第四次实例分析

第五部分 pipe：管道的实现

小组成员：付星辰　徐龙午

中国科学院大学
University of Chinese Academy of Sciences

问题1

pipe 类型的文件操作实现的是什么样的功能？
pipealloc 函数的两个参数分别代表什么？

什么是pipe？

管道是一个小的内核缓冲区，作为一对文件描述符暴露给进程，一个用于读，一个用于写。将数据写入管道的一端就可以从管道的另一端读取数据。管道为进程提供了一种通信方式。

pipe的结构

```
#define PIPESIZE 512

struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read    读取到的字节位置
  uint nwrite;    // number of bytes written 写入到的字节位置
  int readopen;   // read fd is still open   1: 开放 0: 关闭
  int writeopen;  // write fd is still open  1: 开放 0: 关闭
};
```

pipealloc函数

```
int pipealloc(struct file **f0, struct file **f1){
  struct pipe *p;

  p = 0;
  *f0 = *f1 = 0;
  //如果f0, f1 创建失败，则goto bad，返回-1
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  //若p分配失败，则goto bad，返回-1
  if((p = (struct pipe*)kalloc()) == 0)
    goto bad;
  //初始化pipe
  p->readopen = 1;
  p->writeopen = 1;
  p->nwrite = 0;
  p->nread = 0;
  initlock(&p->lock, "pipe");
```

```
  //改变f0, f1
  (*f0)->type = FD_PIPE;  //文件类型改为管道型
  (*f0)->readable = 1;    //f0作为读取端
  (*f0)->writable = 0;
  (*f0)->pipe = p;
  (*f1)->type = FD_PIPE;
  (*f1)->readable = 0;
  (*f1)->writable = 1;    //f1作为写入端
  (*f1)->pipe = p;
  return 0;
```

```
  //如果创建失败，则释放占用的内存、解除对文件的占有
bad:
  if(p)
    kfree((char*)p);
  if(*f0)
    fileclose(*f0);
  if(*f1)
    fileclose(*f1);
  return -1;
}
```

pipealloc函数

```
int pipealloc(struct file **f0, struct file **f1){
  struct pipe *p;

  p = 0;
  *f0 = *f1 = 0;
  //如果f0, f1 创建失败，则goto bad，返回-1
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  //若p分配失败，则goto bad，返回-1
  if((p = (struct pipe*)kalloc()) == 0)
    goto bad;
  //初始化pipe
  p->readopen = 1;
  p->writeopen = 1;
  p->nwrite = 0;
  p->nread = 0;
  initlock(&p->lock, "pipe");
```

参数：两个文件类型的参数

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

pipealloc函数

```c
int pipealloc(struct file **f0, struct file **f1){
  struct pipe *p;

  p = 0;
  *f0 = *f1 = 0;
  //如果f0, f1 创建失败，则goto bad，返回-1
  if((*f0 = filealloc()) == 0 || (*f1 = filealloc()) == 0)
    goto bad;
  //若p分配失败，则goto bad，返回-1
  if((p = (struct pipe*)kalloc()) == 0)
    goto bad;
  //初始化pipe
  p->readopen = 1;
  p->writeopen = 1;
  p->nwrite = 0;
  p->nread = 0;
  initlock(&p->lock, "pipe");
```

```c
// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);     //原子性操作
  //判断文件是否已经创建满
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    //若未满，则创建f
    if(f->ref == 0){
      f->ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

```c
#define NFILE       100  // open files per system
```

pipealloc函数

```
//改变f0，f1
(*f0)->type = FD_PIPE;  //文件类型改为管道型
(*f0)->readable = 1;    //f0作为读取端
(*f0)->writable = 0;
(*f0)->pipe = p;
(*f1)->type = FD_PIPE;
(*f1)->readable = 0;
(*f1)->writable = 1;    //f1作为写入端
(*f1)->pipe = p;
return 0;
```

创建成功

失败

```
 //如果创建失败，则释放占用的内存、解除对文件的占有
bad:
 if(p)
   kfree((char*)p);
 if(*f0)
   fileclose(*f0);
 if(*f1)
   fileclose(*f1);
 return -1;
```

问题2

piperead和pipewrite 函数的流程是什么样的？
请介绍出报错（return -1）的原因，以及核心语句

piperead函数

pipe为空
→进程中断
→return -1

```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  //如果pipe已经读完，并且正在写入，则进入睡眠状态，DOC: pipe-empty
  while(p->nread == p->nwrite && p->writeopen){
    if(myproc()->killed){//进程中断
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  //读取完毕，唤醒写进程
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;            //返回读取的字节长度
}
```

pipewrite函数

pipe已满
→进程中断或读取端关闭
→return -1

data[PIPESIZE - 1]
nread之前视为空

解释为什么用%

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    //如果pipe已经写满
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || myproc()->killed){//读取端关闭或者进程中断
        release(&p->lock);
        return -1;
      }
      //唤醒读进程，写进程进入睡眠
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  //写完唤醒读进程
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

pipeclose函数

```
void
pipeclose(struct pipe *p, int writable)
{
  acquire(&p->lock); // 获取管道锁，避免在关闭的同时进行读写操作
  //判断是否还有未被读取的数据
  if(writable){//如果存在，则关闭写通道，唤醒读进程
    p->writeopen = 0;
    wakeup(&p->nread);
  } else {    //如果不存在，则关闭读通道，唤醒写进程
    p->readopen = 0;
    wakeup(&p->nwrite);
  }
  //当pipe的读写端口都关闭，则释放资源，否则释放pipe锁
  if(p->readopen == 0 && p->writeopen == 0){
    release(&p->lock);
    kfree((char*)p);
  } else
    release(&p->lock);
}
```

sys_pipe

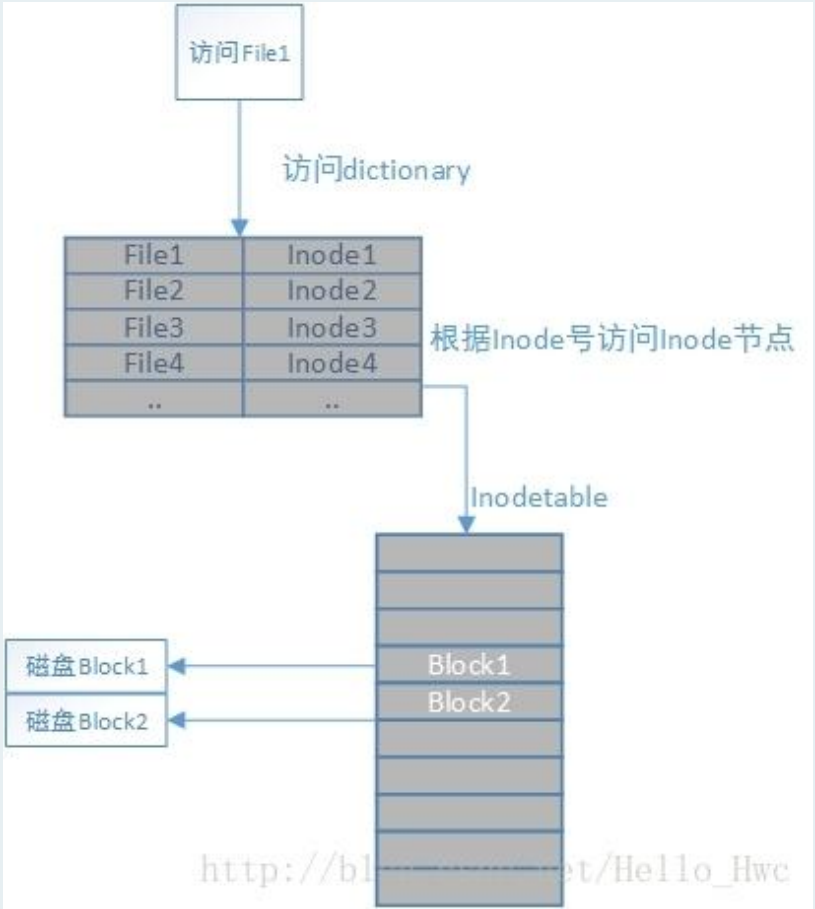
```
int
sys_pipe(void)
{ //创建管道
  int *fd;
  struct file *rf, *wf;
  int fd0, fd1;

  if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)  //判断fd是否分配成功
    return -1;
  //分配一个pipe
  if(pipealloc(&rf, &wf) < 0)
    return -1;
  fd0 = -1;
  //给fd0，fd1分别分配rf、wf的文件描述符
  if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
    //如果分配失败
    if(fd0 >= 0)  //如果fd0分配成功，则将其从打开的文件中关闭
      myproc()->ofile[fd0] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  fd[0] = fd0;
  fd[1] = fd1;
  return 0;
}
```

问题3

pipe 类型的文件读写和 inode 类型的读写有什么区别？
pipe 类型操作的实现目的是什么，有什么优点？

inode的读写

1、根据文件名，通过Directory里的对应关系，找到文件对应的inode number
2、根据inode number读取到文件的inode table
3、再根据inode table中的Pointer读取到相应的Blocks



pipe和inode

- pipe是进程间的通讯方式，具有亲缘关系的进程之间通过建立管道来进行信息交换；pipe是通过环形队列来实现的。

- incode储存的是文件的元信息，每一个incode都有一个编号，而编号在一个文件系统中是唯一的。系统通过编号来对相应的数据进行读取等操作。

为什么要用管道？

管道是一个小的内核缓冲区，作为一对文件描述符暴露给进程，一个用于读，一个用于写。将数据写入管道的一端就可以从管道的另一端读取数据。**管道为进程提供了一种通信方式。**

管道vs临时文件

shell实现了管道（sh.c:100）

在管道写入端调用 fork 和 runcmd，在读取端调用fork 和 runcmd，并等待两者的完成。

```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->left);
  }
  if(fork1() == 0){
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd->right);
  }
  close(p[0]);
  close(p[1]);
  wait();
  wait();
  break;
```

管道vs临时文件

管道：

```
echo hello world | ff
```

不使用管道：

```
echo hello world >/tmp/xyz; ff </tmp/xyz
```

管道vs临时文件

管道：
```
echo hello world | ff
```
不使用管道：
```
echo hello world >/tmp/xyz; ff </tmp/xyz
```

在这种情况下，管道比临时文件至少有四个优势：

1. 管道会自动清理自己；如果是文件重定向，shell 在完成后必须小心翼翼地删除/tmp/xyz。
2. 管道可以传递任意长的数据流，而文件重定向则需要磁盘上有足够的空闲空间来存储所有数据。
3. 管道可以分阶段的并行执行，而文件方式则需要在第二个程序开始之前完成第一个程序。
4. 如果要实现进程间的通信，管道阻塞读写比文件的非阻塞语义更有效率。

Linux

管道写函数通过将字节复制到 VFS 索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。

当写进程向管道中写入时，它利用标准的库函数write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构，file 结构中指定了用来进行写操作的函数（即写入函数）地址，内核调用该函数完成写操作。

写入函数在向内存中写入数据之前，必须首先检查 VFS 索引节点中的信息，同时满足如下条件时，才能进行实际的内存复制工作：

- 内存中有足够的空间可容纳所有要写入的数据；
- 内存没有被读程序锁定。

如果同时满足上述条件，写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存。否则，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。

写入进程实际处于可中断的等待状态，当内存中有足够的空间可以容纳写入数据，或内存被解锁时，读取进程会唤醒写入进程，这时，写入进程将接收到信号。当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒。

参考

[1]https://gitee.com/menchou/xv6study/blob/master/ref-books/XV6-Chinese-2020.pd
[2]https://www.cnblogs.com/fortnight/p/4087934.html
[3]//github.com/mit-pdos/xv6-public.git
[4]https://blog.csdn.net/liuwg1226/article/details/106307951
[5]https://zhuanlan.zhihu.com/p/58489873
[6]https://blog.csdn.net/weixin_39592137/article/details/116616671
[7]https://www.cnblogs.com/xiexj/p/7214502.html
[8]https://www.cnblogs.com/lincappu/p/8536431.html

THANKS

中国科学院大学
University of Chinese Academy of Sciences