操作系统实验八报告

智能科学与技术　　2024 年 6 月 6 日

实验任务

试在 Linux 中采用 C 语言模拟实现 Shell 中执行如下命令的程序，"cat test1.txt test2.txt | sort"。(可使用如下系统调用: fork, exec, pipe, dup, open, close, wait 等)

test1.txt 内容：

0

3

5

7

test2.txt 内容：

2

6

9

1

1　实验环境

1. 电脑操作系统：Windows 11 +wsl（Ubuntu）
2. 编程语言：C 语言
3. 辅助工具：

（1）使用 VScode 用于编辑各种文件。

2　实验原理

首先，我们想要模拟实现 Shell 命令，就需要先了解"cat test1.txt test2.txt | sort" 执行时，完成了怎样的过程。从宏观上看，cat 命令读取 test1.txt 和 test2.txt 的内容，并将它们合并成一个文本流。而 sort 命令接收文本流，并对其内容按行进行升序排序。而两个命令通过管道符号“|”连接。在任何一个 shell 中，都可以使用“|”连接两个命令，shell 会将前后两个进程的输入输出用一个管道相连，以便达到进程间通信的目的。管道本质上就是相当于一个“文件”，前面的进程以写方式打开文件，后面的进程以读方式打开。这样前面写完后面读，于是就实现了通信。因此"cat test1.txt test2.txt | sort" 实现了将 test1.txt test2.txt 合并后的输出通过管道作为 sort 的输入，最终 sort 输出排序后的结果。事实上，将脚本"cat test1.txt test2.txt | sort" 转换为以上进程并执行相关操作是由 Shell 进程进行的。

shell 解释执行键盘命令的过程中，首先会分析命令及参数，然后 fork 拷贝父进程 pcb 相关资源，为命令执行做准备，最后调用 exec 在 fork 得到的子进程中执行命令。

通过以上已知，可以知道 shell 在解释执行"cat test1.txt test2.txt | sort" 时，创建了 cat 和 sort 两个子进程，并且通过管道来进行两个进程的信息交流。解析出这一过程后，再根据实验任务的提示，可以

通过 fork, exec, pipe, dup, open, close, wait 等系统调用模拟实现 Shell 解释执行"cat test1.txt test2.txt | sort" 的过程。在此之前，我们必须先了解一下以上系统调用的使用与功能。

2.1 pipe 系统调用

管道（Pipe）是一种用于进程间通信（IPC）的简单而有效的方式。在 UNIX 和类 UNIX 操作系统（如 Linux）中，管道提供了一种让一个进程将其输出发送给另一个进程的输入的机制。管道通常用于数据流的单向传输。在底层，管道其实是一个由操作系统内核维护的缓冲区。一个进程向管道的一端（写端）写入数据，而另一个进程可以从管道的另一端（读端）读取数据。从用户态的角度来看，管道其实就是一对文件描述符，其中一个用于读，另一个用于写。

在 UNIX-like 操作系统中，pipe() 系统调用用于创建一个新的管道，这是一种允许两个进程进行单向数据传输的 IPC（进程间通信）机制。一个进程写入管道的一端，而另一个进程从管道的另一端读取。

管道是通过以下函数原型创建的：

```
1 #include <unistd.h>
2 int pipe(int pipe_fd[2]);
```

这个函数创建了一个管道，并将两个文件描述符存储在 pipe_fd 数组中，其中 pipe_fd[0] 用于从管道读取数据，pipe_fd[1] 用于向管道写入数据。

2.2 dup 系统调用

dup() 系统调用用于复制文件描述符，并返回一个新的文件描述符。这个新的文件描述符与原来的文件描述符指向同一个文件表项，因此它们共享同一个文件偏移量和文件状态标志。这个调用通常用于重定向标准输入、标准输出和标准错误流，以及在进程间传递文件描述符。

当调用 dup(oldfd) 时，系统会找到当前未使用的最小的文件描述符，然后将 oldfd 所指向的文件表项复制到这个新的文件描述符中。然后，dup() 会返回这个新的文件描述符。这个过程中，不会创建新的文件或改变文件的状态，只是创建了一个新的文件描述符，与原有的文件描述符共享文件表项。

2.3 文件描述符

在以上的分析中，出现了文件描述符 fd 这个比较关键的字眼。通过查询资料，我们可以得到以下信息：文件描述符表是一个记录了当前进程打开的文件描述符与文件之间关联的数据结构。在 UNIX 和类 UNIX 系统中，每个进程都有一个文件描述符表，用于跟踪它所打开的文件。

文件描述符表前三个文件通常分别是：

1. 标准输入（stdin）：文件描述符为 0。它通常与进程的标准输入设备关联，即键盘。程序可以从标准输入读取数据。

2. 标准输出（stdout）：文件描述符为 1。它通常与进程的标准输出设备关联，即屏幕。程序可以向标准输出输出数据。

3. 标准错误（stderr）：文件描述符为 2。它通常与进程的标准错误设备关联，即屏幕。程序可以将错误消息输出到标准错误，以便及时发现程序运行中的错误。

这三个文件描述符在程序中经常被使用，它们提供了与用户交互和输出信息的标准方式。标准输入、标准输出和标准错误通常在启动进程时会被自动打开，并且可以在程序中直接使用，无需额外操

作。

文件描述符的作用是维护进程中打开的文件的状态，包括文件描述符和与之相关的文件表项。通过文件描述符，进程可以对文件进行读写操作，并通过文件符号表来跟踪这些操作。

因此，我们可以通过改变标准输入和标准输出文件描述符的文件表项指向来重定位文件的输出与输入源，从而将两个进程的数据输入或者输出与管道相连，从而实现数据交互。

3 实验过程

根据实验原理，我们将实验分成三部分，创建管道，创造 cat 子进程，和创建 sort 子进程。其中创建管道的方法与实验原理中的一致，重点在于创建子进程，并将标准输入文件和标准输出文件重定位到管道的两端，从而实现 cat 的输出作为 sort 的输入的信息流通。

3.1 创建管道

```
int pipe_fd[2];
    pid_t cat_pid, sort_pid;

    // 创建管道
    if (pipe(pipe_fd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
```

3.2 创建 cat 子进程

在这个过程中，我们使用 close(STDOUT_FILENO) 关闭 cat 子进程的标准输出文件，这样标准输出文件描述符此时为当前未使用的最小的文件描述符，再使用 dup(pipe_fd[1]) 便可把标准输出文件重定向标准输出到管道的输入端，然后 execlp 执行“cat test1.txt test2.txt”得到的输出结果就会进入管道流向管道的输出端。

```
// 创建第一个子进程来执行 'cat'
    if ((cat_pid = fork()) == -1) {
        exit(EXIT_FAILURE);
    }

    if (cat_pid == 0) {
        // 子进程1 (cat)
        // 关闭管道的输出，因为fork出的子进程会也拥有两个□件描述符指向同□管道，管道只
            支持单向通信，所以两个进程只能使用一个管道口，必须关闭另一个
        close(pipe_fd[0]);

        // 重定向标准输出到管道的输入端
        close(STDOUT_FILENO);
        dup(pipe_fd[1]);
```

```
14
15          // 关闭多余的文件描述符
16          close(pipe_fd[1]);
17
18          // 执行 “cat test1.txt test2.txt”
19          execlp("cat", "cat", "test1.txt", "test2.txt", (char *)NULL);
20      }
```

3.3 创建 sort 子进程

在这个过程中，我们使用 close(STDIN_FILENO) 关闭 sort 子进程的标准输入文件，这样标准输入文件描述符此时为当前未使用的最小的文件描述符，再使用 dup(pipe_fd[0]) 便可把标准输入文件重定向标准输出到管道的输出端，然后 execlp 执行“sort”会从管道的输出端得到输入数据。

```
1   // 创建第二个子进程来执行 "sort"
2       if ((sort_pid = fork()) == -1) {
3           exit(EXIT_FAILURE);
4       }
5
6       if (sort_pid == 0) {
7           // 子进程2 (sort)
8           // 关闭管道的写入端
9           close(pipe_fd[1]);
10
11          // 重定向标准输入到管道的读取端
12          close(STDIN_FILENO);
13          dup(pipe_fd[0]);
14
15          // 关闭多余的文件描述符
16          close(pipe_fd[0]);
17
18          // 执行 "sort"
19          execlp("sort", "sort", (char *)NULL);
20
21      }
```

4 实验结果

如图，输入以下命令行后得到了正确的排序结果：

```
1   gcc main.c -o main
2   ./main
```

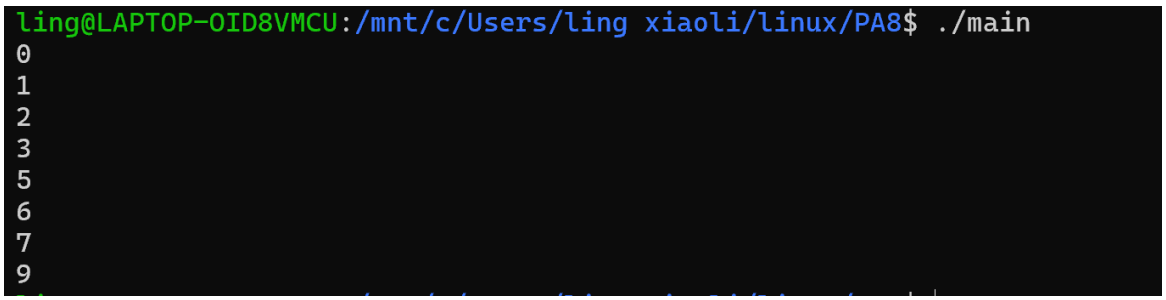

Figure 1: 运行结果

5 实验中的困难与解决

在实验刚开始时，在创建子进程的时候，由于查询资料不全面，或者说没有考虑全面 fork 的本质，在重定向 dup(pipe_fd[1]) 前，并没有进行进程没有用到的管道端的关闭 (以 cat 子进程为例：close(pipe_fd[0]))，导致一直没有输出。后来查询资料发现：

当一个进程（父进程）调用 fork() 创建子进程时，子进程会继承父进程的文件描述符表。这意味着 pipe_fd[0] 和 pipe_fd[1] 在子进程中的值将与父进程中的值相同，它们指向同一个管道。

具体地说，父进程和子进程将拥有指向同一个内核管道对象的文件描述符。这使得父子进程可以通过这个管道进行通信。

但是因为子进程继承了父进程的文件描述符，所以在子进程中，pipe_fd[0] 仍然是管道的读端，pipe_fd[1] 仍然是管道的写端。这就是为什么在创建管道和 fork() 之后，需要 close() 调用关闭其中一端：每个进程通常只需要管道的一端，所以需要关闭不需要的那一端。这样做有助于避免潜在的死锁和资源泄漏。

6 实验的其他探究

6.1 命令行管道的使用限制容量

在查询实验相关资料时了解到，管道本质上是内核的一块缓存，内核维护了一块缓冲区与管道文件相关联，对管道文件的操作，被内核转换成对这块缓冲区内存的操作。既然是文件，那么必然有大小限制，出于对其限制的好奇，不断增加 test1.txt 和 test2.txt 的数字，却仍旧成功，疑惑是否没有限制或者容量比较大？后查询资料，得知了以下命令行可以查询管道的最大容量的限制：

```
ulimit -a
```

结果为“pipe size (512 bytes, -p) 8”，得到最大限制为 8*512 个字节。

6.2 dup2

在开始实验时，是按照老师的思路去利用 dup 会寻找当前未使用的最小的文件描述符的特点，而恰好我们需要重定向的标准输出输入文件的文件描述符属于最小的两个，这种人为利用系统调用特点的思路是非常巧妙的，但同时也意味着局限性。假如我们需要将一个特定文件重定向到目标文件该如何做（也就是我们指定 newold 的 fd 而不是自动分配）。猜想必然存在这种机制，通过查询资料，查询到了 dup2 系统调用。

dup2 函数的语法如下：

```
int dup2(int oldfd, int newfd);
```

其中，oldfd 表示待复制的文件描述符，newfd 表示新的文件描述符。该函数返回新的文件描述符，若出错则返回-1。dup2 函数的工作原理相对简单，主要包括以下几个步骤：

1. 检查 oldfd 和 newfd 是否相等，若相等则不进行任何操作，直接返回 newfd；检查 newfd 的合法性，若已经打开，则先关闭；

2. 复制 oldfd 的文件表项到 newfd，使得两者指向同一个文件表项；

3. 返回 newfd。

这样使用 dup2 便可以实现自定 newfd，在本实验中，可以等价使用以下代码，可以得到一样的结果：

```
// 重定向标准输出到管道的输入端
dup2(pipe_fd[1], STDOUT_FILENO);
```

7 参考文章

1.深入解析 dup2 函数

2.linux 管道的大小

3.Linux- pipe() 系统调用