南京理工大学经济管理学院

课程作业

课程名称：操作系统原理

小组题目：实验十一 Linux 进程间的通信

姓　　名：陈霜澜　　学号：916107390103

姓　　名：李佩尧　　学号：916107390105

姓　　名：李文敬　　学号：916107390106

目录

1 问题描述

用 C 语言编写一个程序，使其用管道实现父子进程间通信。在程序中建立一个管道，同时父进程生成一个子进程，子进程向管道中写入字符串“is sending a message to parent!”；父进程从管道中读出该字符串，并显示到屏幕上。通过一系列操作，实现子进程向父进程发送消息，父进程接受子进程发来的消息，然后终止的过程。

2 工作原理

2.1 相关概念

管道通信方式也称为共享文件(shared file)通信机制，是进程高级通信方式的一种。在该机制中，发送进程以字符流形式把大量数据送入管道（连接读写进程的一个特殊的共享文件），接收进程从管道中接收数据，实质上是利用外存进行数据通信。

2.2 Pipe 的建立和使用

pipe 文件在使用之前，必须先由使用者建立并打开，管道由程序调用 pipe 函数来创建，格式为：int pipe(int fd[2])，pipe 函数会建立管道,并将文件描述词由 fd[2]数组返回,参数数组 fd[2]包含 pipe 使用的两个文件的描述符，fd[0]为读管道，fd[1]为写管道。若创建成功，返回 0，否则返回-1。

建立 pipe 的主要工作是在系统打开文件表中建立该 pipe 的两个系统文件表目，一个表目用于控制该 pipe 的写操作（写入端），另一表目用于控制该 pipe 的读操作（读出端）：

(1)系统文件 write(fd[1],buf,size)

功能：把 buf 中的长度为 size 字符的消息送入管道入口 fd[1]

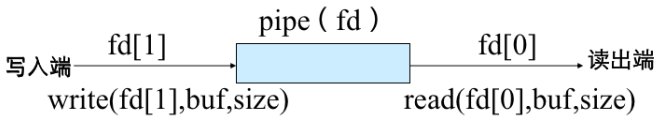
(2)系统文件 read(fd[0],buf,size)

功能：从 pipe 出口 fd[0]读出 size 字符的消息置入 buf 中

2.3 具体实现

发送进程利用文件系统的系统调用 write(fd[1],buf,size)，将 buf 中的长度为

size 个字符的消息送入管道入口 fd[1]；接收进程则使用系统调用 read(fd[0],buf,size)从管道出口 fd[0]读出 size 个字符的消息置入 buf 中。



3 详细设计

3.1 系统函数说明

注：以下函数按在代码中的出现顺序排序

（1）pipe(fd)

调用 pipe 函数建立管道，使用 fd 数组接收文件描述词。若建立成功，则返回 0，否则返回-1。

（2）fork( )

使用 fork 系统调用，从已存在进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

fork 调用一次，返回两次，两次返回分别带回父子进程各自的返回值。若子进程创建成功，则父进程中的返回值是子进程的进程号（id 号），子进程中的返回值为 0；若子进程创建失败，则返回值为-1。因此，可以通过返回值来判定该进程是父进程还是子进程。

使用 while 循环判断(x=fork( ))==-1，即为判断子进程是否创建失败，若子进程创建失败，则一直循环该创建过程，保证子进程创建成功。

（3）int sprintf(char *buffer, const char *format, [argument]...)

sprintf 函数把格式化的数据写入某个字符，在子进程中执行 sprintf(buf,"is sending a passage to parent!")，实现将字符串写入 buf 中。

（4）write(fd[1],buf,size)

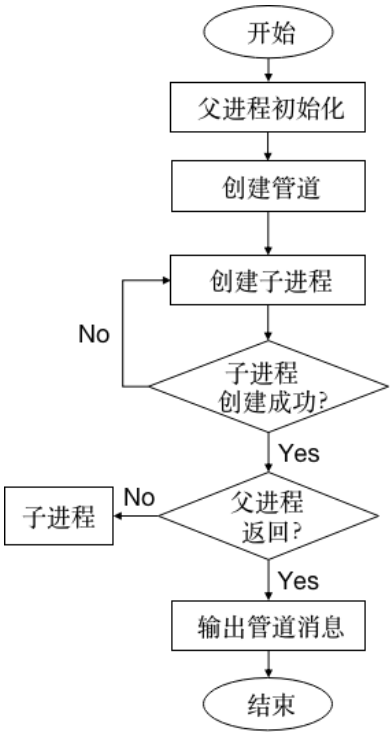
代码 write(fd[1],buf,50)实现将 buf 中的长度为 50 的消息送入管道入口 fd[1]中。

（5）read(fd[0],buf,size)

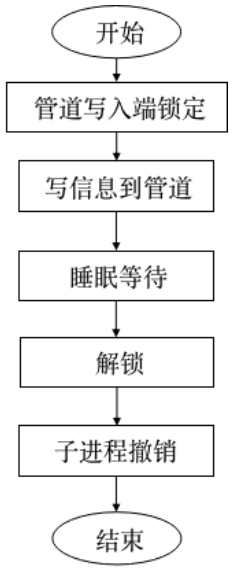
代码 read(fd[0],s,50)实现从 pipe 出口 fd[0]读出 50 字符的消息，并置入 buf 中。

3.2 程序流程图

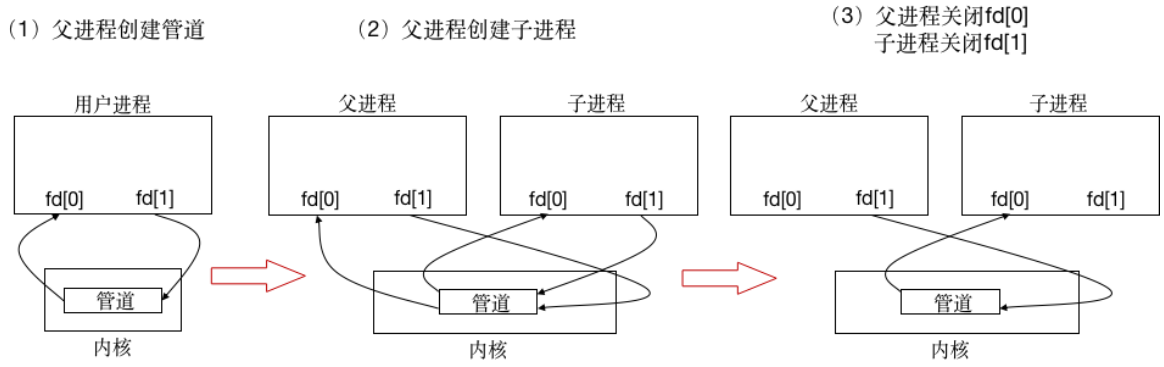
父进程的程序流程图：



子进程的程序流程图：



程序运行图：



(1)父进程创建管道，得到两个文件描述符指向管道的两端

(2)父进程 fork 出子进程，子进程也有两个文件描述符指向同一管道。

(3)父进程关闭 fd[0],子进程关闭 fd[1]，即父进程关闭管道读端,子进程关闭管道写端（因为管道只支持单向通信）。父进程可以往管道里写,子进程可以从管道里读,管道是用环形队列实现的,数据从写端流入从读端流出,这样就实现了进程间通信。

3.3 代码展示

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

main()
{
    int x,fd[2];
    char buf[50],s[50];
    pipe(fd);
    while((x=fork())==-1);
    if(x==0)
    {
        printf("调用子进程，子进程 ID:%d\n",getpid());
        sprintf(buf,"is sending a passage to parent!");
        write(fd[1],buf,50);
                printf("子进程已将消息写入管道！\n");
        exit(0);
    }
    else
    {   printf("调用父进程，父进程 ID：:%d\n",getpid());
        wait(0);
        printf("返回父进程\n");
        read(fd[0],s,50);
        printf("%s%s\n","父进程读取到的消息：",s);
    }
}
```

4 运行结果

在终端中输入命令“./pipe”（pipe 为可执行文件名），运行程序，得到的输出结果如下：

```
调用父进程，父进程ID:11905
调用子进程，子进程ID:11906
子进程已将消息写入管道！
返回父进程
父进程读取到的消息：is sending a passage to parent!
```

5 结果分析

程序调用 fork 函数[1]后，会创建与原进程（除 PID 外）完全相同的子进程，这个子进程将从 fork 后的代码，即 if 条件语句开始执行。父子进程会分别将 if…else…语句执行一遍，因为在两个进程中，fork 的返回值不同，所以条件的匹配结果不同，执行的代码块也不一样。在子进程中，返回值为 0，因此执行 if 后面的语句块，向管道中写入消息；在父进程中返回值为新创建子进程的进程 ID，是一个大于 0 的整数，因此执行 else 后面的语句块，从管道中读出消息。

理论上来说在创建子进程后，父子进程的执行没有固定的先后顺序，但经过多次重复试验，我们发现每次的输出结果都与上图一致，即先执行父进程再执行子进程，这是由 Linux 的进程调度策略决定的。Linux 采用完全公平调度[2]（Completely Fair Scheduler，CFS），用户创建的普通进程都采用 CFS 调度策略，对于该策略有一个控制选项：/proc/sys/kernel/sched_child_runs_first，此值默认为 0，表示父进程优先获得调度，如果将该值改为 1，则将优先调度子进程。在此次试验中该值为默认值 0，因此会出现总是先执行父进程再执行子进程的结果。

由运行结果可以看到，父进程没有全部执行完成就调用了子进程，等待子进程将消息写入管道后才继续执行父进程，将消息从管道中读出。原因是我们在父进程的代码中添加了 wait 函数[3]，用于保证在子进程成功将消息写入管道后，父进程才执行后面的代码，从管道中读取消息。具体的原理为：父进程在调用了 wait 后，立即将自己阻塞起来，由 wait 自动分析子进程是否已经退出，若子进程正常退出，则由阻塞态转为就绪态，再由 CPU 调度继续运行；若没有子进程退出，父进程将会一直阻塞在这里。在本实验中，父进程执行到 wait 时进入阻塞状态，

等待子进程向管道中写消息并退出，然后父进程再被调度，从管道中读消息，实现了父子进程的同步与通信。

6 调试步骤

6.1 结果异常调试

实验过程中，字符串“is sending a message to parent!”以字符串形式存储在数组中，由于最初仅为 buf 数组申请了 30 个字符空间，在程序执行时上述字符串被截断，末尾的“!”没有写入数组。程序运行后的输出结果为“is sending a message to parent”。分析了运行结果之后，我们为 buf 数组分配了 50 个字符空间，并把 write、read 函数中限制向管道中写入、读出的字符数量也改为 50。再次运行程序，程序正确输出“is sending a message to parent!”。

6.2 gdb 调试

gdb 调试的主要目的是通过程序的单步执行，观察父进程和子进程在 Linux 系统下的运行次序，分析其运行结果。

(gdb) gcc -g pipe.c -o pipe　参数-g 用于产生符号调试工具必要的符号信息，是调试源代码必须的选项。

(gdb) gdb pipe　进入调试状态，参数-q 表示不打印版本版权之类的信息[4]。

```
[root@localhost gcc]# gdb -q pipe
Reading symbols from /home/shuanglan/gcc/pipe...done.
```

(gdb) list 显示源代码中 10 行内容。

```
(gdb) list
warning: Source file is more recent than executable.
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <unistd.h>
4
5       main()
6       {
7               int x,fd[2];
8               char buf[50],s[50];
9               pipe(fd);
10              while((x=fork())==-1);
```

（1）子进程正常运行，调试父进程

(gdb) break 11 在行号 11 处设置断点。while((x=fork())==-1)执行后，if(x==0)执行前，程序中断。

```
(gdb) break 11
Breakpoint 1 at 0x400783: file pipe.c, line 11.
```

(gdb) run 运行程序，利用 x=fork()成功创建子进程后，程序在断点处中断。在 fork 创建的多进程环境中，gdb 默认情况下只能在父进程中单步调试。此时输出“Detaching after fork from child process 16014”，调试器跳转到子进程的代码段后执行调试操作[5]。

```
(gdb) run
Starting program: /home/shuanglan/gcc/pipe
Detaching after fork from child process 16014.
调用子进程，子进程ID：16014
子进程已将消息写入管道！

Breakpoint 1, main () at pipe.c:11
11          if(x==0)
```

(gdb) next 执行源代码中的下一条指令，便于观察每一行代码的执行效果。调试结果表明：父进程部分代码正确地执行，父进程运行无误。

```
(gdb) next
20              {       printf("调用父进程，父进程ID：%d\n",getpid());
(gdb) next
调用父进程，父进程ID：16010
21                      wait(0);
(gdb) next
22                      printf("返回父进程\n");
(gdb) next
返回父进程
23                      read(fd[0],s,50);
(gdb) next
24                      printf("%s%s\n","父进程读取到的消息：",s);
(gdb) next
父进程读取到的消息：is sending a passage to parent!
26      }
(gdb) next
__libc_start_main (main=0x40075d <main>, argc=1, argv=0x7fffffffe038,
    init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>,
    stack_end=0x7fffffffe028) at ../csu/libc-start.c:300
300       exit (result);
```

(2)子进程处于暂停状态，调试父进程

set follow-fork-mode [parent | child]

用于设置被调试的进程，默认情况下是 parent。若想调试子进程，将参数设置为 child。

set detach-on-fork [on | off]

on：只调试父进程或子进程中的一个(根据 follow-fork-mode 来决定)，这是默认的模式。

off: 父子进程都在 gdb 的控制之下，其中一个进程正常调试(根据 follow-fork-mode 来决定),另一个进程会被设置为暂停状态[6]。

(gdb) set detach-on-fork off

(gdb) set follow-fork-mode parent

(gdb) break 11 设置断点，调试父进程，子进程处于暂停运行状态。

(gdb) run

(gdb) n 在断点后，单步执行程序。

```
(gdb) set detach-on-fork off
(gdb) set follow-fork-mode parent
(gdb) break 11
Breakpoint 1 at 0x400783: file pipe.c, line 11.
(gdb) run
Starting program: /home/shuanglan/gcc/pipe
[New process 16698]

Breakpoint 1, main () at pipe.c:11
11              if(x==0)
(gdb) n
20              {       printf("调用父进程，父进程ID：%d\n",getpid());
(gdb) n
调用父进程，父进程ID：16694
21                      wait(0);
(gdb) n
```

从运行结果中发现，执行 wait(0)时，父进程阻塞。因为子进程没有被调度执行，所以父进程没能找到一个正常退出的子进程，wait(0)返回值为假，父进程无法继续执行。只有当子进程被调度执行并正常退出后，父进程才能从等待态变为运行态。

（3）父进程正常运行，调试子进程

(gdb) set detach-on-fork on 只调试子进程或父进程中的一个，不被调试的进程正常执行。

(gdb) set follow-fork-mode child 调试子进程，gdb 在子进程中单步调试。

(gdb) break 11 将断点位置设置在子进程调度前。

```
(gdb) set detach-on-fork on
(gdb) set follow-fork-mode child
(gdb) break 11
Breakpoint 3 at 0x400783: /home/shuanglan/gcc/pipe.c:11. (2 locations)
(gdb) run
Starting program: /home/shuanglan/gcc/pipe
[New process 10279]
调用父进程，父进程ID：10278
[Switching to process 10279]

Breakpoint 3, main () at gcc/pipe.c:11
11              if(x==0)
```

从运行结果看出，父进程在 wait(0)阻塞后，进程调度程序才转向子进程。父进程 ID 是 10278，子进程 ID 是 10279。

(gdb) next 单步执行子进程，观察子进程的运行结果。

```
(gdb) next
13                      printf("调用子进程，子进程ID：%d\n",getpid());
(gdb) next
调用子进程，子进程ID：10279
14                      sprintf(buf,"is sending a passage to parent!");
```

```
(gdb) next
15                          write(fd[1],buf,50);
(gdb) next
16                          printf("子进程已将消息写入管道！\n");
(gdb) next
子进程已将消息写入管道！
17                          exit(0);
```

从运行结果看出，子进程部分的代码均被正确执行，子进程运行无误。

(gdb) next 子进程执行 exit(0)并退出，进程调度程序转向父进程。父进程在 wait(0)处能顺利执行，父进程从等待态变为运行态，父进程正常运行直至结束。

```
(gdb) next
返回父进程
[Inferior 3 (process 10279) exited normally]
父进程读取到的消息：is sending a passage to parent!
(gdb) next
The program is not being run.
(gdb)
```

（4）总结

在 fork 创建的多进程环境中，gdb 调试器默认调试父进程，且一次只调试一个进程。当我们想调试子进程时，要用到 follow-fork-mode 和 detach-on-fork 方法。通过对三次调试结果的对比和分析，我们发现在 Linux 系统中，进程调度程序优先调度父进程，只有在父进程阻塞时，才调度子进程；当子进程运行结束并撤销后，进程调度程序再转向父进程。

7 心得体会

在学习使用 Linux 系统后，我们感受到它与 Windows 具有很大的区别。在 Windows 系统中，我们基本上都是在纯图形界面下来操作使用，几乎用不到命令行；而在 Linux 系统中，虽然也具有 GUI 图形界面，但其许多功能的实现如文件的建立、运行需要用命令行来完成，具有更强大的功能。在刚开始使用 Linux 时，由于对各种命令以及使用方法、格式不了解，使用起来很不方便，但当熟悉掌握了之后，效率将会非常高。

在此次试验中，我们了解了在 Linux 下子进程的创建、父子进程的调度方式，对管道通信有了进一步的认识。子进程由 fork 函数创建，创建后子进程与父进程几乎完全一致，但二者执行的代码块是不一样的，这是因为在子进程和父进程中 fork 函数的返回值不同。关于父子进程的调度顺序，由于 Linux 系统自身的系统调度策略，总是先调度父进程，再调度子进程。父子进程通过管道进行通信，首先需要创建一个管道，管道分为两端，一端为输入端 fd[1]，子进程用 write 从输

入端将消息写入管道，另一端为读出端 fd[0]，父进程用 read 从该端读出管道中的消息。父子进程实行管道通信需要保证他们之间的互斥与同步，即父进程/子进程在对管道进行读/写操作时，另一进程必须等待，并且只有子进程向管道中写入数据后，父进程才能进行读操作，当父进程读取数据完成后，子进程才能继续写入数据。在本实验中由于子进程只需向父进程传送一条消息，我们利用 wait 函数来实现父子进程之间的互斥与同步，当子进程向管道中写入数据并退出时，父进程才会从管道中读取信息。

本次实验虽然难度不高，但我们对实验中涉及到的系统函数等的功能进行了深入的学习和探讨，熟悉和掌握了父子进程间如何通过管道进行通信，对管道通信机制有了更深的认识。

8 小组分工

陈霜澜：负责程序调试，主要利用 gdb 调试器完成调试过程。

李佩尧：负责程序的编写，实现父子进程创建和通信。

李文敬：负责程序的编写，对源代码进行优化，增添了输出语句。

参考文献

[1] 学习园社区. linux 中 fork() 函数详解 [EB/OL]. https://www.cnblogs.com/dongguolei/p/8086346.html, 2017-12-22/2019-1-12.

[2]张昺华. Linux 进程的创建函数 fork()及其 fork 内核实现解析[EB/OL]. https://www.cnblogs.com/sky-heaven/p/8073949.html, 2017-12-20/2019-1-12.

[3] 深蓝工作室. Linux 编程基础之进程等待 wait() 函数 [EB/OL]. https://blog.csdn.net/ghostyu/article/details/8083228, 2012-09-14/2019-1-12.

[4] ghostyu. gdb 参数及命令详解（已整理）core dump 调试[EB/OL]. https://blog.csdn.net/ghostyu/article/details/8083228, 2012-10-17/2019-1-12.

[5]ZK 的博客.【Linux】关于理解 fork() 函数的简单例子[EB/OL]. https://blog.csdn.net/ww1473345713/article/details/51708003, 2016-06-19/2019-1-12.

[6] finding. fork 多进程调试 [EB/OL]. https://blog.csdn.net/fingding/article/details/46459095, 2015-06-11/2019-1-12.