

# 第 5 章 插叙：进程 API

## 补充：插叙

本章将介绍更多系统实践方面的内容，包括特别关注操作系统的 API 及其使用方式。如果不关心实践相关的内容，你可以略过。但是你应该喜欢实践内容，它们通常在实际生活中有用。例如，公司通常不会因为不实用的技能而聘用你。

本章将讨论 UNIX 系统中的进程创建。UNIX 系统采用了一种非常有趣的创建新进程的方式，即通过一对系统调用：`fork()`和 `exec()`。进程还可以通过第三个系统调用 `wait()`，来等待其创建的子进程执行完成。本章将详细介绍这些接口，通过一些简单的例子来激发兴趣。

## 关键问题：如何创建并控制进程

操作系统应该提供怎样的进程来创建及控制接口？如何设计这些接口才能既方便又实用？

## 5.1 `fork()`系统调用

系统调用 `fork()`用于创建新进程[C63]。但要小心，这可能是你使用过的最奇怪的接口<sup>①</sup>。具体来说，你可以运行一个程序，代码如图 5.1 所示。仔细看这段代码，建议亲自键入并运行！

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {          // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) { // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {              // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17                rc, (int) getpid());
```

<sup>①</sup> 好吧，我们承认我们并不确定。谁知道你在没人的时候调用过什么？但 `fork()`相当奇怪，不管你的函数调用模式有多不同。

```
18     }
19     return 0;
20 }
```

图 5.1 调用 fork() (p1.c)

运行这段程序 (p1.c)，将看到如下输出：

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

让我们更详细地理解一下 p1.c 到底发生了什么。当它刚开始运行时，进程输出一条 hello world 信息，以及自己的进程描述符 (process identifier, PID)。该进程的 PID 是 29146。在 UNIX 系统中，如果要操作某个进程 (如终止进程)，就要通过 PID 来指明。到目前为止，一切正常。

紧接着有趣的事情发生了。进程调用了 fork() 系统调用，这是操作系统提供的创建新进程的方法。新创建的进程几乎与调用进程完全一样，对操作系统来说，这时看起来有两个完全一样的 p1 程序在运行，并都从 fork() 系统调用中返回。新创建的进程称为子进程 (child)，原来的进程称为父进程 (parent)。子进程不会从 main() 函数开始执行 (因此 hello world 信息只输出了一次)，而是直接从 fork() 系统调用返回，就好像是它自己调用了 fork()。

你可能已经注意到，子进程并不是完全拷贝了父进程。具体来说，虽然它拥有自己的地址空间 (即拥有自己的私有内存)、寄存器、程序计数器等，但是它从 fork() 返回的值是不同的。父进程获得的返回值是新创建子进程的 PID，而子进程获得的返回值是 0。这个差别非常重要，因为这样就很容易编写代码处理两种不同的情况 (像上面那样)。

你可能还会注意到，它的输出不是确定的 (deterministic)。子进程被创建后，我们就需要关心系统中的两个活动进程了：子进程和父进程。假设我们在单个 CPU 的系统上运行 (简单起见)，那么子进程或父进程在此时都有可能运行。在上面的例子中，父进程先运行并输出信息。在其他情况下，子进程可能先运行，会有下面的输出结果：

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

CPU 调度程序 (scheduler) 决定了某个时刻哪个进程被执行，我们稍后将详细介绍这部分内容。由于 CPU 调度程序非常复杂，所以我们不能假设哪个进程会先运行。事实表明，这种不确定性 (non-determinism) 会导致一些很有趣的问题，特别是在多线程程序 (multi-threaded program) 中。在本书第 2 部分中学习并发 (concurrency) 时，我们会看到许多不确定性。

## 5.2 wait() 系统调用

到目前为止，我们没有做太多事情：只是创建了一个子进程，打印了一些信息并退出。事实表明，有时候父进程需要等待子进程执行完毕，这很有用。这项任务由 wait() 系统调用

(或者更完整的兄弟接口 `waitpid()`)。图 5.2 展示了更多细节。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) {           // fork failed; exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) { // child (new process)
15        printf("hello, I am child (pid:%d)\n", (int) getpid());
16    } else {               // parent goes down this path (main)
17        int wc = wait(NULL);
18        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19              rc, wc, (int) getpid());
20    }
21    return 0;
22 }

```

图 5.2 调用 `fork()` 和 `wait()` (p2.c)

在 p2.c 的例子中，父进程调用 `wait()`，延迟自己的执行，直到子进程执行完毕。当子进程结束时，`wait()` 才返回父进程。

上面的代码增加了 `wait()` 调用，因此输出结果也变得确定了。这是为什么呢？想想看。（等你想想看……好了）

下面是输出结果：

```

prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>

```

通过这段代码，现在我们知道子进程总是先输出结果。为什么知道？好吧，它可能只是碰巧先运行，像以前一样，因此先于父进程输出结果。但是，如果父进程碰巧先运行，它会马上调用 `wait()`。该系统调用会在子进程运行结束后才返回<sup>①</sup>。因此，即使父进程先运行，它也会礼貌地等待子进程运行完毕，然后 `wait()` 返回，接着父进程才输出自己的信息。

### 5.3 最后是 `exec()` 系统调用

最后是 `exec()` 系统调用，它也是创建进程 API 的一个重要部分<sup>②</sup>。这个系统调用可以让

① 有些情况下，`wait()` 在子进程退出之前返回。像往常一样，请阅读 man 手册获取更多细节。小心本书中绝对的、无条件的陈述，比如“子进程总是先输出结果”或“UNIX 是世界上最好的东西，甚至比冰淇淋还要好”。

② 实际上，`exec()` 有几种变体：`execl()`、`execle()`、`execlp()`、`execv()` 和 `execvp()`。请阅读 man 手册以了解更多信息。

子进程执行与父进程不同的程序。例如，在 p2.c 中调用 fork()，这只是在你想运行相同程序的拷贝时有用。但时，我们常常想运行不同的程序，exec()正好做这样的事（见图 5.3）。

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
    29    107   1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int
8  main(int argc, char *argv[])
9  {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) {          // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc"); // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL;          // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else { // parent goes down this path (main)
24         int wc = wait(NULL);
25         printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                rc, wc, (int) getpid());
27     }
28     return 0;
29 }
```

图 5.3 调用 fork()、wait()和 exec() (p3.c)

在这个例子中，子进程调用 execvp()来运行字符计数程序 wc。实际上，它针对源代码文件 p3.c 运行 wc，从而告诉我们该文件有多少行、多少单词，以及多少字节。

fork()系统调用很奇怪，它的伙伴 exec()也不一般。给定可执行程序名称（如 wc）及需要的参数（如 p3.c）后，exec()会从可执行程序中加载代码和静态数据，并用它覆写自己的代码段（以及静态数据），堆、栈及其他内存空间也会被重新初始化。然后操作系统就执行该程序，将参数通过 argv 传递给该进程。因此，它并没有创建新进程，而是直接将当前运行的程序（以前的 p3）替换为不同的运行程序（wc）。子进程执行 exec()之后，几乎就像 p3.c 从未运行过一样。对 exec()的成功调用永远不会返回。

## 5.4 为什么这样设计 API

当然，你的心中可能有一个大大的问号：为什么设计如此奇怪的接口，来完成简单的、创建新进程的任务？好吧，事实证明，这种分离 `fork()` 及 `exec()` 的做法在构建 UNIX shell 的时候非常有用，因为这给了 shell 在 `fork` 之后 `exec` 之前运行代码的机会，这些代码可以在运行新程序前改变环境，从而让一系列有趣的功能很容易实现。

### 提示：重要的是做对事（LAMPSON 定律）

Lampson 在他的著名论文《Hints for Computer Systems Design》[L83]中曾经说过：“做对事（Get it right）。抽象和简化都不能替代做对事。”有时你必须做正确的事，当你这样做时，总是好过其他方案。有许多方式来设计创建进程的 API，但 `fork()` 和 `exec()` 的组合既简单又极其强大。因此 UNIX 的设计师们做对了。因为 Lampson 经常“做对事”，所以我们就以他来命名这条定律。

shell 也是一个用户程序<sup>①</sup>，它首先显示一个提示符（prompt），然后等待用户输入。你可以向它输入一个命令（一个可执行程序的名称及需要的参数），大多数情况下，shell 可以在文件系统中找到这个可执行程序，调用 `fork()` 创建新进程，并调用 `exec()` 的某个变体来执行这个可执行程序，调用 `wait()` 等待该命令完成。子进程执行结束后，shell 从 `wait()` 返回并再次输出一个提示符，等待用户输入下一条命令。

`fork()` 和 `exec()` 的分离，让 shell 可以方便地实现很多有用的功能。比如：

```
prompt> wc p3.c > newfile.txt
```

在上面的例子中，`wc` 的输出结果被重定向（redirect）到文件 `newfile.txt` 中（通过 `newfile.txt` 之前的大于号来指明重定向）。shell 实现结果重定向的方式也很简单，当完成子进程的创建后，shell 在调用 `exec()` 之前先关闭了标准输出（standard output），打开了文件 `newfile.txt`。这样，即将运行的程序 `wc` 的输出结果就被发送到该文件，而不是打印在屏幕上。

图 5.4 展示了这样做的一个程序。重定向的工作原理，是基于对操作系统管理文件描述符方式的假设。具体来说，UNIX 系统从 0 开始寻找可以使用的文件描述符。在这个例子中，`STDOUT_FILENO` 将成为第一个可用的文件描述符，因此在 `open()` 被调用时，得到赋值。然后子进程向标准输出文件描述符的写入（例如通过 `printf()` 这样的函数），都会被透明地转向新打开的文件，而不是屏幕。

下面是运行 `p4.c` 的结果：

```
prompt> ./p4
prompt> cat p4.output
    32    109    846 p4.c
prompt>
1  #include <stdio.h>
2  #include <stdlib.h>
```

<sup>①</sup> 有许多 shell，如 `tsh`、`bash` 和 `zsh` 等。你应该选择一个，阅读它的 man 手册，了解更多信息。所有 UNIX 专家都这样做。

```
3  #include <unistd.h>
4  #include <string.h>
5  #include <fcntl.h>
6  #include <sys/wait.h>
7
8  int
9  main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) { // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child: redirect standard output to a file
16         close(STDOUT_FILENO);
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc"); // program: "wc" (word count)
22         myargs[1] = strdup("p4.c"); // argument: file to count
23         myargs[2] = NULL; // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else { // parent goes down this path (main)
26         int wc = wait(NULL);
27     }
28     return 0;
29 }
```

图 5.4 之前所有的工作加上重定向 (p4.c)

关于这个输出，你（至少）会注意到两个有趣的地方。首先，当运行 p4 程序后，好像什么也没有发生。shell 只是打印了命令提示符，等待用户的下一个命令。但事实并非如此，p4 确实调用了 fork 来创建新的子进程，之后调用 execvp() 来执行 wc。屏幕上没有看到输出，是由于结果被重定向到文件 p4.output。其次，当用 cat 命令打印输出文件时，能看到运行 wc 的所有预期输出。很酷吧？

UNIX 管道也是用类似的方式实现的，但用的是 pipe() 系统调用。在这种情况下，一个进程的输出被链接到了一个内核管道 (pipe) 上 (队列)，另一个进程的输入也被连接到了同一个管道上。因此，前一个进程的输出无缝地作为后一个进程的输入，许多命令可以用这种方式串联在一起，共同完成某项任务。比如通过将 grep、wc 命令用管道连接可以完成从一个文件中查找某个词，并统计其出现次数的功能：grep -o foo file | wc -l。

最后，我们刚才只是从较高的层面上简单介绍了进程 API，关于这些系统调用的细节，还有更多需要学习和理解。例如，在本书第 3 部分介绍文件系统时，我们会学习更多关于文件描述符的知识。现在，知道 fork() 和 exec() 组合在创建和操作进程时非常强大就足够了。

#### 补充：RTFM——阅读 man 手册

很多时候，本书提到某个系统调用或库函数时，会建议阅读 man 手册。man 手册是 UNIX 系统中最原生的文档，要知道它的出现甚至早于网络 (Web)。

花时间阅读 man 手册是系统程序员成长的必经之路。手册里有许多有用的隐藏彩蛋。尤其是你正在使用的 shell（如 tcsh 或 bash），以及程序中需要使用的系统调用（以便了解返回值和异常情况）。

最后，阅读 man 手册可以避免尴尬。当你询问同事某个 fork 细节时，他可能会回复：“RTFM”。这是他在有礼貌地督促你阅读 man 手册（Read the Man）。RTFM 中的 F 只是为这个短语增加了一点色彩……

## 5.5 其他 API

除了上面提到的 `fork()`、`exec()` 和 `wait()` 之外，在 UNIX 中还有其他许多与进程交互的方式。比如可以通过 `kill()` 系统调用向进程发送信号（`signal`），包括要求进程睡眠、终止或其他有用的指令。实际上，整个信号子系统提供了一套丰富的向进程传递外部事件的途径，包括接受和执行这些信号。

此外还有许多非常有用的命令行工具。比如通过 `ps` 命令来查看当前在运行的进程，阅读 man 手册来了解 `ps` 命令所接受的参数。工具 `top` 也很有用，它展示当前系统中进程消耗 CPU 或其他资源的情况。有趣的是，你常常会发现 `top` 命令自己就是最占用资源的，它或许有一点自大狂。此外还有许多 CPU 检测工具，让你方便快速地了解系统负载。比如，我们总是让 MenuMeters（来自 Raging Menace 公司）运行在 Mac 计算机的工具栏上，这样就能随时了解当前的 CPU 利用率。一般来说，对现状了解得越多越好。

## 5.6 小结

本章介绍了在 UNIX 系统中创建进程需要的 API：`fork()`、`exec()` 和 `wait()`。更多的细节可以阅读 Stevens 和 Rago 的著作 [SR05]，尤其是关于进程控制、进程关系及信号的章节。其中的智慧让人受益良多。

## 参考资料

[C63] “A Multiprocessor System Design” Melvin E. Conway  
AFIPS '63 Fall Joint Computer Conference  
New York, USA 1963

早期关于如何设计多处理系统的论文。文中可能首次在讨论创建新进程时使用 `fork()` 术语。

[DV66] “Programming Semantics for Multiprogrammed Computations” Jack B. Dennis and Earl C. Van Horn  
Communications of the ACM, Volume 9, Number 3, March 1966

一篇讲述多道程序计算机系统基础知识的经典文章。毫无疑问，它对 Project MAC、Multics 以及最终的 UNIX 都有很大的影响。

[L83] “Hints for Computer Systems Design” Butler Lampson

ACM Operating Systems Review, 15:5, October 1983

Lampson 关于计算机系统如何设计的著名建议。你应该抽时间读一读它。

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago Addison-Wesley, 2005

在这里可以找到使用 UNIX API 的所有细节和妙处。买下这本书！阅读它，最重要的是靠它谋生。

### 补充：编码作业

编码作业是小型练习。你可以编写代码在真正的机器上运行，从而获得一些现代操作系统必须提供的基本 API 的体验。毕竟，你（可能）是一名计算机科学家，因此应该喜欢编码，对吧？当然，要真正成为专家，你必须花更多的时间来破解机器。实际上，要找一切借口来写一些代码，看看它是如何工作的。花时间，成为智者，你可以做到的。

## 作业（编码）

在这个作业中，你要熟悉一下刚读过的进程管理 API。别担心，它比听起来更有趣！如果你找到尽可能多的时间来编写代码，通常会增加成功的概率<sup>①</sup>，为什么不现在就开始呢？

## 问题

1. 编写一个调用 `fork()` 的程序。在调用 `fork()` 之前，让主进程访问一个变量（例如  $x$ ）并将其值设置为某个值（例如 100）。子进程中的变量有什么值？当子进程和父进程都改变  $x$  的值时，变量会发生什么？

2. 编写一个打开文件的程序（使用 `open()` 系统调用），然后调用 `fork()` 创建一个新进程。子进程和父进程都可以访问 `open()` 返回的文件描述符吗？当它们并发（即同时）写入文件时，会发生什么？

3. 使用 `fork()` 编写另一个程序。子进程应打印 “hello”，父进程应打印 “goodbye”。你应该尝试确保子进程始终先打印。你能否不在父进程调用 `wait()` 而做到这一点呢？

4. 编写一个调用 `fork()` 的程序，然后调用某种形式的 `exec()` 来运行程序 `/bin/lis`。看看是否可以尝试 `exec()` 的所有变体，包括 `execl()`、`execle()`、`execlp()`、`execv()`、`execvp()` 和 `execvpP()`。

---

<sup>①</sup> 如果你不喜欢编码，但想成为计算机科学家，这意味着你需要变得非常擅长计算机科学理论，或者也许要重新考虑你一直在说的“计算机科学”这回事。

为什么同样的基本调用会有这么多变种？

5. 现在编写一个程序，在父进程中使用 `wait()`，等待子进程完成。`wait()`返回什么？如果你在子进程中使用 `wait()`会发生什么？

6. 对前一个程序稍作修改，这次使用 `waitpid()`而不是 `wait()`。什么时候 `waitpid()`会有用？

7. 编写一个创建子进程的程序，然后在子进程中关闭标准输出（`STDOUT_FILENO`）。如果子进程在关闭描述符后调用 `printf()`打印输出，会发生什么？

8. 编写一个程序，创建两个子进程，并使用 `pipe()`系统调用，将一个子进程的标准输出连接到另一个子进程的标准输入。