实验三：Linux进程管理

目录

[一、知识铺垫 1](#_Toc28390269)

[1.Makefile 1](#_Toc28390270)

[2.相关函数 1](#_Toc28390271)

[3. POSIX 信号量 2](#_Toc28390272)

[4. 全双工，半工 3](#_Toc28390273)

[二、管道 3](#_Toc28390274)

[1. 知识点 3](#_Toc28390275)

[2.具体实现 6](#_Toc28390276)

[三、消息队列 7](#_Toc28390277)

[四、共享内存 7](#_Toc28390278)

## 一、知识铺垫

### 1.Makefile

[Makefile参考资料1](https://www.cnblogs.com/tp-16b/p/8955462.html)，[Makefile参考资料2](https://www.cnblogs.com/tp-16b/p/8955462.html)， [Makefile中的all和PHONY的作用](https://www.cnblogs.com/tp-16b/p/8955462.html)

make是一个 解释Makefile中指令的命令工具

由于Makefile只能有一个目标，所以可以构造一个没有规则的终极目标**all**。

### 2.相关函数

#### 2.1 fork()

[fork入门](https://blog.csdn.net/jason314/article/details/5640969)，[fork出的子进程和父进程](https://blog.csdn.net/u013851082/article/details/76902046)

fork通常作为一个函数被调用。这个函数会有两次返回，**将子进程的PID返回给父进程，0返回给子进程**(只是返回值是0，但是子进程的pid不是0)。实际上，子进程总可以查询自己的PPID来知道自己的父进程是谁，这样，一对父进程和子进程就可以随时查询对方。

通常在调用fork函数之后，程序会设计一个if选择结构。当PID等于0时，说明该进程为子进程，那么让它执行某些指令,比如说使用exec库函数(library function)读取另一个程序文件，并在当前的进程空间执行 (这实际上是我们使用fork的一大目的: 为某一程序创建进程)；而当PID为一个正整数时，说明为父进程，则执行另外一些指令。由此，就可以在子进程建立之后，让它执行与父进程不同的功能。

*\*注意事项：*

* 创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。
* fork只拷贝下一个要执行的代码到新的进程。

pid = fork() # fork()返回的就是子进程的PID

#### 2.2 execl()

#include <unistd.h>  
int execl(const char \*path, const char \*arg, ...);

execl()其中后缀"l"代表list也就是参数列表的意思，第一参数path字符指针所指向要执行的文件路径，参数实际上就是用户在执行这个可执行文件时所需的全部命令选项字符串（包括该可执行程序名本身),接下来的参数代表执行该文件时传递的参数列表：argv[0],argv[1]... **最后一个参数须用空指针NULL作结束**。

函数返回值： **成功则不返回值**， 失败返回-1， 失败原因存于errno中，可通过perror()打印

程序调用exec的时候，进程清空自身内存空间的text, global data, heap和stack，并根据新的程序文件重建text, global data, heap和stack (此时heap和stack大小都为0)，并开始运行。

当进程调用一种exec函数时，该进程完全由新程序代换，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。

#### 2.3 wait()

**wait就是经过包装的waitpid**

static inline pid\_t wait(int \* wait\_stat)  
{  
 return waitpid(-1,wait\_stat,0);  
}

父进程一旦调用了wait就立即阻塞自己，由wait自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止。

wait()要与fork()配套出现,如果在使用fork()之前调用wait(),wait()的返回值则为-1,正常情况下wait()的返回值为**子进程的PID**.

#### 2.4 write() 和 read()

[深入理解linux下write()和read()函数](https://blog.csdn.net/hhhlizhao/article/details/71552588)

函数定义：ssize*t write (int fd, const void \* buf, size*t count);

函数说明：write()会把参数buf所指的内存写入count个字节到参数fd所指的文件内。

返回值：如果顺利write()会返回实际写入的字节数（len）。当有错误发生时则返回-1，错误代码存入errno中。

### 3. POSIX 信号量

[POSIX信号量机制](https://www.cnblogs.com/nufangrensheng/p/3564306.html),[信号量机制](https://www.cnblogs.com/nufangrensheng/p/3564306.html)

#### 3.1 sem\_wait()

如果对一个值为0的信号量调用sem*wait()，这个函数就会原地等待直到有其它线程增加了这个值使它不再是0为止。如果有两个线程都在sem*wait()中等待同一个信号量变成非零值，那么当它被第三个[线程](https://baike.baidu.com/item/线程/103101)增加 一个“1”时，等待线程中只有一个能够对信号量做减法并继续执行，另一个还将处于等待状态。sem*trywait(sem*t \*sem)是函数sem\_wait的非阻塞版，它直接将信号量sem减1，同时返回错误代码。

### 4. 全双工，半工

全双工（Full Duplex）是指在发送数据的同时也能够接收数据，两者同步进行，这好像我们平时打电话一样，说话的同时也能够听到对方的声音。目前的网卡一般都支持全双工。

半双工（Half Duplex），所谓半双工是指一段时间内，只有一种动作发生，例如一条窄路，而只有一辆车可以通行，当当前有两辆车相对时，在这种情况下，只有一辆车先开，等到另一辆车的头部再开，这个例子生动地说明了原始半双工。

## 二、管道

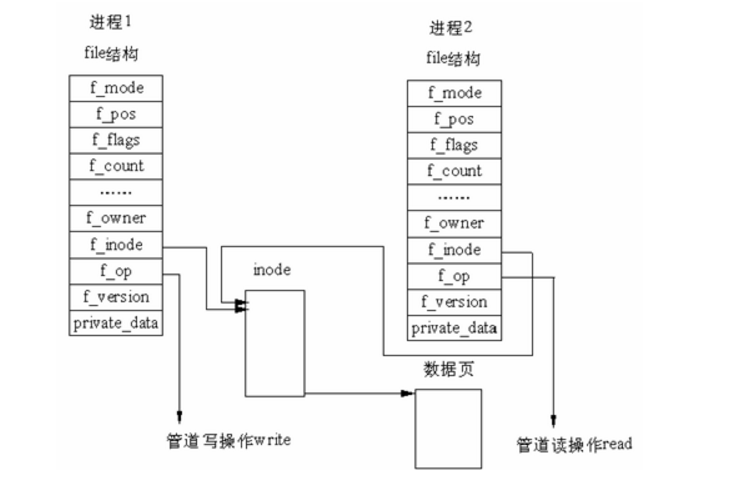
[管道](https://blog.csdn.net/qq_42914528/article/details/82023408),[实验三](https://imwjc.xyz/2019/04/os-lab-3-2/),[Linux环境进程间通信（一）](https://www.ibm.com/developerworks/cn/linux/l-ipc/part1/)

### 1. 知识点

管道是半双工的，即同一时间同一进程只能读取或者写入。管道又分为有名管道和无名管道两种，无名管道存在于高速缓存 cache 中，用于有亲缘关系的父子进程或兄弟进程之间的通信，有名管道存在于磁盘中，是看得见摸得着的真实文件，只要知道路径名就可以调用，所以它可以用于任意进程之间的通信。

在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的file结构和VFS的索引节点inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。

[实现细节](https://www.cnblogs.com/biyeymyhjob/archive/2012/11/03/2751593.html)



有两个 file 数据结构，但它们定义文件操作例程地址是不同的，其中一个是向管道中写入数据的例程地址，而另一个是从管道中读出数据的例程地址。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。

#### 1. 1管道读写行为

使用管道需要注意以下4种特殊情况（假设都是阻塞I/O操作，没有设置O\_NONBLOCK标志）：

1. 如果所有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），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，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关闭（管道写端引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可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（管道读端引用计数为0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。（**只有管道中读端存在时，向管道中写入数据才有意义，如果要写入的话至少保证读端有引用**）
4. 如果有指向管道读端的文件描述符没关闭（管道读端引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了才写入数据并返回。

总结：

① 读管道：

1. 管道中有数据，read返回实际读到的字节数。
2. 管道中无数据：
   * 管道写端被全部关闭，read返回0 (好像读到文件结尾)
   * 写端没有全部被关闭，read阻塞等待(不久的将来可能有数据递达，此时会让出cpu)

② 写管道：

1. 管道读端全部被关闭， 进程异常终止(也可使用捕捉SIGPIPE信号，使进程不终止)
2. 管道读端没有全部关闭：
   * 管道已满，write阻塞。
   * 管道未满，write将数据写入，并返回实际写入的字节数。

[管道读写行为](https://blog.csdn.net/qq_42914528/article/details/82023408)

int pipe(int pipefd[2]); //成功：0；失败：-1，设置errno

#### 1. 2 fcntl()

规定：fd[0] → r； fd[1] → w

当管道满的时候（不断往管道写入），write函数也有两种可能：

如果文件状态标志(O\_NONBLOCK disable)阻塞（未设置），**write调用阻塞**；

文件状态标志（O\_NONBLOCK）非阻塞, write返回-1 errno=EAGAIN.

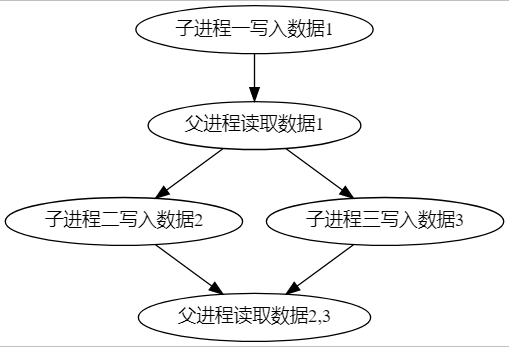
**阻塞模式的话，如果read无数据可读就阻塞；如果非阻塞模式的话，read无数据可读，则立即返回-1.**

### 2.具体实现

使用了三个信号量，分别是 write\_mutex、read\_mutex1 和 read\_mutex2，简单分析一下子进程和父进程之间的关系可以明白：

* 子进程一先将 64K 的数据写入管道，父进程才能第一时间将数据全部读取出来（来自一进程的数据）
* 父进程将子进程一的数据读取之后，子进程二、三才能写入数据
* 子进程二、三将数据写入后，父进程随后才能读取第二批数据（来自二、三进程的数据）

关系大致如下图所示：



子进程写入数据1 和父进程读取数据1 利用 wait(0) 限制了先后关系，父进程必须接收到子进程结束之后返回的 0，才能继续运行，否则阻塞。

write\_mutex 限制了父进程先读取数据，然后子进程二、三写入数据，read\_mutex1 和 read\_mutex2 分别限制了子进程二、三写入数据 2,3 和父进程读取数据 2,3 先后关系，只有子进程二、三均完成后，父进程才允许读取管道。

子进程一使用了非阻塞性写，子进程二、三均为阻塞性写，父进程为非阻塞性读。

非阻塞写和非阻塞读的目的在于，阻塞写时，管道满了之后进程被阻塞，无法设置终止条件从而结束写，读也是一样，管道空了之后进程被阻塞，无法设置终止条件从而结束读。

## 三、消息队列

[从 0 开始学习 Linux 系列之23.消息队列 Msg Queue](https://www.jianshu.com/p/7598e5ed5200)，消息队列[Linux环境进程间通信（三）](https://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html)

#### 3.2 msgsnd (将消息写入到消息队列)

int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

#### 3.3 msgrcv (从消息队列读取消息)

ssize*t msgrcv(int msqid, void \*msgp, size*t msgsz, long msgtyp, int msgflg);

功能：从标识符为msgid的消息队列里接收一个指定类型的消息 并 存储于msgp中 读取后 把消息从消息队列中删除

## 四、共享内存

[从 0 开始学习 Linux 系列之22.共享内存 Shared Memory](%20https://www.jianshu.com/p/494c2d32e3bb%20)

**内核开辟一片内存区域，然后多个用户进程可以将这片区域映射到它们自己的地址空间中进行读写**。

共享内存（SHM）的操作主要分为下面 4 个步骤：

1. 创建或获取 SHM
2. 映射 SHM 到进程地址空间
3. 操作映射后的区域，即读写
4. 关闭进程地址空间的映射区域

/\*  
 \* key：SHM 标识  
 \* size：SHM 大小  
 \* shmflg：创建或得到的属性，例如 IPC\_CREAT  
 \* return：成功返回 shmid，失败返回 -1，并设置 erron  
 创建  
 \*/  
int shmget(key\_t key, size\_t size, int shmflg);  
  
/\*  
 \* shmid：SHM ID  
 \* shmaddr：SHM 内存地址  
 \* shmflg：SHM 权限  
 \* return：成功返回 SHM 的地址，失败返回 (void \*) -1，并设置 erron  
 映射  
 \*/  
void \*shmat(int shmid, const void \*shmaddr, int shmflg);  
  
/\*  
 \* shmaddr：已经映射的 SHM 地址  
 \* return：成功返回 0，失败返回 -1，并设置 erron  
 关闭  
 \*/  
int shmdt(const void \*shmaddr);