

四、进程通信

4.1 管道

4.1.1 匿名管道

4.1.2 命名管道

练习：两个进程使用 fifo 互发消息聊天

4.2 内存映射

4.3 消息队列

四、进程通信

进程间通信简称IPC(Inter process communication)，进程间通信就是不同进程之间传播或交换信息。由于各个运行进程之间具有独立性，这个独立性主要体现在数据层面，而代码逻辑层面可以私有也可以公有(例如父子进程)，因此各个进程之间要实现通信是非常困难的。各个进程之间若想实现通信，一定要借助第三方资源，这些进程就可以通过向这个第三方资源写入或是读取数据，进而实现进程之间的通信，这个第三方资源实际上就是操作系统提供的一段内存区域。



- 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，
 - 资源共享：多个进程之间共享同样的资源，
 - 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，比如进程终止时需要通知其父进程。(信号)
 - 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

4.1 管道

管道 默认是阻塞的，普通文件默认不会阻塞。

4.1.1 匿名管道

匿名指的是文件名，通过pipe创建的管道文件，在磁盘中是没有实体的文件的。因为没有文件名，到不同的程序中，没有文件名就找不到这个管道文件。因此匿名管道只能在有血缘关系的进程之间通讯（父子进程之间），

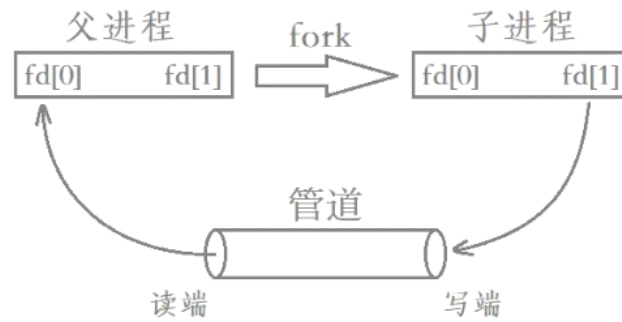
pipe 函数

- 函数描述：
- 创建匿名管道
- 函数原型：
- ```
int pipe(int pipefd[2]);
```
- 函数参数：
- pipefd 是一个传出参数，用于返回两个指向管道读端和写端的文件描述符
- pipefd[0]:管道读端的文件描述符
- pipefd[1]:管道写端的文件描述符

函数返回值:

成功返回 0

失败返回 -1, 设置 errno



### 注意:

管道只能进行单向通信, 因此当父进程创建完子进程后, 需要确认父子进程谁读谁写然后关闭相应的读写端。

从管道写端写入的数据会被存到内核缓冲, 直到从管道的读端被读取。

- (1) 管道中没有数据: write 返回成功写入的字节数, 读端进程阻塞在 read 上
- (2) 管道中有数据没满: write 返回成功写入的字节数, read 返回读取的字节数
- (3) 管道已满: 写端进程阻塞在 write 上, read 返回读取的字节数  
读取一定大小的内容后, 写端才会接着写
- (4) 写端全部关闭: read 正常读, 返回读到的字节数(没有数据返回0, 不阻塞)
- (5) 读端全部关闭: 写端进程 write 会异常终止进程(被信号 SIGPIPE 杀死的)

### 练习

借助管道和 execlp 函数, 实现 ls | wc -l

```
// 练习: 借助管道和 execlp 函数, 实现 ls | wc -l
int main(int argc, char* argv[])
{
 // 创建匿名管道--0 读端、1 写端
 int pipefd[2];
 pipe(pipefd);

 // 创建子进程
 int pid = fork();
 if(pid == 0) // 子进程写入数据
 {
 close(pipefd[0]); // 关闭读端
 dup2(pipefd[1], 1); // 将 标准输出 给到 管道写端
 execlp("ls", "ls", "/", NULL);
 }
 else // 父进程读取数据
 {
 close(pipefd[1]); // 关闭写端
 dup2(pipefd[0], 0); // 将标准输入 给到 管道读端
 execlp("wc", "wc", "-l", NULL);
 }
 return 0;
}
```

### 4.1.2 命名管道

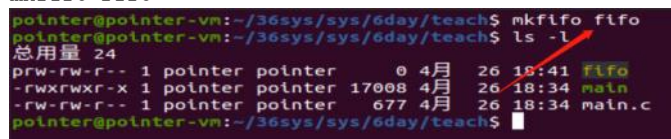
匿名管道只能用于具有共同祖先的进程(具有亲缘关系的进程)之间的通信,通常,一个管道由一个进程创建,然后该进程调用fork,此后父子进程之间就可应用该管道。如果要实现两个毫不相关进程之间的通信,可以使用命名管道来做到。命名管道就是一种特殊类型的文件,两个进程通过命名管道的文件名打开同一个管道文件,此时这两个进程也就看到了同一份资源,进而就可以进行通信了。命名管道和匿名管道一样,都是内存文件,只不过命名管道在磁盘有一个简单的映像,但这个映像的大小永远为0,因为命名管道和匿名管道都不会将通信数据刷新到磁盘当中。

命名管道,命名指的就是文件名。

#### 1、创建命名管道

##### (1) 系统命令创建

mkfifo fifo



```
pointer@pointer-vn:~/36sys/sys/6day/teach$ mkfifo fifo
pointer@pointer-vn:~/36sys/sys/6day/teach$ ls -l
总用量 24
prw-rw-r-- 1 pointer pointer 0 4月 26 18:41 fifo
-rwxrwxr-x 1 pointer pointer 17008 4月 26 18:34 main
-rw-rw-r-- 1 pointer pointer 677 4月 26 18:34 main.c
pointer@pointer-vn:~/36sys/sys/6day/teach$
```

##### (2) 函数创建——mkfifo 函数

**函数描述:**程序中创建命名管道

**头文件:**

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

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

**函数原型:**

```
int mkfifo(const char *pathname, mode_t mode),
```

**函数参数:**

pathname:表示要创建的命名管道文件

mode:表示创建命名管道文件的默认权限

**函数返回值:**

成功, 返回 0

失败, 返回 -1

#### 命名管道在父子进程间通信

```
int main(int argc, char* argv[])
{
 // 创建命名管道
 mkfifo("./fifo",0644);
 // 打开命名管道
 int fd = open("./fifo",O_RDWR);
 // 创建子进程
 int pid = fork();
 if(pid == 0) // 子进程, 向命名管道写入内容
 {
 write(fd,"hello fifo",11);
 }
 else
 {
 // 父进程读取管道中的内容
 char buf[64];
 int ret = read(fd,buf,sizeof(buf));
 printf("ret = %d,buf = %s\n",ret,buf);
 }
 return 0;
}
```

#### 命名管道在没有血缘关系之间的进程间通信

```
write.c
int main(int argc, char* argv[])
{
 // 以写入方式, 打开命名管道
 int fd = open("./fifo",O_WRONLY);
 // 打印写入时的文件描述符
 printf("wr_fd = %d\n",fd);
 // 向命名管道写入内容
```

```

 int ret = write(fd, "hello fifo1", 12);
 // 打印写入的字符
 printf("wr_ret = %d\n", ret);
 sleep(2);
 printf("l1l1\n");
 return 0;
}

```

#### read.c

```

int main(int argc, char* argv[])
{
 // 以读取方式, 打开命名管道
 int fd = open("./fifo", O_RDONLY);
 // 打印读取的文件描述符
 printf("read_fd = %d\n", fd);
 // 读取管道内容
 char buf[64];
 int ret = read(fd, buf, sizeof(buf));
 // 打印读到的信息
 printf("rd_ret = %d, buf = %s\n", ret, buf);
 return 0;
}

```

在没有写入内容时, 读端会一直阻塞, 等待写端写入。  
 在没有读取时, 写入也会一直阻塞, 直到有读端打开

这里读写的阻塞是根据打开管道文件时的权限来判断是读还是写操作的。因此, 在打开命名管道时, **一定要确定好打开时的读写方式。**

#### 命名管道的打开规则:

- 读进程打开 FIFO, 并且没有写进程打开时:
  - 没有O\_NONBLOCK: 阻塞直到有写进程打开该 FIFO
  - 有O\_NONBLOCK: 立刻返回成功
- 写进程打开 FIFO, 并且没有读进程打开时:
  - 没有O\_NONBLOCK: 阻塞直到有读进程打开该FIFO
  - 有O\_NONBLOCK: 立刻返回失败, 错误码为ENXIO

#### 练习: 两个进程使用 fifo 互发消息聊天

1. 在终端输入和打印消息
2. 能一直发消息和接收消息
3. 互发(两个管道, 两个进程)

#### wfifo.c

```

int main(int argc, char* argv[])
{
 char buf[1024];
 int ret;
 int pid = fork();
 printf("pid = %d", pid);
 if(pid == 0) // 写
 {
 int fd = open("./wr_fifo", O_WRONLY);
 while(1)
 {
 int read_count = read(0, buf, sizeof(buf));
 write(fd, buf, read_count);
 }
 }
 else // 读
 {
 int fd1 = open("./rd_fifo", O_RDONLY);
 while(1)
 {
 int ret = read(fd1, buf, sizeof(buf));
 write(1, buf, ret);
 }
 }
}

```

```

 return 0;
}

rfifo.c
int main(int argc, char* argv[])
{
 char buf[1024];
 int ret;
 int pid = fork();
 if(pid == 0) // 写
 {
 int fd = open("./rd_fifo", O_WRONLY);
 while(1)
 {
 int read_count = read(0, buf, sizeof(buf));
 write(fd, buf, read_count);
 }
 }
 else // 读
 {
 int fd1 = open("./wr_fifo", O_RDONLY);
 while(1)
 {
 int ret = read(fd1, buf, sizeof(buf));
 write(1, buf, ret);
 }
 }
 return 0;
}

```

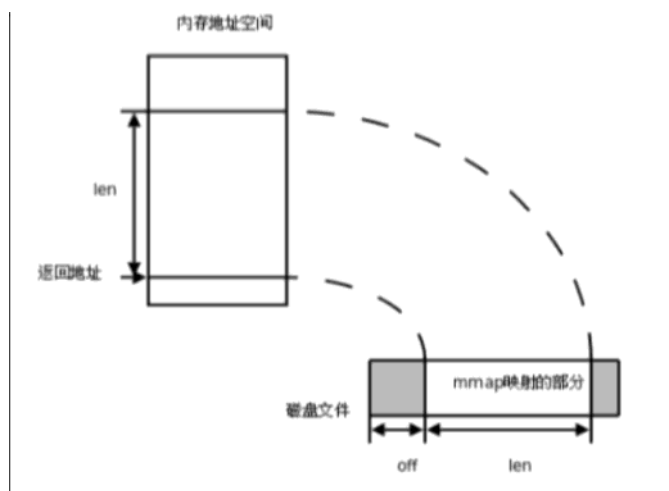
## 4.2 内存映射

管道的本质还是文件，不同进程通过访问管道文件来进行通信，但管道是在内存上的文件，并不在磁盘进行更新。

我们能否自定义一个内存空间，来让不同的进程来访问这块空间，来实现共享内存，实现进程通信呢？

内存映射即可解决上面的问题。

内存映射，实际上是拿一个磁盘文件，将其映射到内存上（物理内存），然后将这块内存给到不同的进程，以实现内存共享。



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。

**映射分为两种：**

文件映射：将文件的一部分映射到调用进程的虚拟内存中。对文件映射部分的访问转化为对相应内存区域的字节操作。映射页面会按需自动从文件中加载。

匿名映射：一个匿名映射没有对应的文件。其映射页面的内容会被初始化为 0。

**一个进程所映射的内存可以与其他进程的映射共享，共享的两种方式：**

两个进程对同一文件的同一区域映射。

fork() 创建的子进程继承其父进程的映射。

## mmap() 函数

### 函数描述:

在调用进程的虚拟地址空间中创建一个新内存映射。

### 头文件:

<sys/mman.h>

### 函数原型:

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

### 函数参数:

addr: 指向欲映射的内存起始地址，通常设为NULL，代表系统自动选定地址

length: 映射的长度。

prot: 映射区域的保护方式:

PROT\_READ: 映射区域可读取

PROT\_WRITE: 映射区域可修改

flags: 影响映射区域的特性。必须指定MAP\_SHARED(共享) 或 MAP\_PRIVATE(私有)

MAP\_SHARED: 创建共享映射。对映射的写入会写入文件里，其他共享映射的进程可见

MAP\_PRIVATE: 创建私有映射。对映射的写入不会写入文件里，其他映射进程不可见

MAP\_ANONYMOUS: 创建匿名映射。此时会忽略参数fd(设为-1)，不涉及文件，没有血缘关系的进程不能共享

fd: 要映射的文件描述符，匿名映射设为 -1

offset: 文件映射的偏移量，通常设置为0，代表从文件最前方开始对应，offset必须是分页大小(4k)的整数倍。

### 函数返回值:

若映射成功则返回映射区的内存起始地址，

失败返回MAP\_FAILED(-1)，错误原因存于 errno 中。

## munmap() 函数

### 函数描述:

解除映射区域

### 头文件:

<sys/mman.h>

### 函数原型:

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

### 函数参数:

addr: 指向要解除映射的内存起始地址

length: 解除映射的长度

## 父子进程间通信——匿名映射

// 匿名映射

```
int main(int argc, char* argv[])
{
 // void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
 // 创建匿名映射
 char* str = (char*)mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
 int pid = fork();
 if(pid == 0)
 {
 strcpy(str, "hello mmap");
 printf("child str = %s\n", str);
 }
 else
 {
 // 内存映射没有阻塞，读端得加 sleep()，确保读者之间就写入了。
 sleep(2);
 printf("father str = %s\n", str);
 }
 return 0;
}
```

## 不同进程间通信——命名映射

#### w\_mmap.c

```
// 命名映射-写端
int main(int argc, char* argv[])
{
 int fd = open("./mmap.txt", O_RDWR | O_CREAT, 0644);
 // 这里会报错, 因为要将磁盘上的文件映射到1024大小的内存上, 但是文件刚刚创建, 大小是0, 无法映射
 // 因此要扩容
 // 截断函数, 截断到某一位置, 0就是清除内容。
 ftruncate(fd, 1024);
 char* str = (char*)mmap(NULL, 1024, PROT_WRITE, MAP_SHARED, fd, 0);
 // 映射失败报错
 if(str == MAP_FAILED)
 {
 perror("mmap error");
 exit(1);
 }
 int i = 0;
 while(1)
 {
 // 格式化打印, 打印内容到指定位置, 而不是终端
 sprintf(str, "---hello %d---\n", i++);
 sleep(1);
 // 每秒打印 ---hello i--- 到内存映射中
 }

 return 0;
}
```

#### r\_mmap.c

```
// 命名映射-读端
int main(int argc, char* argv[])
{
 int fd = open("./mmap.txt", O_RDWR);
 char* str = (char*)mmap(NULL, 1024, PROT_READ, MAP_SHARED, fd, 0);
 while(1)
 {
 // 每秒从内存映射中读取一次
 printf("read str = %s", str);
 sleep(1);
 }
 return 0;
}
```

## 4.3 消息队列

消息队列是保存在内核中的消息链表, 消息队列是面向消息进行通信的, 一次读取一条完整的消息, 每条消息中还包含一个整数表示优先级, 可以根据优先级读取消息。进程A可以往队列中写入消息, 进程 B读取消息。并且, 进程A写入消息后就可以终止, 进程B在需要的时候再去读取。

每条消息通常具有以下属性:

- (1) 一个表示优先级的整数
- (2) 消息数据部分的长度
- (3) 消息数据本身

### 消息队列函数

头文件:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqqueue.h>
```

打开和关闭消息队列:

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

```

 int mq_close(mqd_t mqdes);
获取和设置消息队列属性:
 int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
 int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);
在队列中写入和读取一条消息:
 int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
 ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
删除消息队列名:
 int mq_unlink(const char *name);

```

函数参数和返回值:

name:消息队列名  
 oflag:打开方式, 类似 open 函数。  
 必选项:O\_RDONLY, O\_WRONLY, O\_RDWR  
 可选项:O\_NONBLOCK, O\_CREAT, O\_EXCL  
 mode:访问权限, oflag 中含有 O\_CREAT 且消息队列不存在时提供该参数  
 attr:队列属性, open 时传 NULL 表示默认属性  
 mqdes:表示消息队列描述符  
 msg\_ptr:指向缓冲区的指针  
 msg\_len:缓冲区大小  
 msg\_prio:消息优先级

返回值:

成功返回 0, open 返回消息队列描述符, mq\_receive 返回写入成功字节数  
 失败返回 -1

### 注意

在编译时报 undefined reference to mq\_open、undefined reference to mq\_close 时, 除了要包含头文件 #include<mqqueue>#include <fcntl>外, 还需要加上编译选项 -lrt。

### 消息队列的关闭与删除

使用 mq\_close 函数**关闭消息队列**, 关闭后消息队列并不从系统中删除。一个**进程结束**会自动调用关闭打开着的消息队列。

使用 mq\_unlink 函数**删除一个消息队列名**, 使当前进程无法使用该消息队列。并将队列标记为在所有进程关闭该队列后删除该队列。

Posix 消息队列具备随内核的持续性。所以即使当前没有进程打开着某个消息队列, 该队列及其上的消息也将一直存在, 直到调用 mq\_unlink 并最后一个进程关闭该消息队列时, 将会被删除。操作系统会维护一个消息队列的引用计数, 记录有多少进程正在使用该消息队列。只有当引用计数减为零时, 消息队列才会被真正删除。

### 查看消息队列的状态

```

int main(int argc, char* argv[])
{
 // 打开消息队列
 mqd_t mqd = mq_open("/mymsg", O_RDONLY);
 if(mqd == -1)
 {
 perror("mq_open error\n");
 exit(1);
 }

 // 接收消息队列状态
 struct mq_attr attr;
 int ret = mq_getattr(mqd, &attr);
 if(ret == -1)
 {
 perror("mq_getattr error\n");
 exit(1);
 }

 printf("mq_curmsgs = %ld\n", attr.mq_curmsgs); // 现有消息数量
 printf("mq_msgsize = %ld\n", attr.mq_msgsize); // 消息队列大小
 printf("mq_maxmsg = %ld\n", attr.mq_maxmsg); // 最大消息数量
}

```



```
 return 0;
}
```