


12.1.2 进程的优劣


对于在父、子进程间共享状态信息，进程有一个非常清晰的模型：共享文件表，但是不共享用户地址空间。进程有独立的地址空间既是优点也是缺点。这样一来，一个进程不可能不小心覆盖另一个进程的虚拟内存，这就消除了许多令人迷惑的错误——这是一个明显的优点。

另一方面，独立的地址空间使得进程共享状态信息变得更加困难。为了共享信息，它们必须使用显式的 IPC(进程间通信)机制。(参见下面的旁注。)基于进程的设计的另一个缺点是，它们往往比较慢，因为进程控制和 IPC 的开销很高。

旁注 Unix IPC

在本书中，你已经遇到好几个 IPC 的例子了。第 8 章中的 `waitpid` 函数和信号是基本的 IPC 机制，它们允许进程发送小消息到同一主机上的其他进程。第 11 章的套接字接口是 IPC 的一种重要形式，它允许不同主机上的进程交换任意的字节流。然而，术语 Unix IPC 通常指的是所有允许进程和同一主机上其他进程进行通信的技术。其中包括管道、先进先出(FIFO)、系统 V 共享内存，以及系统 V 信号量(semaphore)。这些机制超出了我们的讨论范围。Kerrisk 的著作[62]是很好的参考资料。

 **练习题 12.1** 在图 12-5 中，并发服务器的第 33 行上，父进程关闭了已连接描述符后，子进程仍然能够使用该描述符和客户端通信。为什么？

 **练习题 12.2** 如果我们要删除图 12-5 中关闭已连接描述符的第 30 行，从没有内存泄漏的角度来说，代码将仍然是正确的。为什么？

12.2 基于 I/O 多路复用的并发编程

假设要求你编写一个 `echo` 服务器，它也能对用户从标准输入键入的交互命令做出响应。在这种情况下，服务器必须响应两个互相独立的 I/O 事件：1)网络客户端发起连接请求，2)用户在键盘上键入命令行。我们先等待哪个事件呢？没有哪个选择是理想的。如果在 `accept` 中等待一个连接请求，我们就不能响应输入的命令。类似地，如果在 `read` 中等待一个输入命令，我们就不能响应任何连接请求。

针对这种困境的一个解决办法就是 I/O 多路复用(I/O multiplexing)技术。基本的思路就是使用 `select` 函数，要求内核挂起进程，只有在一个或多个 I/O 事件发生后，才将控制返回给应用程序，就像在下面的示例中一样：

- 当集合{0, 4}中任意描述符准备好读时返回。
- 当集合{1, 2, 7}中任意描述符准备好写时返回。
- 如果在等待一个 I/O 事件发生时过了 152.13 秒，就超时。

`select` 是一个复杂的函数，有许多不同的使用场景。我们将只讨论第一种场景：等待一组描述符准备好读。全面的讨论请参考[62, 110]。

```
#include <sys/select.h>
```

```
int select(int n, fd_set *fdset, NULL, NULL, NULL);
```

返回已准备好的描述符的非零的个数，若出错则为-1。

```
FD_ZERO(fd_set *fdset);          /* Clear all bits in fdset */
FD_CLR(int fd, fd_set *fdset);    /* Clear bit fd in fdset */
FD_SET(int fd, fd_set *fdset);    /* Turn on bit fd in fdset */
FD_ISSET(int fd, fd_set *fdset);  /* Is bit fd in fdset on? */
```

处理描述符集合的宏。