xv6©-rev10
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)

I/O subsystem

Carmi Merimovich

Tel-Aviv Academic College

January 20, 2017

I/O: The user mode side.

I/O system calls

- System calls using file descriptors:

```
read(int fd, char *buf, int len);
write(int fd, char *buf, int len);
stat(int fd, struct stat *);
dup(int fd);
close(int fd);
```

- Name services:

```
link(char *oldpath, char *newpath);
fd = open(char *path, int flags);
unlink(char *path);
mkdir(char *path);
rmdir(char *path);
```

- Strange one:

```
pipe(int pipefd[2]);
```

- A vast number of different devices is served by few system calls!

I/O: The kernel side

Layered system

1. System calls.
2. File layer.
    - pipe subsystem.
    - inode subsystem:
        - Name layer.
        - inode layer.
        - Buffer layer.
        - Driver layer.

The layers model is not perfect, e.g.,

- System calls call the file layer, name layer, and inode layer.

How?

- Different structures.
- Different methods.
- Different hardware.
- How to present uniform interface to user mode?

xv6 ‘abstract’ file type

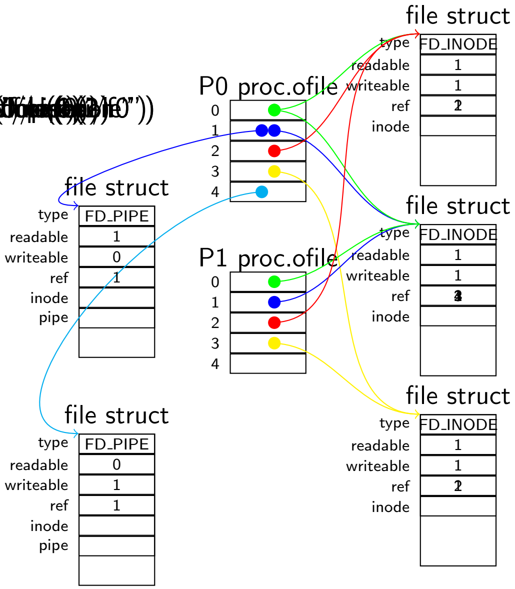
```
struct file {
 enum {FD_NONE, FD_PIPE, FD_INODE} type;
 int ref; // reference count
 char readable;
 char writable;
 struct pipe  *pipe;
 struct inode *ip;
 uint off;
};
```

File descriptor vs. struct file *

- The `struct file` is what `open` SHOULD have returned to the user.
- The user SHOULD have supplied this pointer to the other system calls.
- However! The user is NOT trustable.
- Hence the kernel HIDES the `struct file` pointer and supplies the index `fd`.

ofile and file structures relation

[illegible]



I/O subsystems

- FD_PIPE.
- FD_INODE:
  - T_FILE.
  - T_DIR.
  - T_DEV.

Each I/O subsystem is defined by a structure and a set of operations.

FD_PIPE

```
6762 struct pipe {
      struct spinlock lock;
      char data[PIPESIZE];
      uint nread;  // number of bytes read
      uint nwrite; // number of bytes written
      int readopen;  // read fd is still open
      int writeopen; // write fd is still open
     };
```

- pipalloc.
- pipeclose.
- piperead.
- pipewrite.

FD_INODE

```
4162 struct inode {
      uint dev; // Device number
      uint inum; // Inode number
      int ref; // Reference count
      struct sleeplock lock; // protects everything below
      int valid;

      short type; // copy of disk inode
      short major;
      short minor;
      short nlink;
      uint size;
      uint addrs[NDIRECT+1];
    };
```

FD_INODE methods

Listing 1: T_FILE or T_DIR

```
namei
create
ilock
readi
writei
stati
iunlock
iput
```

Listing 2: T_DEV (example)

```
consoleread
consolewrite
```

System calls referencing the file layer

- sys_dup.
- sys_read.
- sys_write.
- sys_fstat.
- sys_close.

file layer functions used by fd system calls

```
struct file *filedup(struct file *f)

fileread(struct file *f, char *buf,  int len);
filewrite(struct file *f, char *buf, int len);

filestat(struct file *f, struct stat *s);

fileclose(struct file *f);
```

fdalloc: Getting an ofile slot

- fdalloc returns the minimally numbered free slot.

```
static int fdalloc(struct file *f) {
 int fd;

 for (fd = 0; fd < NOFILE; fd++) {
  if (myproc()->ofile[fd] == 0) {
   myproc()->ofile[fd] = f;
   return fd;
  }
 }
 return -1;
}
```

argfd: Getting a file descriptor argument

```
argfd(int n, int *pfd, struct file **pf)
{
int fd;
struct file *f;

if (argint(n, &fd) < 0)
 return -1;
if (fd < 0 || fd >= NOFILE ||
     (f=myproc()->ofile[fd]) == 0)
 return -1;
if (pfd)
 *pfd = fd;
if (pf)
 *pf = f;
return 0;
}
```

sys_dup

```
sys_dup(void) {
 struct file *f;
 int fd;

 if (argfd(0, 0, &f) < 0)
  return -1;
 if ((fd=fdalloc(f)) < 0)
  return -1;
 filedup(f);
 return fd;
}
```

sys_read

```
int sys_read(void) {
 struct file *f;
 int n;
 char *p;

 if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 ||
     argptr(1, &p, n) < 0)
  return -1;
 return fileread(f, p, n);
}
```

sys_write

```
6151 int sys_write(void) {
      struct file *f;
      int n;
      char *p;

      if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 ||
            argptr(1, &p, n) < 0)
        return -1;
      return filewrite(f, p, n);
    }
```

sys_fstat

```
int sys_fstat(void) {
 struct file *f;
 struct stat *st;

 if (argfd(0, 0, &f) < 0 || argptr(1, (void*)&st,
            sizeof(*st)) < 0)
  return -1;
 return filestat(f, st);
}
```

sys_close

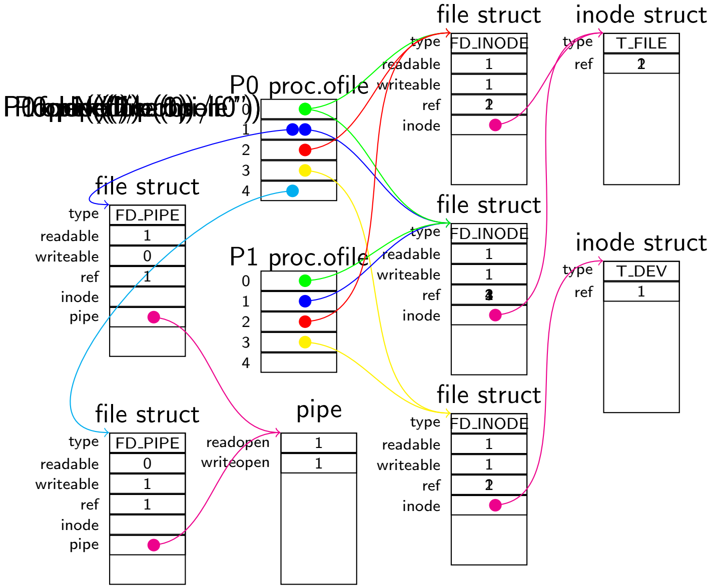
```
int sys_close(void) {
 int fd;
 struct file *f;

 if (argfd(0, &fd, &f) < 0)
  return -1;
 myproc()->ofile[fd] = 0;
 fileclose(f);
 return 0;
}
```

fie layer implementation:

- filedup.
- fileread.
- filewrite.
- filestat.
- fileclose.

ofile/file/inode/pipe structures relation



file layer dispatching

The file layer dispathces to one of the following subsystems:

1. pipe.
2. inode.

functions used, from pipe subsystem and inode layer

Pipe subsystem:

- piperead(pipe *p, char *addr, int len);
- pipewrite(pipe *p, char *addr, int len);
- pipeclose(pipe *, int writeside);

inode subsystem:

- ilock(inode *ip);
- iunlock(inode *ip);
- readi(inode *ip, char *adr, int len);
- writei(inode *ip, char *adr, int len);
- iput(inode *ip);
- stati(inode *ip, stat *stat);
- begin_trans();
- commit_trans();

filedup

```
struct file *filedup(struct file *f) {
 acquire(&ftable.lock);
 if (f->ref < 1)
  panic("filedup");
 f->ref++;
 release(&ftable.lock);
 return f;
}
```

fileread explanation

- According to the type, fileread delegates the read to one of:
  - piperead.
  - readi.
- For FD_INODE type the file position is handled.

fileread

```
int fileread(struct file *f, char *addr, int n) {
  int r;

  if (f->readable == 0)
    return -1;
  if (f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if (f->type == FD_INODE) {
    ilock(f->ip);
    if ((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```

`filewrite` logic

- According to the type, `filewrite` delegates the `write` to one of:
  - `pipewrite`.
  - `writei`.
- For `FD_INODE` type the file position is handled.
- The logic around `writei` is due to the limited log file size.

filewrite

```
int filewrite(struct file *f, char *addr, int n) {
 if (f->writable == 0) return -1;
 if (f->type == FD_PIPE) return pipewrite(f->pipe, addr,
 if (f->type == FD_INODE) {
  int max = ((LOGSIZE-1-1-2) / 2) * 512;
  for (int i=0; i < n; ) {
   int n1 = n - i;
   if (n1 > max) n1 = max;
   begin_op();  ilock(f->ip);
   if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
    f->off += r;
   iunlock(f->ip);  end_op();
   if (r < 0)  break;
   if (r != n1)  panic("short filewrite");
   i += r;
  }
  return i == n ? n : -1;
 }
 panic("filewrite");
```

`close` logic

- Reference count is updated.
- If reference count drops to zero we delegate to one of:
  - `pipeclose`.
  - `iput`.
- Note due to `iput` we have to `releaseaquire`, hence the need to copy the `file` structure.

filestat

```
5952 int filestat(struct file *f, struct stat *st) {
  if (f->type == FD_INODE) {
    ilock(f->ip);
    stati(f->ip, st);
    iunlock(f->ip);
    return 0;
  }
  return -1;
}
```

fileclose

```
5914  void fileclose(struct file *f) {
        acquire(&ftable.lock);
        if (f->ref < 1) panic("fileclose");
        if (--f->ref > 0) {
          release(&ftable.lock);
          return;
        }
        struct ff = *f;
        f->ref = 0;
        f->type = FD_NONE;
        release(&ftable.lock);
        if (ff.type == FD_PIPE)
          pipeclose(ff.pipe, ff.writable);
        else if (ff.type == FD_INODE) {
          begin_op();  iput(ff.ip); end_op();
        }
      }
```

file structures pool

- There is no dynamic allocation for `file` structures.
- There is a vector with all `file` structures, protected with a spinlock.
- A slot is free if the `ref` field is zero.

```
5863  struct {
        struct spinlock lock;
        struct file file[NFILE];
      } ftable;
```

- Allocation is done in sys_open() and sys_pipe().

filealloc: Allocating file struct from pool

```
struct file *filealloc(void) {
 struct file *f;

 acquire(&ftable.lock);
 for (f = ftable.file;f < ftable.file + NFILE;f++) {
  if (f->ref == 0) {
   f->ref = 1;
   release(&ftable.lock);
   return f;
  }
 }
 release(&ftable.lock);
 return 0;
}
```

Who sets values in the `file` structure

One of:

- `sys_open()`, by calling the inode layer.
- `sys_pipe()`, by calling the pipe layer.

pipe logic

- Delegate to `pipealloc`.
- Hide the returned `file` pointers in the `ofile` vector.
- (I tend to think `pipealloc` should just returned `pipe` pointer and it is our function job to allocate the `file` structure.

sys_pipe

```
6551 int sys_pipe(void) {
  int *fd;
  struct file *rf, *wf;
  int fd1;
  if (argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)  return
  if (pipealloc(&rf, &wf) < 0)  return -1;
  int fd0 = -1;
  if((fd0=fdalloc(rf)) < 0 || (fd1=fdalloc(wf)) < 0) {
   if (fd0 >= 0)
    myproc()->ofile[fd0] = 0;
   fileclose(rf);
   fileclose(wf);
   return -1;
  }
  fd[0] = fd0;
  fd[1] = fd1;
  return 0;
}
```