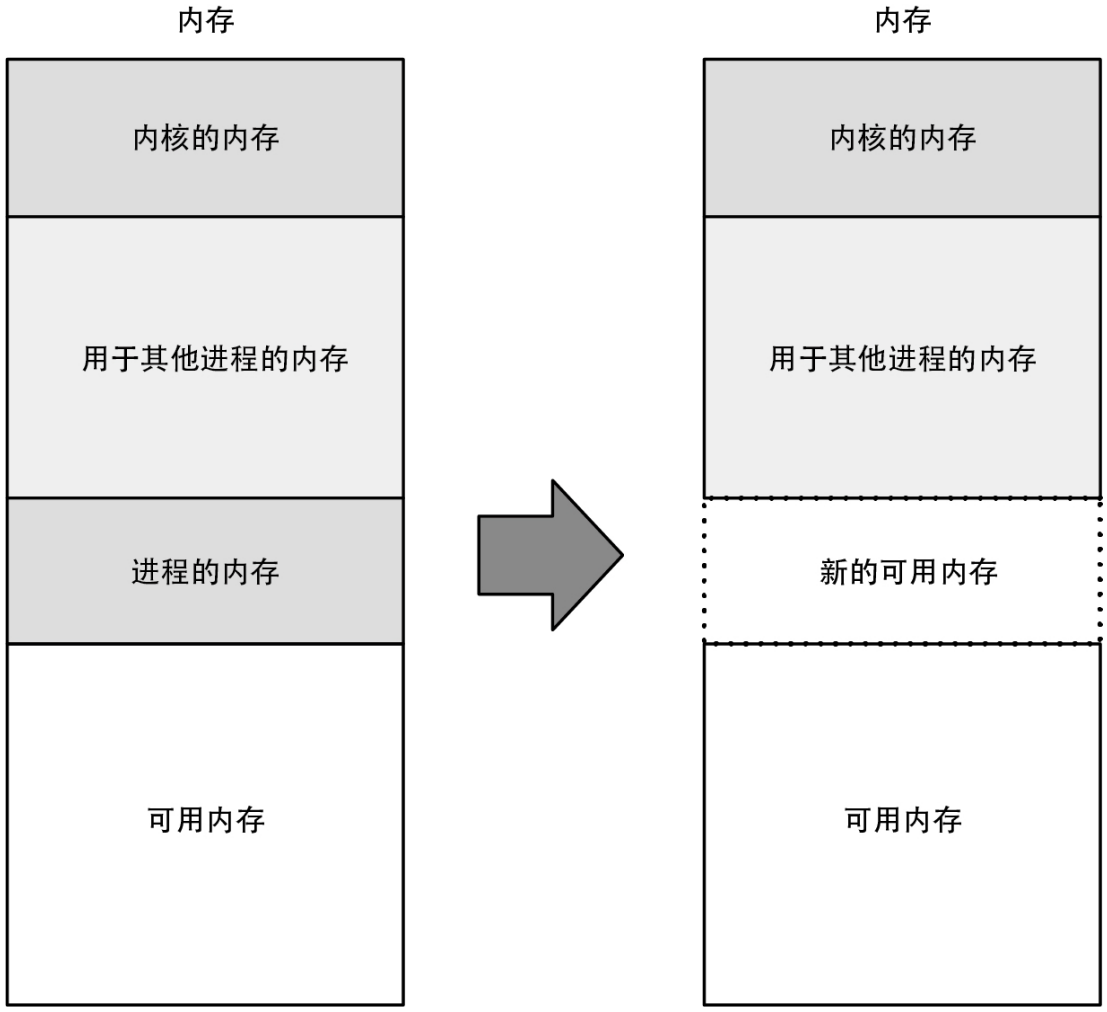

图 3-7 在进程运行结束时回收其内存

不过，通常我们很少会直接调用 _exit() 函数，而是通过调用 C 标准库中的 exit() 函数来结束进程的运行。在这种情况下，C 标准库会在调用完自身的终止处理后调用 _exit() 函数。在从 main() 函数中恢复时也是同样的方式。

第 4 章　进程调度器

Linux 内核具有进程调度器（以下简称“调度器”）的功能，它使得多个进程能够同时运行（准确来说，是看起来在同时运行）。大家在使用 Linux 系统时通常是意识不到调度器的存在的。为了加深对调度器的理解，本章将深入探究调度器的运作方式。

在计算机相关的图书中，一般是这样介绍调度器的。

- 一个 CPU 同时只运行一个进程
- 在同时运行多个进程时，每个进程都会获得适当的时长 [1]，轮流在 CPU 上执行处理

[1]称为时间片（time slice）。

例如，当存在 p0、p1 和 p2 这 3 个进程时，调度器的运作方式如图 4-1 所示。

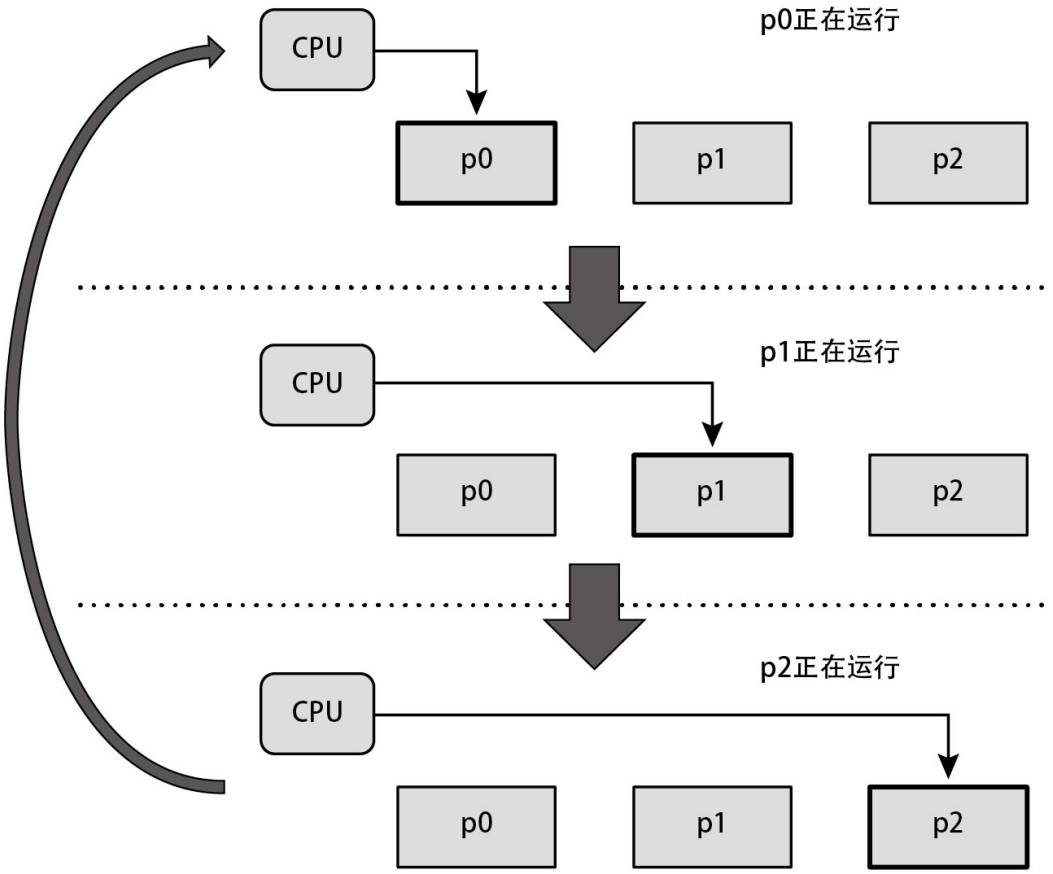

图 4-1　教科书中描述的调度器的运作方式

本章将利用实验程序来验证调度器是否真的如上面描述的那样运作。

需要指出的是，Linux 会将多核 CPU（现在几乎都是这样的 CPU）上的每一个核心都识别为一个 CPU。在本书中，我们将系统识别出来的 CPU（这里是指 CPU 核心）称为**逻辑 CPU**。另外，在开启了超线程功能时，每一个线程都会被识别为一个逻辑 CPU。我们将在第 6 章讲解超线程的相关内容。

4.1 关于实验程序的设计

在同时运行一个或多个一味消耗 CPU 时间执行处理的进程时，采集以下统计信息。

- 在某一时间点运行在逻辑 CPU 上的进程是哪一个
- 每个进程的运行进度

通过分析这些信息，来确认本章开头对调度器的描述是否正确。实验程序的设计如下。

- 命令行参数
  - 第 1 个参数（**n**）：同时运行的进程数量
  - 第 2 个参数（**total**）：程序运行的总时长（单位：毫秒）
  - 第 3 个参数（**resol**）：采集统计信息的间隔（单位：毫秒）
- 令 *n* 个进程同时运行，然后在全部进程都结束后结束程序的运行。各个进程按照以下要求运行
  - 在消耗 **total** 毫秒的 CPU 时间后结束运行
  - 每 **resol** 毫秒记录一次以下 3 个数值：①每个进程的唯一 ID（0 ~ *n* - 1 的各个进程独有的编号）；②从程序开始运行到记录的时间点为止经过的时间（单位：毫秒）；③进程的进度（单位：%）
  - 在结束运行后，把所有统计信息用制表符分隔并逐行输出

图 4-2 所示为实验程序的动作。