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

pid_t fork(void);
```

返回：子进程返回0，父进程返回子进程的PID，如果出错，则为−1。

新创建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同的(但是独立的)一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用fork时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大的区别在于它们有不同的PID。

fork函数是有趣的(也常常令人迷惑)，因为它只被调用一次，却会返回两次：一次是在调用进程(父进程)中，一次是在新创建的子进程中。在父进程中，fork返回子进程的PID。在子进程中，fork返回0。因为子进程的PID总是为非零，返回值就提供一个明确的方法来分辨程序是在父进程还是在子进程中执行。

图8-15展示了一个使用fork创建子进程的父进程的示例。当fork调用在第6行返回时，在父进程和子进程中x的值都为1。子进程在第8行加一并输出它的x的副本。相似地，父进程在第13行减一并输出它的x的副本。

code/ecf/fork.c

```
1   int main()
2   {
3       pid_t pid;
4       int x = 1;
5
6       pid = Fork();
7       if (pid == 0) {  /* Child */
8           printf("child : x=%d\n", ++x);
9           exit(0);
10      }
11
12      /* Parent */
13      printf("parent: x=%d\n", --x);
14      exit(0);
15  }
```

code/ecf/fork.c

图8-15　使用fork创建一个新进程

当在Unix系统上运行这个程序时，我们得到下面的结果：

```
linux> ./fork
parent: x=0
child : x=2
```

这个简单的例子有一些微妙的方面。

- 调用一次，返回两次。fork函数被父进程调用一次，但是却返回两次——一次是返回到父进程，一次是返回到新创建的子进程。对于只创建一个子进程的程序来说，这还是相当简单直接的。但是具有多个fork实例的程序可能就会令人迷惑，需要仔细地推敲了。
- 并发执行。父进程和子进程是并发运行的独立进程。内核能够以任意方式交替执行它们的逻辑控制流中的指令。在我们的系统上运行这个程序时，父进程先完成它的printf语句，然后是子进程。然而，在另一个系统上可能正好相反。一般而言，