

学习目标

1. 熟练使用**pipe**进行父子进程间通信 |
2. 熟练使用**pipe**进行兄弟进程间通信
3. 熟练使用**fifo**进行无血缘关系的进程间通信
4. 熟练掌握**mmap**函数的使用
5. 掌握**mmap**创建匿名映射区的方法
6. 使用**mmap**进行有血缘关系的进程间通信
7. 使用**mmap**进行无血缘关系的进程间通信

进程回收

1. 创建子进程：fork

○ 特点：

- 两个返回值：
 - 父进程：子进程的pid > 0
 - 子进程：返回值为0
- 父子进程的执行属性：随机
- 子进程从代码的什么位置执行：
 - 父进程执行到的位置开始执行
- 父子进程数据永远一样吗？
 - fork完成之后，一样
 - 读时共享，写时复制 - 有血缘关系

2. 常用函数：

- a. getpid ()
- b. getppid ()
- c. execl ("/home/kvin/hello") ;
- d. execlp ("ps") ;

3. 进程回收

○ wait - 阻塞函数

- pid_t wait(int* status);
 - 调用一次回收一个子进程资源
- 返回值：
 - >0: 回收的子进程的pid
 - -1: 没有子进程可以回收了
- status -- 传出参数
 - 获取退出时候的返回值(正常退出)
 - ◆ return 0; -- main
 - ◆ exit(0);
 - ◆ w if exited(status) > 0
 - ◇ w exit status(status)
 - 被信号杀死
 - ◆ w if signaled(status) > 0

w term sig(status)

○ waitpid --

▪ pid_t waitpid(pid_t pid, int *status, int options);

□ pid:

◆ pid > 0: 某个子进程的pid

◆ pid == -1: 回收所有的子进程

◇ 循环回收,

◇ while ((wpid = waitpid(-1, &status, WNOHANG) != -1) ?

◆ pid == 0:

◇ 回收当前进程组所有的子进程

◆ pid < 0: 子进程的pid取反(加减号)

□ options:

◆ 0 - waitpid阻塞

◆ WNOHANG - 非阻塞

□ 返回值:

◆ -1: 回收失败, 没有子进程

◆ >0: 被回收的子进程的pid

◆ 如果为非阻塞:

◇ =0: 子进程处于运行状态

1 - 进程间通信相关概念

1. 什么是IPC

a. 进程间通信

i. InterProcess Communication

2. 进程间通信常用的4种方式

a. 管道 - 简单

b. 信号 - 系统开销小

c. 共享映射区 - (有无血缘关系的进程间通信都可以)

d. 本地套接字 - 稳定

2- 管道(匿名)

1. 管道的概念

- 本质:

 - 内核缓冲区

 - 伪文件 - 不占用磁盘空间

- 特点:

 - 两部分:

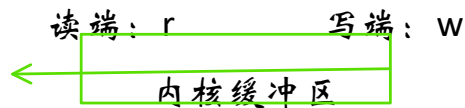
 - 读端, 写端, 对应两个文件描述符

 - 数据写端流入, 读端流出

 - 操作管道的进程被销毁之后, 管道自动被释放了

 - 管道默认是阻塞的。

 - 读写



2. 管道的原理

- 内部实现方式: 队列

 - 环形队列

 - 特点: 先进先出

- 缓冲区大小:

 - 默认4k

 - 大小会根据实际情况做适当调整

3. 管道的局限性

- 队列:

 - 数据只能读取一次, 不能重复读取

- 半双工:

 - 单工: 遥控器

 - 半双工: 对讲机

 - 数据传输的方向是单向的

 - 双工: 电话

- 匿名管道:

 - 适用于有血缘关系的进程

4. 创建匿名管道

- `int pipe(int fd[2]);`

 - fd- 传出参数

 - fd[0] - 读端

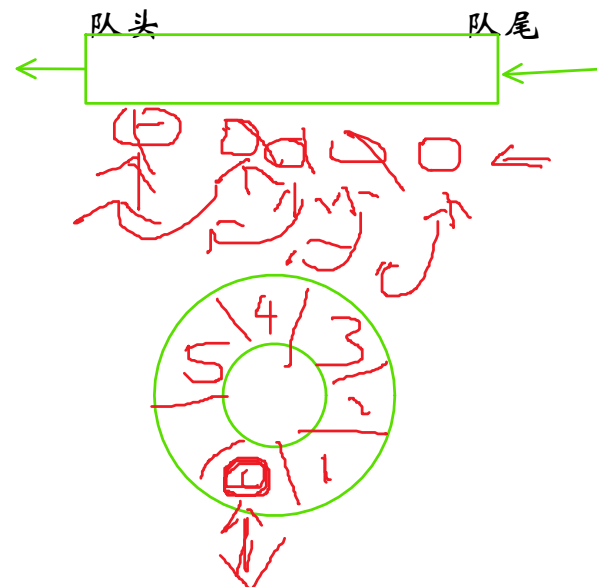
 - fd[1] - 写端

5. 父子进程使用管道通信

- 思考:

 - 单个进程能否使用管道完成读写操作?

 - 可以



- 父子进程间通信是否需要sleep函数?

- 父写 -- 写的慢

- 子读 -- 读的快

- 注意事项:

- 父进程读

- 关闭写端

- 子进程写

- 关闭读端

- 练习

- 父子进程间通信, 实现ps aux | grep bash

- 数据重定向: dup2

- execlp

- 兄弟进程间通信, 实现ps aux | grep bash

- 父亲 - 资源回收

1. 管道的读写行为

- 读操作

- 有数据

- read (fd) - 正常读, 返回读出的字节数

- 无数据

- 写端全部关闭

- ◆ read解除阻塞, 返回0

- ◆ 相当于读文件读到了尾部

- 没有全部关闭

- ◆ read阻塞

- 写操作

- 读端全部关闭

- 管道破裂, 进程被终止

- ◆ 内核给当前进程发信号SIGPIPE

- 读端没全部关闭

- 缓冲区写满了

- ◆ write阻塞

- 缓冲区没有满

- ◆ write继续写

- 如何设置非阻塞?

- 默认读写两端都阻塞

- 设置读端为非阻塞pipe (fd)

- fcntl - 变参函数

- ◆ 复制文件文件描述符 - dup

- ◆ 修改文件属性 - open的时候对应flag

层 址

· 复制文件文件描述符 dup

- ◆ 修改文件属性 - open的时候对应flag属性

□ 设置方法:

获取原来的flags

```
int flags = fcntl(fd[0], F_GETFL);
```

// 设置新的flags

```
flag |= O_NONBLOCK;
```

```
// flags = flags | O_NONBLOCK;
```

```
fcntl(fd[0], F_SETFL, flags);
```

2. 查看管道缓冲区大小

○ 命令

- ulimit -a

○ 函数

- fpathconf

3 - fifo

1. 特点

- 有名管道
- 在磁盘上有这样一个文件 `ls -l -> p`
- 伪文件，在磁盘大小永远为0
- 在内核中有一个对应的缓冲区
- 半双工的通信方式

2. 使用场景

- 没有血缘关系的进程间通信

3. 创建方式

- 命令: `mkfifo` 管道名
- 函数: `mkfifo`

4. fifo文件可以使用IO函数进行操作

- `open/close`
- `read/write`
- 不能执行 `lseek` 操作

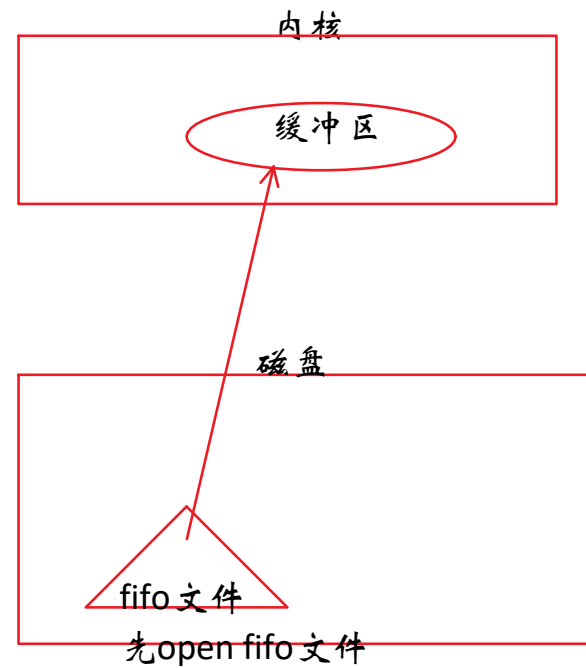
5. 进程间通信

a. fifo文件 --- `myfifo`

- 两个不相干的进程 `A(a.c)` `B(b.c)`
- `a.c` ---> `read`
 - `int fd = open("myfifo", O_RDONLY);`
 - `read(fd, buf, sizeof(buf));`
 - `close(fd);`

b. `b.c` ----- `write`

```
int fd1 = open("myfifo", O_WRONLY);
write(fd1, "hello, world", 11);
close(fd1);
```



4 - 内存映射区

1. mmap - 创建内存映射

- 作用: 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
- 函数原型:

```
void *mmap(  
    void *addr,      // 映射区首地址, 传NULL  
    size_t length,   // 映射区的大小  
        ◆ 100byte - 4k  
        ◆ 不能为0  
        ◆ 一般文件多大, length就指定多大  
    int prot,        // 映射区权限  
        PROT_READ -- 映射区必须要有读权限  
        PROT_WRITE  
        PROT_READ | PROT_WRITE  
    int flags,       // 标志位参数  
        ◆ MAP_SHARED  
            ◇ 修改了内存数据会同步到磁盘  
        ◆ MAP_PRIVATE  
            ◇ 修改了内存数据不会同步到磁盘  
    int fd,          // 文件描述符  
        ◆ 干嘛的文件描述符?  
            ◇ 要映射的文件对应fd  
        ◆ 怎么得到?  
            ◇ open ()  
    off_t offset      // 映射文件的偏移量  
        ◆ 映射的时候文件指针的偏移量  
            ◇ 必须是4k的整数倍  
            ◇ 0  
);
```

- 返回值:

- 映射区的首地址 - 调用成功
- 调用失败: MAP_FAILED

2. munmap - 释放内存映射区

- 函数原型: `int munmap(void *addr, size_t length);`
 - `addr` -- `mmap`的返回值, 映射区的首地址
 - `length` -- `mmap`的第二个参数, 映射区的长度

3. 思考问题:

- 如果对`mmap`的返回值(`ptr`)做++操作(`ptr++`), `munmap`是否能够成功?
- 如果`open`时`O_RDONLY`, `mmap`时`prot`参数指定`PROT_READ | PROT_WRITE`会怎样?
- 如果文件偏移量为1000会怎样?
- 如果不检测`mmap`的返回值会怎样?
- `mmap`什么情况下会调用失败?
- 可以`open`的时候`O_CREAT`一个新文件来创建映射区吗?
- `mmap`后关闭文件描述符, 对`mmap`映射有没有影响?
- 对`ptr`越界操作会怎样?

4. 进程间通信

- a. 有血缘关系的
 - i. 父子进程共享内存映射区
- b. 没有血缘关系的进程间通信
 - i. 如何通信?

5. mmap 实现内存映射:

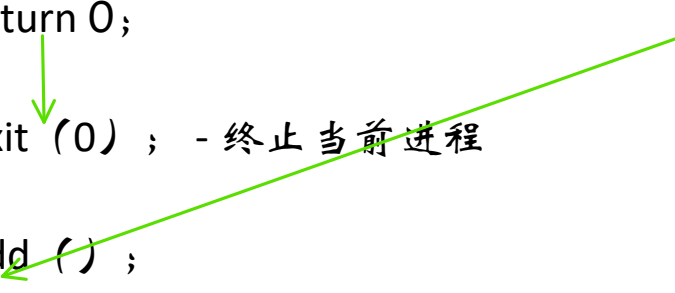
- a. 必须有一个文件
- b. 文件数据什么时候有用:
 - i. 单纯文件映射
 - ii. 进程间通信:
 - 1) 文件数据是没有用的

```
main ( )
{
return 0;
exit (0) ; - 终止当前进程
add ( ) ;

XXXXXX

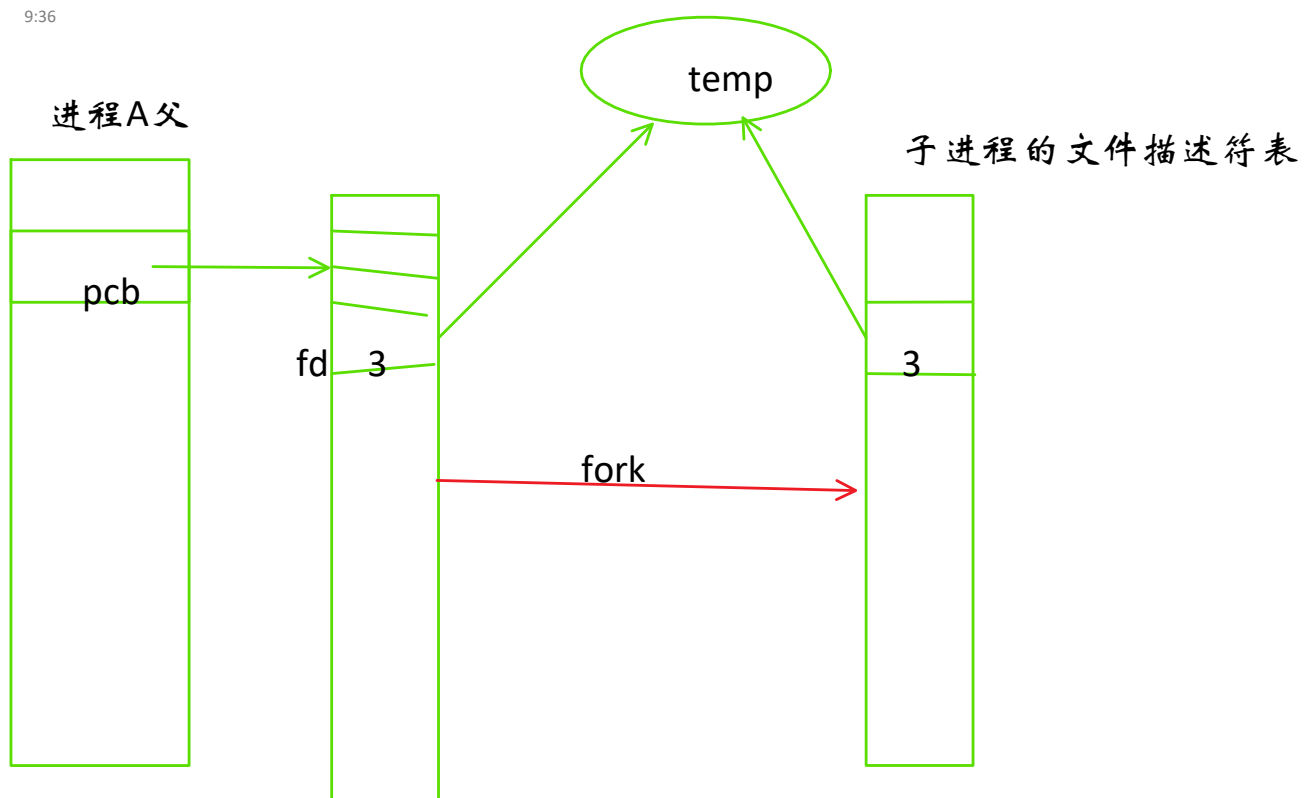
}

int add ( )
{
return 1; -- 返回到调用者的位置
XXXX
exit (0) ;
}
```



父子进程共享文件描述符

2017年3月21日 9:36



mmap - 创建内存映射区

malloc - free
new - delete
mmap - munmap

