File I/O, File Sharing

Hongik University

Eunsung Jung

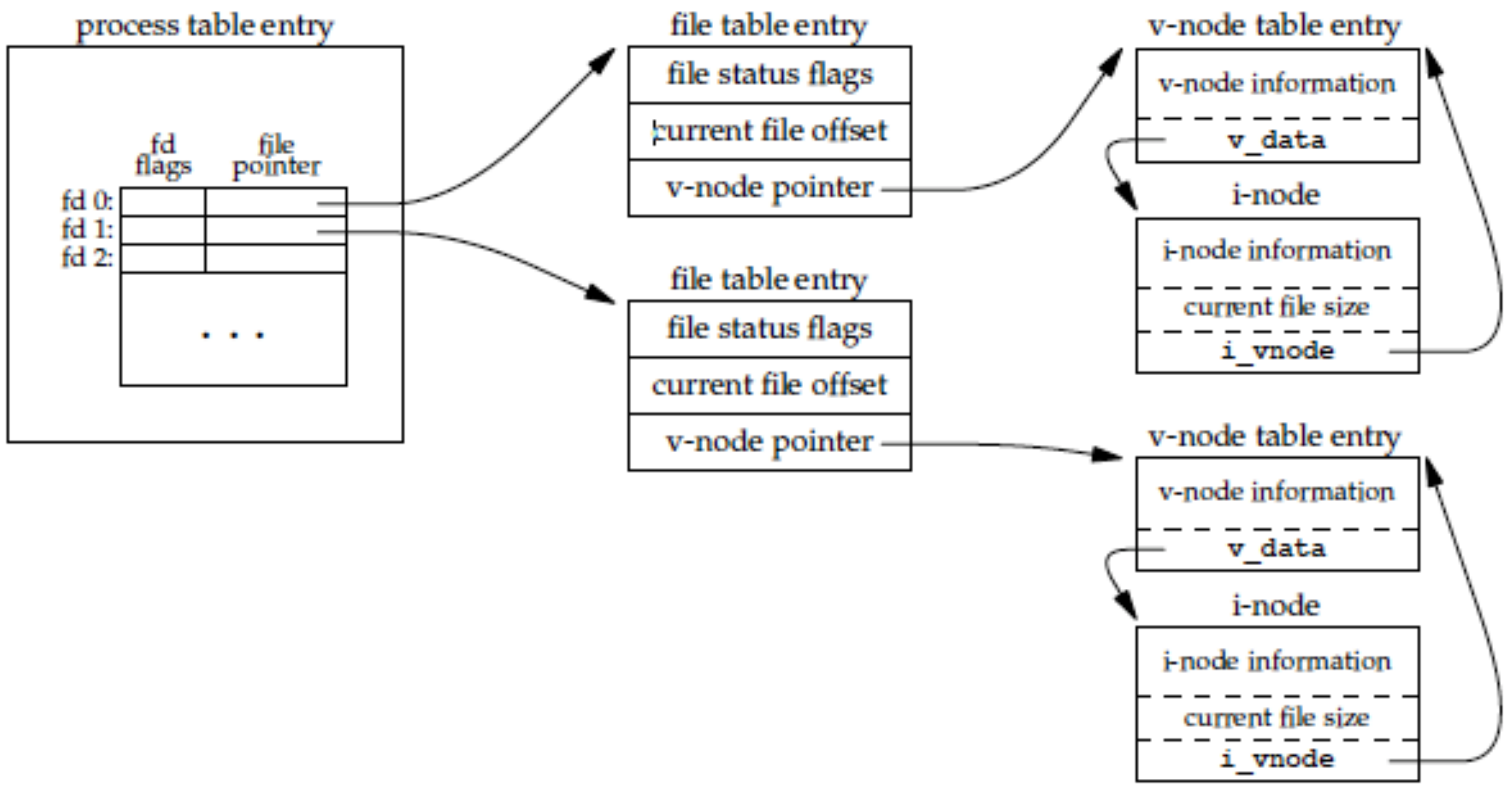
Disclaimer: The slides are borrowed from many sources!

File Sharing

- Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (See: Stevens, pp 75)

Kernel data structures for open files

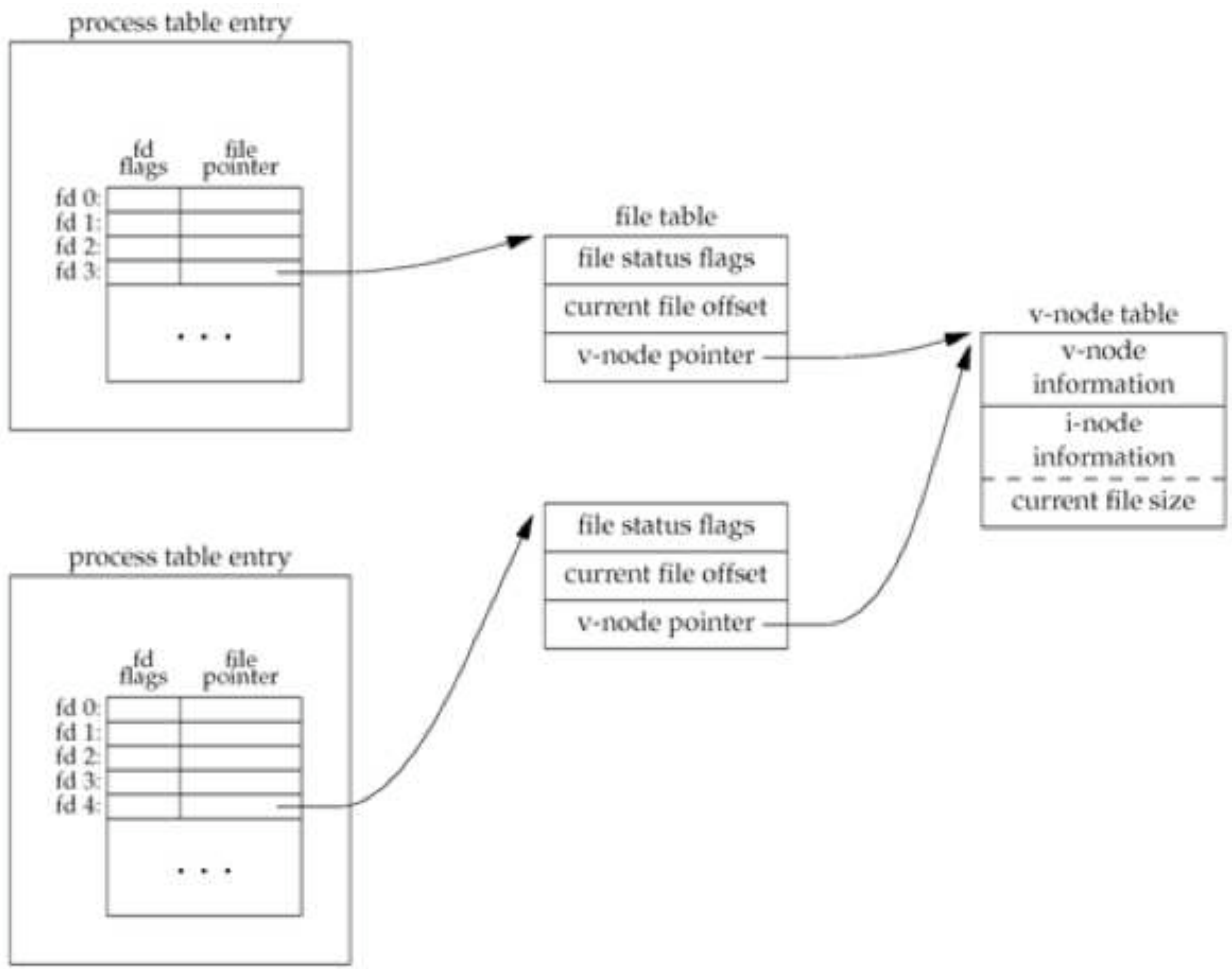
- Each process table entry has a table of file descriptors, which contain
  - the file descriptor flags (ie FD_CLOEXEC, see fcntl(2))
  - a pointer to a file table entry



Kernel data structures for open files

- the kernel maintains a file table; each entry contains
  - file status flags (O_APPEND, O_SYNC, O_RDONLY, etc.)
  - current offset
  - pointer to a vnode table entry
- a vnode structure (for a open file) contains
  - vnode information
  - inode information (such as current file size)

Kernel data structures for open files



What will happen in File Sharing?

- After each write completes, the current file offset in the file table entry is incremented.
  - If current file offset > current file size, change current file size in i-node table entry.
- If file was opened O_APPEND set corresponding flag in file status flags in file table. For each write, current file offset is first set to current file size from the i-node entry.
- lseek simply adjusts current file offset in file table entry
- To lseek to the end of a file, just copy current file size into current file offset.

Atomic Operations

- In order to ensure consistency across multiple writes, we require *atomicity* in some operations. An operation is atomic if either all of the steps are performed or none of the steps are performed.
- Suppose UNIX didn't have O_APPEND (early versions didn't). To append, you'd have to do this:

```
if (lseek(fd, 0L, 2) < 0) {        /* position to EOF */
    fprintf(stderr, "lseek error\n");
    exit(1);
}

if (write(fd, buff, 100) != 100) { /* ...and write */
    fprintf(stderr, "write error\n");
    exit(1);
}
```

- What if another process was doing the same thing to the same file?

pread(2) and pwrite(2)

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, void *buf, size_t count, off_t offset);
```

Both return number of bytes read/written, -1 on error

- Atomic read/write at offset without invoking lseek(2).
- Current offset is not updated.

Concurrent write()

- What will happen if two processes do
  - Write(“1234”); by p1
  - Write(“5678”); by p2

- Results can be
  - 15234678?
  - Or what?

dup(2) and dup2(2)

```
#include <unistd.h>

int dup(int oldd);
int dup2(int oldd, int newd);
```

Both return new file descriptor if OK, -1 on error

- An existing file descriptor can be duplicated with dup(2) or duplicated to a particular file descriptor value with dup2(2). As with open(2), dup(2) returns the lowest numbered unused file descriptor.

dup(2) and dup2(2)

- newfd = dup(1);
  - We assume that the next available descriptor is 3. What about 0, 1, 2?
- Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset.
  - But each fd has its own fd flags. (e.g. close-on-exec fd flag). File status flags?

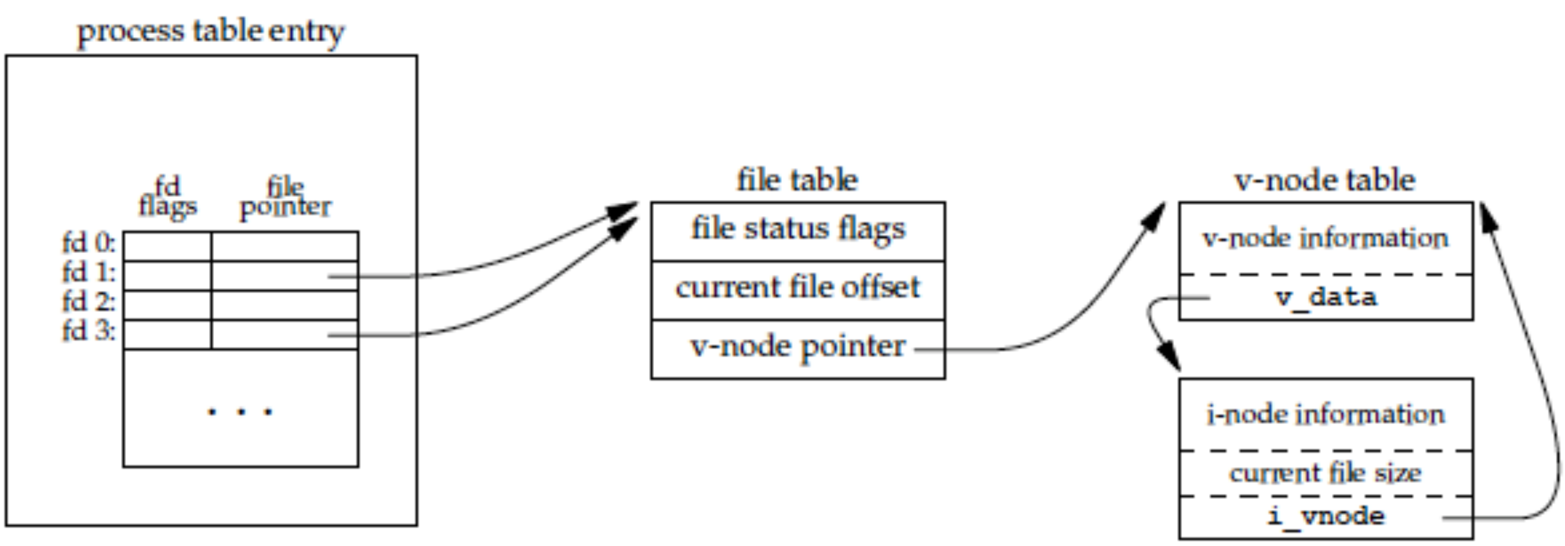


Figure 3.9 Kernel data structures after dup(1)

File Status Flag

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only
O_SEARCH	open directory for searching only
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

sync, fsync, fdatasync

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);

                                            Returns: 0 if OK, –1 on error

void sync(void);
```

- UNIX System have a buffer cache or page cache in the kernel through which most disk I/O passes.
- The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.
- The function sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update . This guarantees regular flushing of the kernel's block buffers. The command sync (1) also calls the sync function.

fcntl(2)

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int filedes, int cmd, ...  /* int arg */);
```

Returns: depend on *cmd* if OK, -1 on error

- fcntl(2) is on of those ”catch-all” functions with a myriad of purposes. They all relate to changing properties of an already open file.

fcntl(2)

cmd	effect	return value
F_DUPFD	duplicate *filedes* (FD_CLOEXEC file descriptor flag is cleared	new filedes
F_GETFD	get the file descriptor flags for *filedes*	descriptor flags
F_SETFD	set the file descriptor flags to the value of the third argument	not -1
F_GETFL	get the file status flags	status flags
F_SETFL	set the file status flags	not -1

Lab:
Figure 3.11 Print file flags for specified descriptor

- Takes a single command-line argument that specifies a file descriptor and prints a description of selected file flags for that descriptor
- Can you interpret the following results?
  - 5<>temp.foo opens the file temp.foo for reading and writing on file descriptor 5.

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```

ioctl(2)

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* 4.3+BSD */

int ioctl(int filedes, int request, ...);
```

Returns: -1 on error, something else if OK

- Another catch-all function, this one is designed to handle device specifics that can't be specified via any of the previous function calls. For example, terminal I/O, magtape access, socket I/O, etc. Mentioned here mostly for completeness's sake.

/dev/fd

```
[esjung@hpclab fileio]$ ls -l /dev/stdin /dev/stdout /dev/stderr
lrwxrwxrwx. 1 root root 15 Aug 19 11:37 /dev/stderr -> /proc/self/fd/2
lrwxrwxrwx. 1 root root 15 Aug 19 11:37 /dev/stdin -> /proc/self/fd/0
lrwxrwxrwx. 1 root root 15 Aug 19 11:37 /dev/stdout -> /proc/self/fd/1
[esjung@hpclab fileio]$ ls -l /dev/fd/
total 0
lrwx------. 1 esjung hpclab 64 Sep 21 15:34 0 -> /dev/pts/9
lrwx------. 1 esjung hpclab 64 Sep 21 15:34 1 -> /dev/pts/9
lrwx------. 1 esjung hpclab 64 Sep 21 15:34 2 -> /dev/pts/9
lr-x------. 1 esjung hpclab 64 Sep 21 15:34 3 -> /proc/114140/fd
[esjung@hpclab fileio]$ echo first >file1
[esjung@hpclab fileio]$ echo third >file2
[esjung@hpclab fileio]$ echo second | cat file1 /dev/fd/0 file2
```

- /proc file system?
- /dev/pts?
  - Pseudo terminal
  - cat test > /dev/pts/9 → ?
  - cat test > /dev/pts/8 → ?
- Results of the last command line?

/dev/fd

```
[esjung@hpclab fileio]$ echo second | cat file1 /dev/fd/0 file2
first
second
third
[esjung@hpclab fileio]$
```

Lab#3

- Implement a program which behaves as follows.
  - If there is no files named 1.dat ~ 1024.dat under the current directory, create them.
  - If there is a file,
    - If no content, put 1.
    - Otherwise, increment it.
    - If the content is 10, remove it.
- Build a team for open source project.
  - Strict requirement: 2-3 people per team.
  - Consider what open source project you will be involved in.