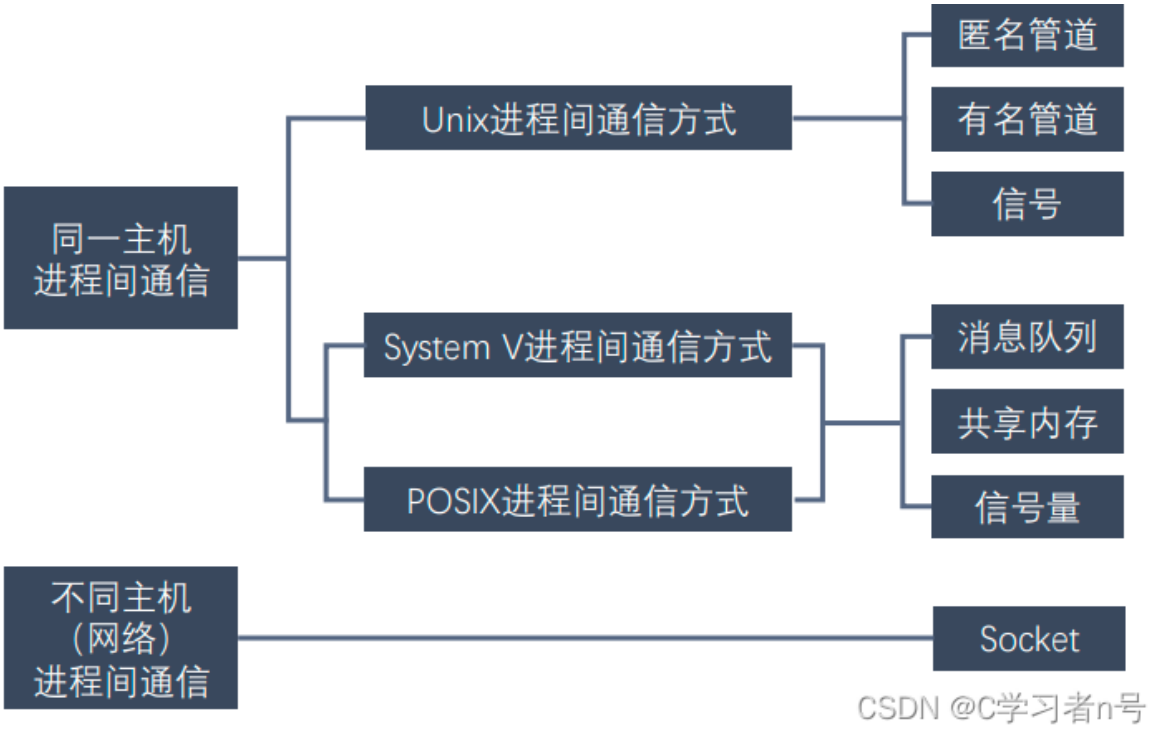
进程间的通信

进程间通信概念

■ 进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。
■ 但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信(IPC：Inter Processes Communication)。
■ 进程间通信的目的：

- 数据传输：一个进程需要将它的数据发送给另一个进程。
- 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
- 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制。
- 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

Linux 进程间通信的方式



通信方式之一匿名管道

概述

■ 管道也叫无名（匿名）管道，它是是 UNIX 系统 IPC（进程间通信）的最古老形式，所有的 UNIX 系统都支持这种通信机制。
■ 统计一个目录中文件的数目命令：ls | wc –l，为了执行该命令，shell 创建了两个进程来分别执行 ls 和 wc。

管道特点

■ 管道其实是一个在内核内存中维护的缓冲器，这个缓冲器的存储能力是有限的，不同的操作系统大小不一定相同。

■ 管道拥有文件的特质：读操作、写操作，匿名管道没有文件实体，有名管道有文件实体，但不存储数据。可以按照操作文件的方式对管道进行操作。

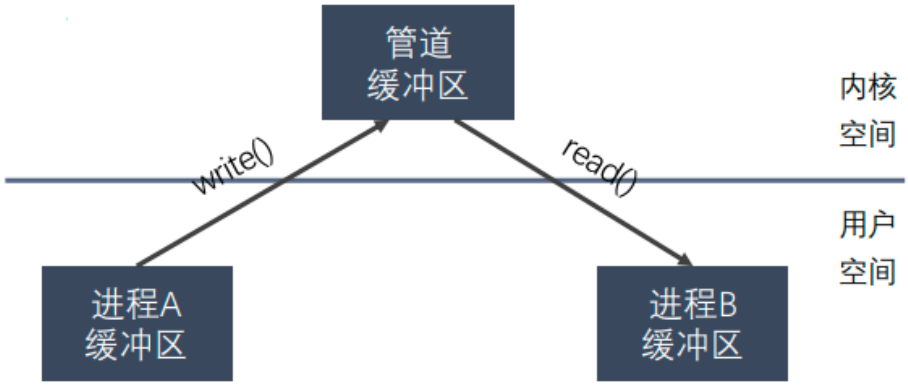
■ 一个管道是一个字节流，使用管道时不存在消息或者消息边界的概念，从管道读取数据的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少。

■ 通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺序是完全一样的。

■ 在管道中的数据的传递方向是单向的，一端用于写入，一端用于读取，`管道是半双工的`。

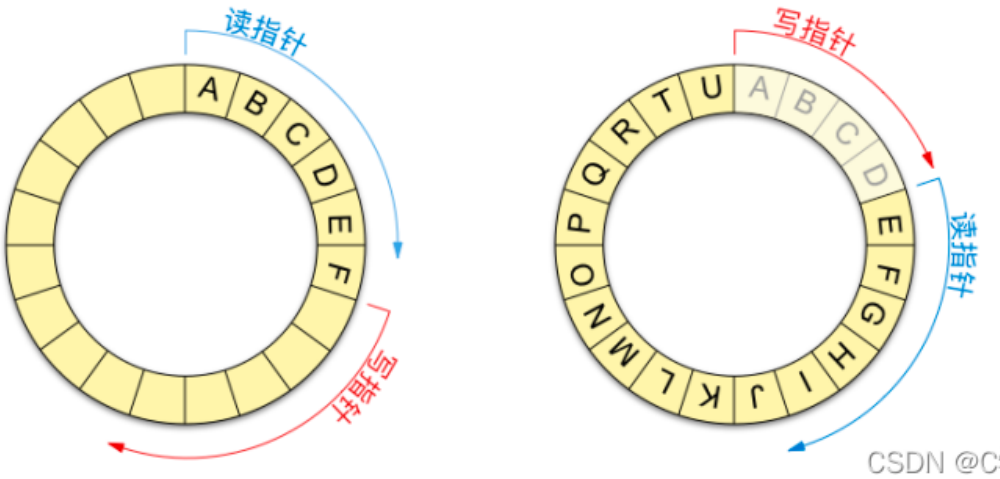
■ 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据，`在管道中无法使用 lseek() 来随机的访问数据`。

■ 匿名管道只能在具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用。



管道数据结构

是一个循环数组，防止溢出。如果超出大小就会覆盖原来的。



函数原型和命令

```
// 创建匿名管道
#include <unistd.h>
int pipe(int pipefd[2]);
/*
功能：创建一个匿名管道，用来进程间通信。
        参数：int pipefd[2] 这个数组是一个传出参数。
            pipefd[0] 对应的是管道的读端
            pipefd[1] 对应的是管道的写端
        返回值：
            成功 0
            失败 -1

    管道默认是阻塞的：如果管道中没有数据，read阻塞，如果管道满了，write阻塞

    注意：匿名管道只能用于具有关系的进程之间的通信（父子进程，兄弟进程）
    */

// 查看管道缓冲大小命令
ulimit -a

// 查看管道缓冲大小函数
#include <unistd.h>
long fpathconf(int fd, int name);
```

查看管道的大小

```
kiko@Hkiko:~/lesson$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31383
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1048576
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31383
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
kiko@Hkiko:~/lesson$
```


实例一 简单的通信

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

int main(int argc, char **argv)
{
```

```
    //要在创建子进程之前得到匿名管道的两个文件描述符
    int fd[2];
    int ret = pipe(fd);

    if (ret == -1)
    {
        perror("pipe");
    }

    //创建子线程
    pid_t pid = fork();

    if (pid > 0)
    {
        close(fd[1]);  //关闭写端
        char buf[1024] = {0};
        int len = read(fd[0], buf, sizeof(buf));
        printf("%s\n", buf);
    }
    else if (pid == 0)
    {
        close(fd[0]);  //关闭读端
        sleep(3);
        char *buf = "123456789";
        write(fd[1], buf, strlen(buf));
    }

    return 0;
}
```

实例二 循环通信

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

int main(int argc, char **argv)
{
    //要在创建子进程之前得到匿名管道的两个文件描述符
    int fd[2];
    int ret = pipe(fd);

    if (ret == -1)
    {
        perror("pipe");
    }

    //创建子线程
    pid_t pid = fork();

    if (pid > 0)
    {
        //父进程，从管道中写入数据
        while (1)
        {
            char buf[1024] = {0};
            int len = read(fd[0], buf, sizeof(buf));
```

```
            printf("%s\n", buf);
        }
    }
    else if (pid == 0)
    {
        //子进程,向管道写入数据
        while (1)
        {
            sleep(2);
            char *buf = "123456789";
            write(fd[1], buf, strlen(buf));
        }
    }

    return 0;
}
```

实例三 函数获取管道大小

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

int main(int argc, char **argv)
{
    int fd[2];

    int ret = pipe(fd);

    long size = fpathconf(fd[0], _PC_PIPE_BUF);

    printf("管道最大值 %ld\n", size);

    return 0;
}
```

```
kiko@Hkiko:~/lesson/myprocess$ ./myfpathconf
管道最大值 4096
```


案例

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wait.h>

/**
    模拟 ps aux | grep root

    子进程： ps aux, 子进程结束后，将数据发送给父进程
    父进程：获取到数据，过滤
    pipe()
    execlp()
    子进程将标准输出 stdout_fileno 重定向到管道的写端。   dup2
```

```c
 */

int main(int argc, char *argv[])
{
    //创建管道
    int fds[2];
    int ret = pipe(fds);

    if (ret == -1)
    {
        perror("pipe");
        exit(1);
    }

    //创建子进程
    pid_t pid = fork();

    if (pid > 0)
    {
        //父进程

        //关闭写端
        close(fds[1]);

        //从管道读取数据
        char buf[1024] = {0};
        int len = -1;

        while ((len = read(fds[0], buf, sizeof(buf))) > 0)
        {
            printf("%s\n", buf);
            memset(buf, 0, sizeof(buf));
        }

        wait(NULL);
    }
    else if (pid == 0)
    {
        //子进程

        //关闭读端
        close(fds[0]);

        //文件描述符重定向 stdout_fileno --> fds[1]
        dup2(fds[1], STDIN_FILENO);

        //执行ps aux命令
        execlp("ps", "ps", "aux", NULL);
    }
    else
    {
        perror("fork");
        exit(1);
    }
}
```

管道读写特点

使用管道时，需要注意以下几种特殊的情况（假设都是`阻塞I/O`操作）

1.所有的指向管道写端的文件描述符都关闭了（管道写端引用计数为0），有进程从管道的读端读数据，那么管道中剩余的数据被读取以后，再次read会返回0，就像读到文件末尾一样。

2.如果有指向管道写端的文件描述符没有关闭（管道的写端引用计数大于0），而持有管道写端的进程也没有往管道中写数据，这个时候有进程从管道中读取数据，那么管道中剩余的数据被读取后，再次read会阻塞，直到管道中有数据可以读了才读取数据并返回。

3.如果所有指向管道读端的文件描述符都关闭了（管道的读端引用计数为0），这个时候有进程向管道中写数据，那么该进程会收到一个信号SIGPIPE, 通常会导致进程异常终止。

4.如果有指向管道读端的文件描述符没有关闭（管道的读端引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道中写数据，那么在管道被写满的时候再次write会阻塞，直到管道中有空位置才能再次写入数据并返回。

```
总结：
    读管道：
        管道中有数据，read返回实际读到的字节数。
        管道中无数据：
            写端被全部关闭，read返回0（相当于读到文件的末尾）
            写端没有完全关闭，read阻塞等待

    写管道：
        管道读端全部被关闭，进程异常终止（进程收到SIGPIPE信号）
        管道读端没有全部关闭：
            管道已满，write阻塞
            管道没有满，write将数据写入，并返回实际写入的字节数
```

设置管道阻塞

```c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
/*
    设置管道非阻塞
    int flags = fcntl(fd[0], F_GETFL);  // 获取原来的flag
    flags |= O_NONBLOCK;            // 修改flag的值
    fcntl(fd[0], F_SETFL, flags);   // 设置新的flag
*/
int main()
{

    // 在fork之前创建管道
    int pipefd[2];
    int ret = pipe(pipefd);
    if (ret == -1)
    {
        perror("pipe");
        exit(0);
    }
```

```
    // 创建子进程
    pid_t pid = fork();
    if (pid > 0)
    {
        // 父进程
        printf("i am parent process, pid : %d\n", getpid());

        // 关闭写端
        close(pipefd[1]);

        // 从管道的读取端读取数据
        char buf[1024] = {0};

        int flags = fcntl(pipefd[0], F_GETFL); // 获取原来的flag
        flags |= O_NONBLOCK;                   // 修改flag的值
        fcntl(pipefd[0], F_SETFL, flags);      // 设置新的flag

        while (1)
        {
            int len = read(pipefd[0], buf, sizeof(buf));
            printf("len : %d\n", len);
            printf("parent recv : %s, pid : %d\n", buf, getpid());
            memset(buf, 0, 1024);
            sleep(1);
        }
    }
    else if (pid == 0)
    {
        // 子进程
        printf("i am child process, pid : %d\n", getpid());
        // 关闭读端
        close(pipefd[0]);
        char buf[1024] = {0};
        while (1)
        {
            // 向管道中写入数据
            char *str = "hello,i am child";
            write(pipefd[1], str, strlen(str));
            sleep(5);
        }
    }
    return 0;
}
```

```
kiko@Hkiko:~/lesson/myprocess$ ./mypipe5
i am parent process, pid : 130147
len : -1
parent recv : , pid : 130147
i am child process, pid : 130148
len : 16
parent recv : hello,i am child, pid : 130147
len : -1
parent recv : , pid : 130147
len : -1
parent recv : , pid : 130147
len : -1
parent recv : , pid : 130147
len : 16
parent recv : hello,i am child, pid : 130147
^C
```


通信方式之二有名管道

概述

■ 匿名管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道（FIFO），也叫命名管道、FIFO文件。
■ 有名管道（FIFO）不同于匿名管道之处在于它提供了一个路径名与之关联，以 FIFO 的文件形式存在于文件系统中，并且其打开方式与打开一个普通文件是一样的，这样即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信，因此，通过 FIFO 不相关的进程也能交换数据。
■ 一旦打开了 FIFO，就能在它上面使用与操作匿名管道和其他文件的系统调用一样的I/O系统调用了（如read()、write()和close()）。与管道一样，FIFO 也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。FIFO 的名称也由此而来：先入先出。
■ 有名管道（FIFO)和匿名管道（pipe）有一些特点是相同的，不一样的地方在于：

1. FIFO 在文件系统中作为一个特殊文件存在，但 FIFO 中的内容却存放在内核区的内存中，和匿名管道一样。
2. 当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。
3. FIFO 有名字，不相关的进程可以通过打开有名管道进行通信。

函数

```
/*
    创建fifo文件
    1.通过命令： mkfifo 名字
    2.通过函数：int mkfifo(const char *pathname, mode_t mode);

    函数原型：
        #include <sys/types.h>
        #include <sys/stat.h>
        int mkfifo(const char *pathname, mode_t mode);
        参数：
            - pathname: 管道名称的路径
            - mode: 文件的权限 和 open 的 mode 是一样的，是一个八进制的数
        返回值：成功返回0，失败返回-1，并设置错误号

*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
```

```c
#include <unistd.h>

int main() {


    // 判断文件是否存在
    int ret = access("fifo1", F_OK);
    if(ret == -1) {
        printf("管道不存在，创建管道\n");

        ret = mkfifo("fifo1", 0664);

        if(ret == -1) {
            perror("mkfifo");
            exit(0);
        }

    }

    return 0;
}
```

管道文件

实例

两个无关的进程，一个读一个写。
写端

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

// 向管道中写数据
/*
    有名管道的注意事项：
        1.一个为只读而打开一个管道的进程会阻塞，直到另外一个进程为只写打开管道
        2.一个为只写而打开一个管道的进程会阻塞，直到另外一个进程为只读打开管道

    读管道：
        管道中有数据，read返回实际读到的字节数
        管道中无数据：
```

```
            管道写端被全部关闭，read返回0，（相当于读到文件末尾）
            写端没有全部被关闭，read阻塞等待

    写管道：
        管道读端被全部关闭，进行异常终止（收到一个SIGPIPE信号）
        管道读端没有全部关闭：
            管道已经满了，write会阻塞
            管道没有满，write将数据写入，并返回实际写入的字节数。
*/
int main() {

    // 1.判断文件是否存在
    int ret = access("test", F_OK);
    if(ret == -1) {
        printf("管道不存在，创建管道\n");

        // 2.创建管道文件
        ret = mkfifo("test", 0664);

        if(ret == -1) {
            perror("mkfifo");
            exit(0);
        }

    }

    // 3.以只写的方式打开管道
    int fd = open("test", O_WRONLY);
    if(fd == -1) {
        perror("open");
        exit(0);
    }

    // 写数据
    for(int i = 0; i < 100; i++) {
        char buf[1024];
        sprintf(buf, "hello, %d\n", i);
        printf("write data : %s\n", buf);
        write(fd, buf, strlen(buf));
        sleep(1);
    }

    close(fd);

    return 0;
}
```

读端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

// 从管道中读取数据
```

```c
int main() {

    // 1.打开管道文件
    int fd = open("test", O_RDONLY);
    if(fd == -1) {
        perror("open");
        exit(0);
    }

    // 读数据
    while(1) {
        char buf[1024] = {0};
        int len = read(fd, buf, sizeof(buf));
        if(len == 0) {
            printf("写端断开连接了...\n");
            break;
        }
        printf("recv buf : %s\n", buf);
    }

    close(fd);

    return 0;
}
```

案例实现相互间通信

server端

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv)
{
    // 1.创建有名管道
    int ret = access("fifo1", F_OK);
    if (ret == -1)
    {
        //创建
        ret = mkfifo("fifo1", 0777);
        printf("管道不存在, 创建管道\n");
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access("fifo2", F_OK);
    if (ret == -1)
    {
        // 文件不存在
```

```
        printf("管道不存在，创建对应的有名管道\n");
        ret = mkfifo("fifo2", 0664);
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    // 2.以只写的方式打开管道fifo1
    int fdw = open("fifo1", O_WRONLY);
    if (fdw == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo1成功，等待写入...\n");

    // 3.以只读的方式打开管道fifo2
    int fdr = open("fifo2", O_RDONLY);
    if (fdr == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo2成功，等待读取...\n");

    char buf[128];
    //循环读写数据
    while (1)
    {
        //写数据
        memset(buf, 0, sizeof(buf));    //清空数据
        fgets(buf, sizeof(buf), stdin); //从标准输入中获取数据

        int size = write(fdw, buf, strlen(buf)); //向管道读端写入数据
        if (size == -1)
        {
            perror("write");
            exit(0);
        }

        //读数据
        memset(buf, 0, sizeof(buf));
        size = read(fdr, buf, sizeof(buf));
        if (size <= 0)
        {
            perror("read");
            break;
        }
        printf("buf: %s\n", buf);
    }
}
```

client端

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

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv)
{
    // 1.判断有名管道文件是否存在
    int ret = access("fifo1", F_OK);
    if (ret == -1)
    {
        // 文件不存在
        printf("管道不存在，创建对应的有名管道\n");
        ret = mkfifo("fifo1", 0664);
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access("fifo2", F_OK);
    if (ret == -1)
    {
        // 文件不存在
        printf("管道不存在，创建对应的有名管道\n");
        ret = mkfifo("fifo2", 0664);
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    // 2.以只读的方式打开管道fifo1
    int fdr = open("fifo1", O_RDONLY);
    if (fdr == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo1成功，等待读取...\n");
    // 3.以只写的方式打开管道fifo2
    int fdw = open("fifo2", O_WRONLY);
    if (fdw == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo2成功，等待写入...\n");

    char buf[128];

    // 4.循环的读写数据
    while (1)
    {
```

```
        // 5.读管道数据
        memset(buf, 0, 128);
        ret = read(fdr, buf, 128);
        if (ret <= 0)
        {
            perror("read");
            break;
        }
        printf("buf: %s\n", buf);

        memset(buf, 0, 128);
        // 获取标准输入的数据
        fgets(buf, 128, stdin);
        // 写数据
        ret = write(fdw, buf, strlen(buf));
        if (ret == -1)
        {
            perror("write");
            exit(0);
        }
    }
}
```

通信升级版

利用父子进程实现一个进程中隔离读写操作。这样可以在一个进程多次连续的读或者写。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv)
{
    // 1.创建有名管道
    int ret = access("fifo1", F_OK);
    if (ret == -1)
    {
        //创建
        ret = mkfifo("fifo1", 0777);
        printf("管道不存在，创建管道\n");
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access("fifo2", F_OK);
    if (ret == -1)
    {
        // 文件不存在
        printf("管道不存在，创建对应的有名管道\n");
        ret = mkfifo("fifo2", 0664);
```

```
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    // 2.以只写的方式打开管道fifo1
    int fdw = open("fifo1", O_WRONLY);
    if (fdw == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo1成功，等待写入...\n");

    // 3.以只读的方式打开管道fifo2
    int fdr = open("fifo2", O_RDONLY);
    if (fdr == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo2成功，等待读取...\n");

    char buf[128];

    pid_t pid = fork();

    //这样就避免了因为一个读操作或者写操作的阻塞而影响到另一个管道的读写。
    if (pid > 0) //父进程负责写
    {
        while (1)
        {
            //写数据
            memset(buf, 0, sizeof(buf));    //清空数据
            fgets(buf, sizeof(buf), stdin); //从标准输入中获取数据

            int size = write(fdw, buf, strlen(buf)); //向管道读端写入数据
            if (size == -1)
            {
                perror("write");
                exit(0);
            }
        }
    }
    else if (pid <= 0) //子进程负责读
    {
        //循环读写数据
        while (1)
        {
            //读数据
            memset(buf, 0, sizeof(buf));
            int size = read(fdr, buf, sizeof(buf));
            if (size <= 0)
            {
                perror("read");
                break;
```

```
            }
            printf("buf: %s\n", buf);
        }
    }
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char **argv)
{
    // 1.判断有名管道文件是否存在
    int ret = access("fifo1", F_OK);
    if (ret == -1)
    {
        // 文件不存在
        printf("管道不存在，创建对应的有名管道\n");
        ret = mkfifo("fifo1", 0664);
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    ret = access("fifo2", F_OK);
    if (ret == -1)
    {
        // 文件不存在
        printf("管道不存在，创建对应的有名管道\n");
        ret = mkfifo("fifo2", 0664);
        if (ret == -1)
        {
            perror("mkfifo");
            exit(0);
        }
    }

    // 2.以只读的方式打开管道fifo1
    int fdr = open("fifo1", O_RDONLY);
    if (fdr == -1)
    {
        perror("open");
        exit(0);
    }
    printf("打开管道fifo1成功，等待读取...\n");
    // 3.以只写的方式打开管道fifo2
    int fdw = open("fifo2", O_WRONLY);
    if (fdw == -1)
    {
        perror("open");
        exit(0);
```

```
    }
    printf("打开管道fifo2成功，等待写入...\n");

    char buf[128];

    pid_t pid = fork();

    if (pid > 0) //父进程负责读操作
    {
        while (1)
        {
            // 5.读管道数据
            memset(buf, 0, 128);
            ret = read(fdr, buf, 128);
            if (ret <= 0)
            {
                perror("read");
                break;
            }
            printf("buf: %s\n", buf);
        }
    }
    else if (pid <= 0)
    {
        // 4.循环的读写数据
        while (1)
        {
            memset(buf, 0, 128);
            // 获取标准输入的数据
            fgets(buf, 128, stdin);
            // 写数据
            ret = write(fdw, buf, strlen(buf));
            if (ret == -1)
            {
                perror("write");
                exit(0);
            }
        }
    }
}
```

通信方式之三内存映射

概述

■ 内存映射（Memory-mapped I/O）是将磁盘文件的数据映射到内存，用户通过修改内存就能修改磁盘文件。

API介绍

函数原型：

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

    void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t
offset);
```

- 功能：将一个文件或者设备的数据映射到内存中
- 参数：
  - void *addr: 传入NULL，由内核指定
  - length ： 要映射的数据的长度，这个值不能为0。建议使用文件的长度。
    获取文件的长度：stat / lseek
  - prot ： 对申请的内存映射区的操作权限
    -PROT_EXEC ：可执行的权限
    -PROT_READ ：读权限
    -PROT_WRITE ：写权限
    -PROT_NONE ：没有权限
    要操作映射内存，必须要有读的权限。
    PROT_READ、PROT_READ|PROT_WRITE
  - flags ：
    - MAP_SHARED ： 映射区的数据会自动和磁盘文件进行同步，进程间通信，必须要设置这个选项
    - MAP_PRIVATE ：不同步，内存映射区的数据改变了，对原来的文件不会修改，会重新创建一个新的文件。（copy on write）
  - fd: 需要映射的那个文件的文件描述符
    - 通过open得到，open的是一个磁盘文件
    - 注意：文件的大小不能为0，open指定的权限不能和prot参数有冲突。port权限《= open权限
      prot: PROT_READ open:只读/读写
      prot: PROT_READ | PROT_WRITE open:读写
  - offset：偏移量，一般不用。必须指定的是4k的整数倍，0表示不便宜。
- 返回值：返回创建的内存的首地址
  失败返回MAP_FAILED，(void *) -1，并设置对应的errno

```
int munmap(void *addr, size_t length);
```

- 功能：释放内存映射
- 参数：
  - addr ： 要释放的内存的首地址
  - length ： 要释放的内存的大小，要和mmap函数中的length参数的值一样。

使用内存映射实现进程间通信：

1.有关系的进程（父子进程）

- 还没有子进程的时候
  - 通过唯一的父进程，先创建内存映射区
- 有了内存映射区以后，创建子进程
- 父子进程共享创建的内存映射区

2.没有关系的进程间通信

- 准备一个大小不是0的磁盘文件
- 进程1 通过磁盘文件创建内存映射区
  - 得到一个操作这块内存的指针
- 进程2 通过磁盘文件创建内存映射区
  - 得到一个操作这块内存的指针
- 使用内存映射区通信

注意：内存映射区通信，是非阻塞。

实例父子进程通信

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
```

```c
#include <string.h>
#include <stdlib.h>
#include <wait.h>

// 作业:使用内存映射实现没有关系的进程间的通信。
int main() {

    // 1.打开一个文件
    int fd = open("test.txt", O_RDWR);
    int size = lseek(fd, 0, SEEK_END);  // 获取文件的大小

    // 2.创建内存映射区
    void *ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(ptr == MAP_FAILED) {
        perror("mmap");
        exit(0);
    }

    // 3.创建子进程
    pid_t pid = fork();
    if(pid > 0) {
        wait(NULL);
        // 父进程
        char buf[64];
        strcpy(buf, (char *)ptr);
        printf("read data : %s\n", buf);

    }else if(pid == 0){
        // 子进程
        strcpy((char *)ptr, "nihao a, son!!!");
    }

    // 关闭内存映射区
    munmap(ptr, size);

    return 0;
}
```

内存映射区注意事项

1.如果对mmap的返回值(ptr)做++操作(ptr++), munmap是否能够成功?
void * ptr = mmap(...);
ptr++; 可以对其进行++操作
munmap(ptr, len); // 错误,要保存地址

2.如果open时O_RDONLY, mmap时prot参数指定PROT_READ | PROT_WRITE会怎样?
错误，返回MAP_FAILED
open()函数中的权限建议和prot参数的权限保持一致。

3.如果文件偏移量为1000会怎样?
偏移量必须是4K的整数倍，返回MAP_FAILED

4.mmap什么情况下会调用失败?
- 第二个参数：length = 0
- 第三个参数：prot
  - 只指定了写权限

- prot PROT_READ | PROT_WRITE
  第5个参数fd 通过open函数时指定的 O_RDONLY / O_WRONLY

5.可以open的时候O_CREAT一个新文件来创建映射区吗?
- 可以的，但是创建的文件的大小如果为0的话，肯定不行
- 可以对新的文件进行扩展
  - lseek()
  - truncate()

6.mmap后关闭文件描述符，对mmap映射有没有影响?
int fd = open("XXX");
mmap(,,,,fd,0);
close(fd);
映射区还存在，创建映射区的fd被关闭，没有任何影响。

7.对ptr越界操作会怎样?
void * ptr = mmap(NULL, 100,,,,,);
4K
越界操作操作的是非法的内存 -> 段错误

实例内存映射区实现文件拷贝

```
// 使用内存映射实现文件拷贝的功能
/*
    思路:
        1.对原始的文件进行内存映射
        2.创建一个新文件（拓展该文件）
        3.把新文件的数据映射到内存中
        4.通过内存拷贝将第一个文件的内存数据拷贝到新的文件内存中
        5.释放资源
*/
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

int main() {

    // 1.对原始的文件进行内存映射
    int fd = open("english.txt", O_RDWR);
    if(fd == -1) {
        perror("open");
        exit(0);
    }

    // 获取原始文件的大小
    int len = lseek(fd, 0, SEEK_END);

    // 2.创建一个新文件（拓展该文件）
    int fd1 = open("cpy.txt", O_RDWR | O_CREAT, 0664);
    if(fd1 == -1) {
        perror("open");
        exit(0);
```

```
    }

    // 对新创建的文件进行拓展
    truncate("cpy.txt", len);
    write(fd1, " ", 1);

    // 3.分别做内存映射
    void * ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    void * ptr1 = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);

    if(ptr == MAP_FAILED) {
        perror("mmap");
        exit(0);
    }

    if(ptr1 == MAP_FAILED) {
        perror("mmap");
        exit(0);
    }

    // 内存拷贝
    memcpy(ptr1, ptr, len);

    // 释放资源
    munmap(ptr1, len);
    munmap(ptr, len);

    close(fd1);
    close(fd);

    return 0;
}
```

匿名映射区（父子进程通信）

```
/*
    匿名映射：不需要文件实体进程一个内存映射
*/

#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {

    // 1.创建匿名内存映射区
    int len = 4096;
    void * ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    if(ptr == MAP_FAILED) {
        perror("mmap");
```

```
        exit(0);
    }

    // 父子进程间通信
    pid_t pid = fork();

    if(pid > 0) {
        // 父进程
        strcpy((char *) ptr, "hello, world");
        wait(NULL);
    }else if(pid == 0) {
        // 子进程，因为映射区非阻塞，所以子进程要先等父进程写入数据
        sleep(1);
        printf("%s\n", (char *)ptr);
    }

    // 释放内存映射区
    int ret = munmap(ptr, len);

    if(ret == -1) {
        perror("munmap");
        exit(0);
    }
    return 0;
}
```

映射区实现两个独立进程的通信

注意事项：

和父子进程通信不同，独立进程的通信完全依赖于借助的文件。另一个进程要读取进程写入的数据，也要打开这个文件，并且映射，所以，文件的大小必须要大于进程写入的数据大小，否则另一个进程打开文件的数据必然是不完整的，甚至可能没有读到数据。而父子进程则可以依赖共享内存实现超过了文本大小的数据读取。

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <wait.h>

int main(int argc, char **argv)
{
    // 1.打开一个文件
    int fd = open("test.txt", O_RDWR);
    //将文件拓展，因为这是要给另一个独立的进程发送数据，如果文件小于要发送的数据，另一个进程就只能读到被文件大小限制后的内容，是不完整的
    truncate("test.txt", 128);
    write(fd, (char *)"1", 1);

    int len = lseek(fd, 0, SEEK_END); // 获取文件的大小

    //创建内存映射区
    void *ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

```
    if (ptr == MAP_FAILED)
    {
        perror("mmap");
        exit(0);
    }

    //向内存中写入数据
    strcpy(ptr, (char *)"你好");

    sleep(10);

    //关闭映射区
    close(fd);
    munmap(ptr, len);

    return 0;
}
```

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <wait.h>

int main(int argc, char **argv)
{
    //文件
    int fd = open("test.txt", O_RDWR);
    int len = lseek(fd, 0, SEEK_END); //文件长度

    //创建内存映射区
    void *ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    if (ptr == MAP_FAILED)
    {
        perror("mmap");
        exit(0);
    }

    //向内存中写入数据
    char buf[128];
    strcpy(buf, ptr);
    printf("%s\n", buf);

    //关闭映射区
    munmap(ptr, len);
}
```