Outline

3 File I/O 3-1

3.1 File I/O overview 3-3

3.2 open(), read(), write(), and close() 3-7

3.3 The file offset and lseek() 3-21

3.4 Atomicity 3-30

3.5 Relationship between file descriptors and open files 3-35

3.6 Duplicating file descriptors 3-44

3.7 File status flags (and fcntl()) 3-50

3.8 Other file I/O interfaces 3-58

Relationship between file descriptors and open files

- Multiple file descriptors can refer to same open file
- 3 kernel data structures describe relationship:

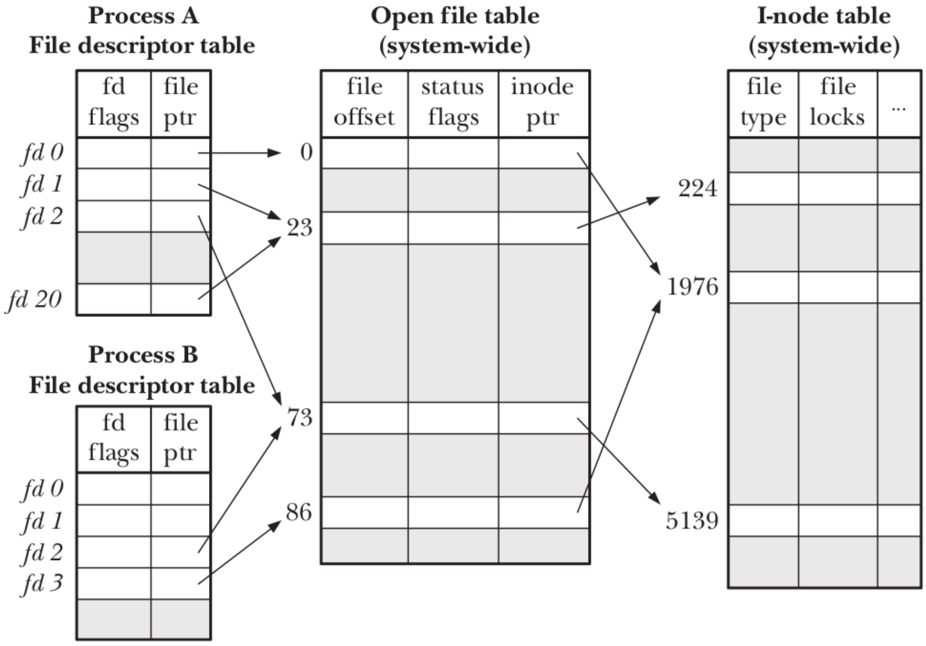

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

File descriptor table

Per-process table with one entry for each FD opened by process:

- Flags controlling operation of FD (close-on-exec flag)
- Reference to open file description
- *struct fdtable* in `include/linux/fdtable.h`

Open file table (table of open file descriptions)

System-wide table, one entry for each open file on system:

- File offset
- File access mode (R / W / R-W, from *open()*)
- File status flags (from *open()*)
- Signal-driven I/O settings
- Reference to inode object for file
- *struct file* in `include/linux/fs.h`

Following terms are commonly treated as synonyms:

- **open file description (OFD)** (POSIX)
- **open file table entry** or **open file handle**
  - (These two are ambiguous; POSIX terminology is preferable)

(In-memory) inode table

System-wide table drawn from file inode information in filesystem:

- File type (regular file, FIFO, socket, . . .)
- File permissions
- Other file properties (size, timestamps, . . .)
- *struct inode* in `include/linux/fs.h`

Duplicated file descriptors (intraprocess)

A process may have multiple FDs referring to same OFD

- Achieved using *dup()* or *dup2()*

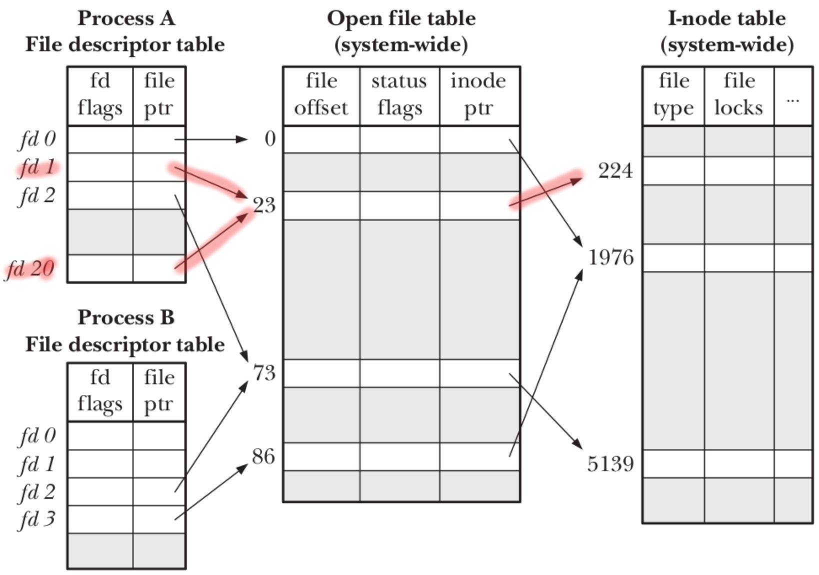

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Duplicated file descriptors (between processes)

Two processes may have FDs referring to same OFD

- Can occur as a result of *fork()*

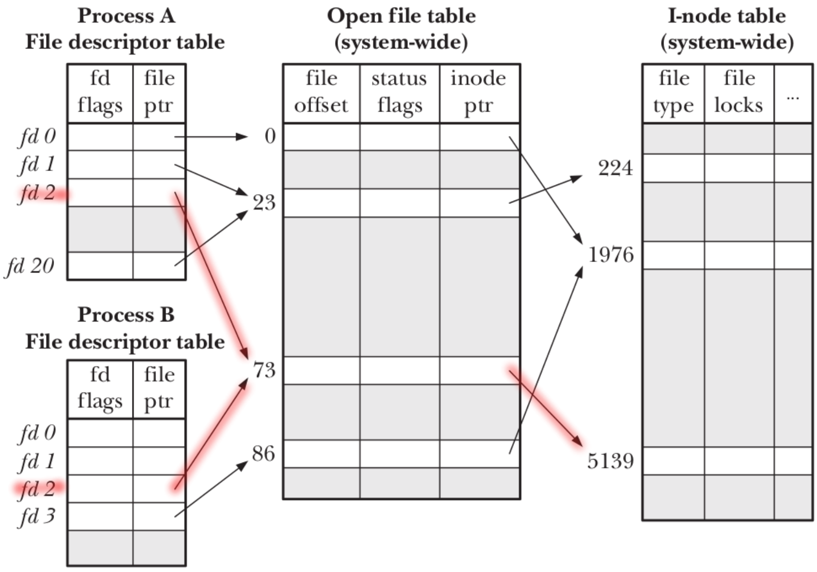

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Distinct open file table entries referring to same file

Two processes may have FDs referring to distinct OFDs that refer to same inode

- Two processes independently *open()*ed same file

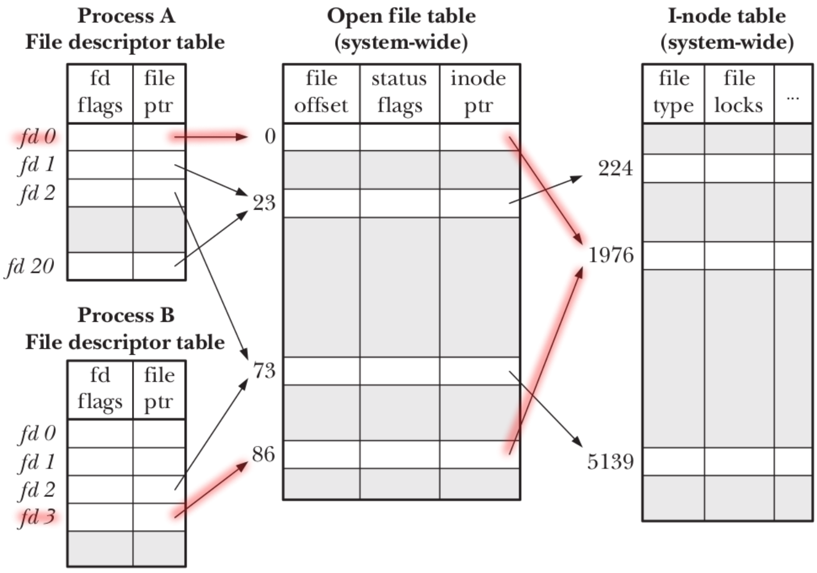

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Why does this matter?

- Two different FDs referring to same OFD share file offset
  - (File offset == location for next *read()*/*write()*)
  - Changes (*read()*, *write()*, *lseek()*) via one FD visible via other FD
  - Applies to both intraprocess & interprocess sharing of OFD
- Similar scope rules for status flags (`O_APPEND`, `O_SYNC`, ...)
  - Changes via one FD are visible via other FD
    - (`fcntl(F_SETFL)` and `fcntl(F_GETFL)`)
- Conversely, changes to FD flags (held in FD table) are private to each process and FD
- *kcmp(2)* `KCMP_FILE` operation can be used to test if two FDs refer to same OFD
  - Linux-specific

[TLPI §5.4]

Outline

3 File I/O 3-1

3.1 File I/O overview 3-3

3.2 open(), read(), write(), and close() 3-7

3.3 The file offset and lseek() 3-21

3.4 Atomicity 3-30

3.5 Relationship between file descriptors and open files 3-35

3.6 Duplicating file descriptors 3-44

3.7 File status flags (and fcntl()) 3-50

3.8 Other file I/O interfaces 3-58

A problem

```
./myprog > output.log 2>&1
```

- What does the shell syntax, `2>&1`, do?
- How does the shell do it?
- Open file twice, once on FD 1, and once on FD 2?
  - FDs would have separate OFDs with distinct file offsets ⇒ standard output and error would overwrite
  - File may not even be *open()*-able:
    - e.g., `./myprog 2>&1 | less`
- Need a way to create duplicate FD that refers to same OFD

[TLPI §5.5]

Duplicating file descriptors

```
#include <unistd.h>
int dup(int oldfd);
```

- Arguments:
  - *oldfd*: an existing file descriptor
- Returns new file descriptor (on success)
- **New file descriptor is guaranteed to be lowest available**

Duplicating file descriptors

- FDs 0, 1, and 2 are normally always open, so shell can achieve 2>&1 redirection by:

```
close(STDERR_FILENO);          /* Frees FD 2 */
newfd = dup(STDOUT_FILENO);    /* Reuses FD 2 */
```

- But what if FD 0 was closed?

Duplicating file descriptors

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- Like *dup()*, but uses *newfd* for the duplicate FD
  - **Silently** closes *newfd* if it was open
  - Closing + reusing *newfd* is done atomically
    - Important: otherwise *newfd* might be re-used in between
  - Does nothing if *newfd* == *oldfd*
  - Returns new file descriptor (i.e., *newfd*) on success
- `dup2(STDOUT_FILENO, STDERR_FILENO);`
- See *dup2(2)* man page for more details

[TLPI §5.5]

Outline

3	File I/O	3-1
3.1	File I/O overview	3-3
3.2	open(), read(), write(), and close()	3-7
3.3	The file offset and lseek()	3-21
3.4	Atomicity	3-30
3.5	Relationship between file descriptors and open files	3-35
3.6	Duplicating file descriptors	3-44
3.7	File status flags (and fcntl())	3-50
3.8	Other file I/O interfaces	3-58

File status flags

- Control semantics of I/O on a file
  - (`O_APPEND`, `O_NONBLOCK`, `O_SYNC`, . . .)
- Associated with open file description
- Set when file is opened
- Can be retrieved and modified using *fcntl()*

[TLPI §5.3]

fcntl(): file control operations

```
#include <fcntl.h>
int fcntl(int fd, int cmd /* , arg */ );
```

Performs control operations on an open file

- Arguments:
  - *fd*: file descriptor
  - *cmd*: the desired operation
  - *arg*: optional, type depends on *cmd*
- Return on success depends on *cmd*; -1 returned on error
- Many types of operation
  - file locking, signal-driven I/O, file descriptor flags . . .

Retrieving file status flags and access mode

- Retrieving flags (both access mode and status flags)

```
flags = fcntl(fd, F_GETFL);
```

- Check access mode

```
amode = flags & O_ACCMODE;
if (amode == O_RDONLY || amode == O_RDWR)
    printf("File is readable\n");
```

- ⚠ 'read' and 'write' are not separate bits!

```
if (flags & O_RDONLY)         /* Wrong!! */
    printf("File is readable\n");
```

  - Access mode is a 2-bit field that is an enumeration:
    - 00 == `O_RDONLY`
    - 01 == `O_WRONLY`
    - 10 == `O_RDWR`
- Access mode can't be changed after file is opened

Retrieving and modifying file status flags

- Retrieving file status flags

```
flags = fcntl(fd, F_GETFL);
if (flags & O_NONBLOCK)
    printf("Nonblocking I/O is in effect\n");
```

- Setting a file status flag

```
flags = fcntl(fd, F_GETFL);     /* Retrieve flags */
flags |= O_APPEND;              /* Set "append" bit */
fcntl(fd, F_SETFL, flags);      /* Modify flags */
```

  - ⚠ Not thread-safe...
    - (But in many cases, flags can be set when FD is created, e.g., by *open()*)

- Clearing a file status flag

```
flags = fcntl(fd, F_GETFL);     /* Retrieve flags */
flags &= ~O_APPEND;             /* Clear "append" bit */
fcntl(fd, F_SETFL, flags);      /* Modify flags */
```

Exercise

1. Show that duplicate file descriptors share file offset and file status flags by writing a program ([template: `fileio/ex.fd_sharing.c`]) that:
   - Opens an existing file (supplied as *argv[1]*) and duplicates (*dup()*) the resulting file descriptor, to create a second file descriptor.
   - Displays the file offset and the state of the `O_APPEND` file status flag via the first file descriptor.
     - Initially the file offset will be zero, and the `O_APPEND` flag will not be set
   - Changes the file offset (*lseek()*) and enables (turns on) the `O_APPEND` file status flag (*fcntl()*) via the second file descriptor.
   - Displays the file offset and the state of the `O_APPEND` file status flag via the first file descriptor.

   Hints:
   - Remember to update the `Makefile`!
   - `while inotifywait -q . ; do echo; echo; make; done`

Exercise

2. Read about the `KCMP_FILE` operation in the *kcmp(2)* man page. Extend the program created in the preceding exercise to use this operation to verify that the two file descriptors refer to the same open file description (i.e., use *kcmp(getpid(), getpid(), KCMP_FILE, fd1, fd2)*). Note: because there is currently no *kcmp()* wrapper function in glibc, you will have to write one yourself using *syscall(2)*:

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/kcmp.h>

static int kcmp(pid_t pid1, pid_t pid2, int type,
                unsigned long idx1, unsigned long idx2)
{
    return syscall(SYS_kcmp, pid1, pid2, type,
                   idx1, idx2);
}
```