Advanced Programming in the UNIX Environment

Week 02, Segment 4: File Sharing

Department of Computer Science
Stevens Institute of Technology

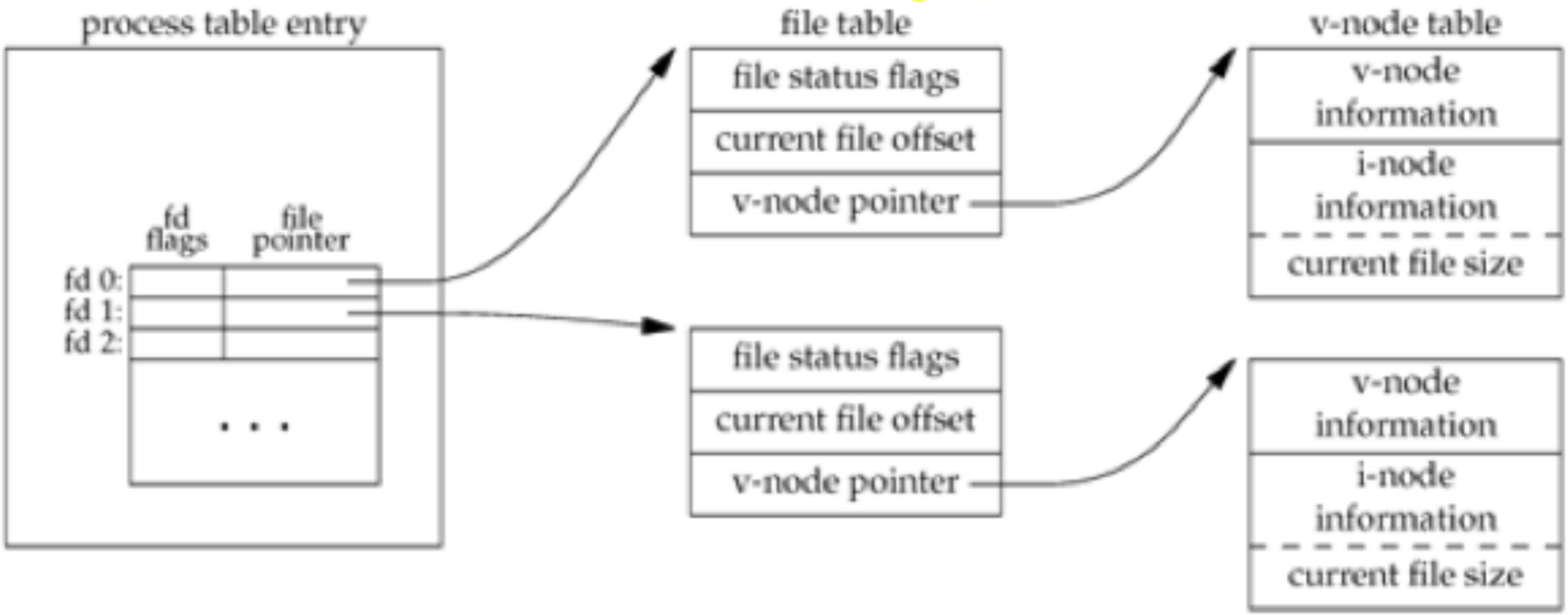
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

File Sharing

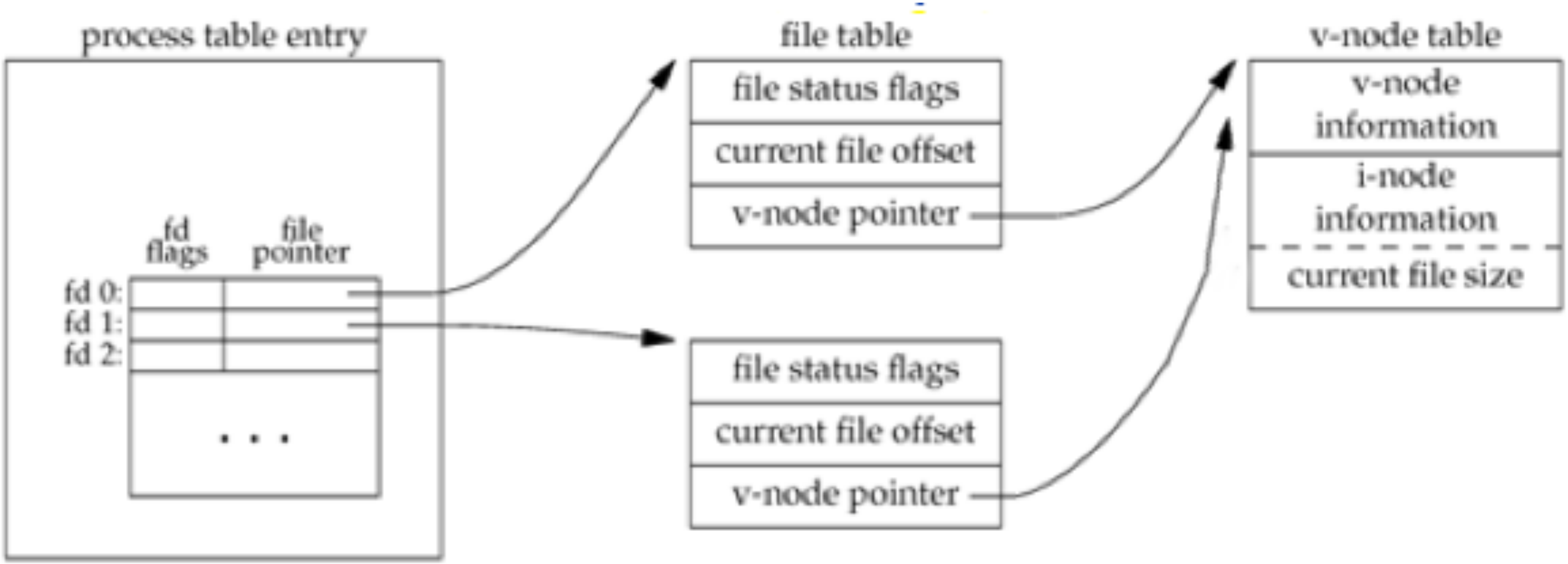
Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files.

- each *process table entry* has a table of file descriptors, which contain:
  - the file descriptor flags (e.g. `FD_CLOEXEC`, see `fcntl(2)`)
  - a pointer to a file table entry
- 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 contains
  - vnode information
  - inode information (such as current file size)

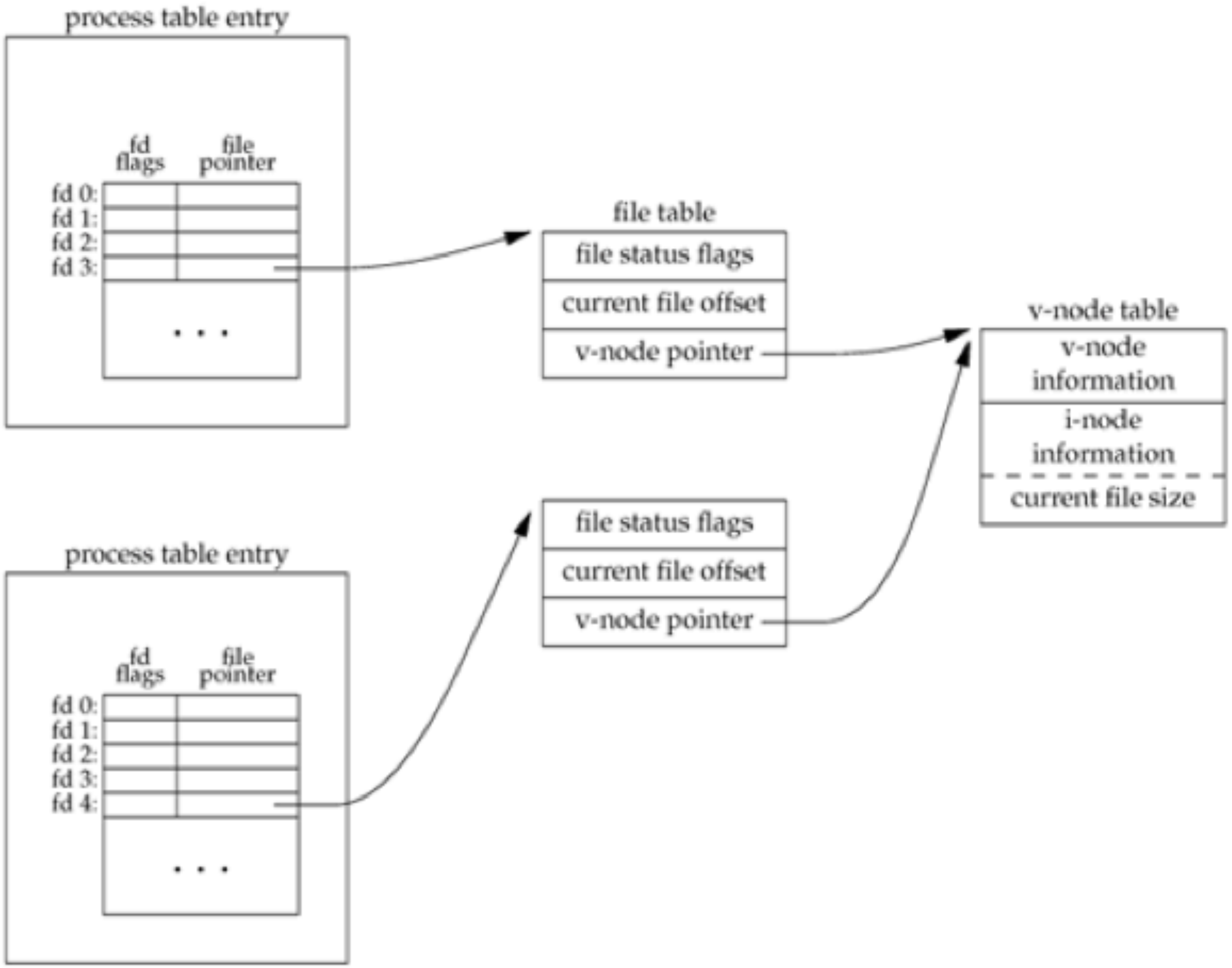
File Sharing

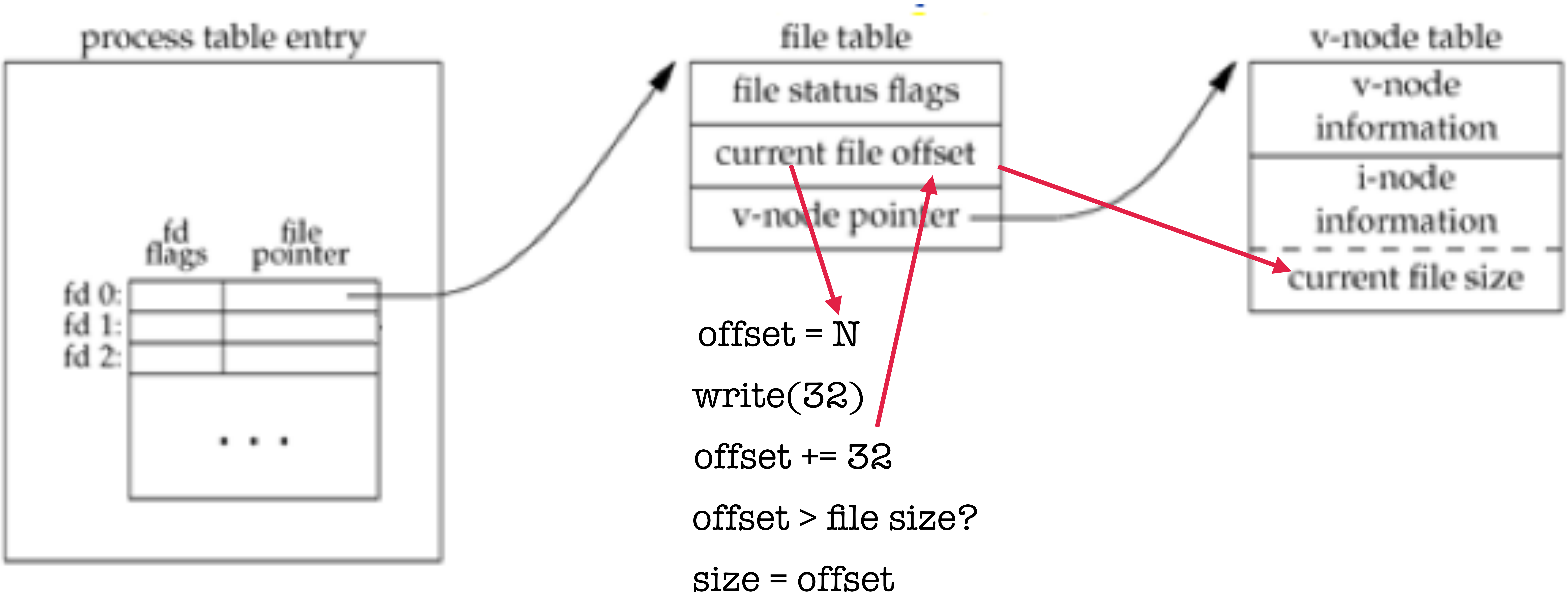


File Sharing



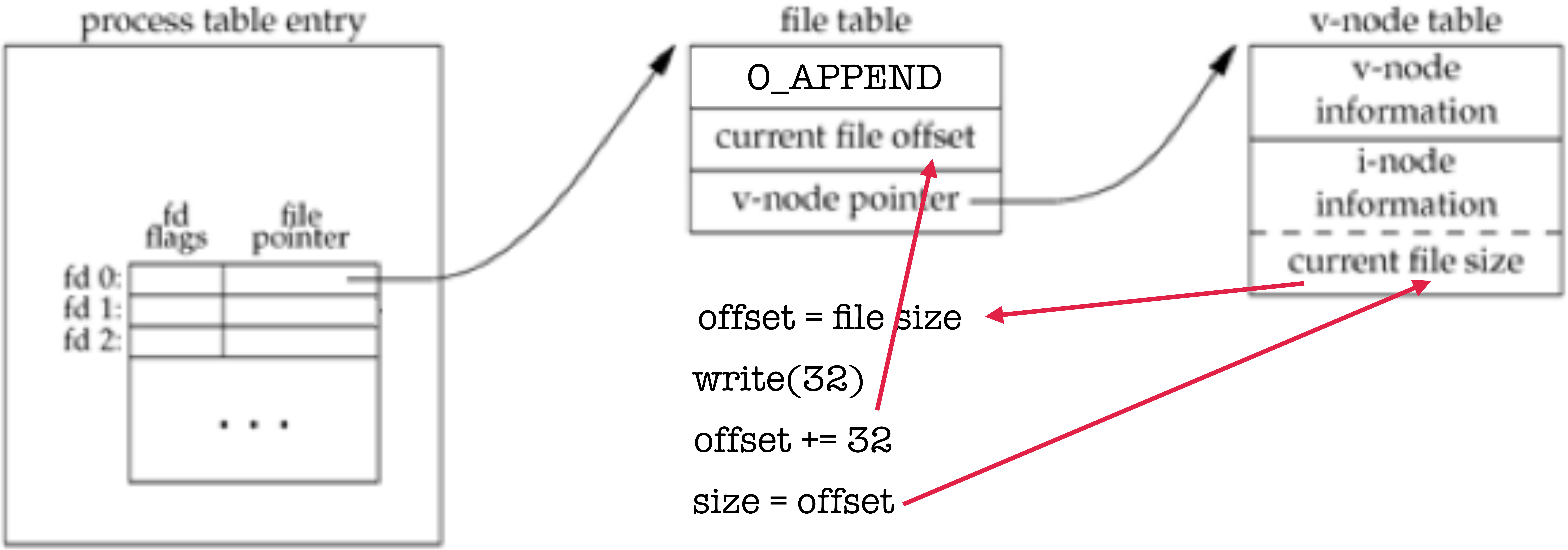
File Sharing





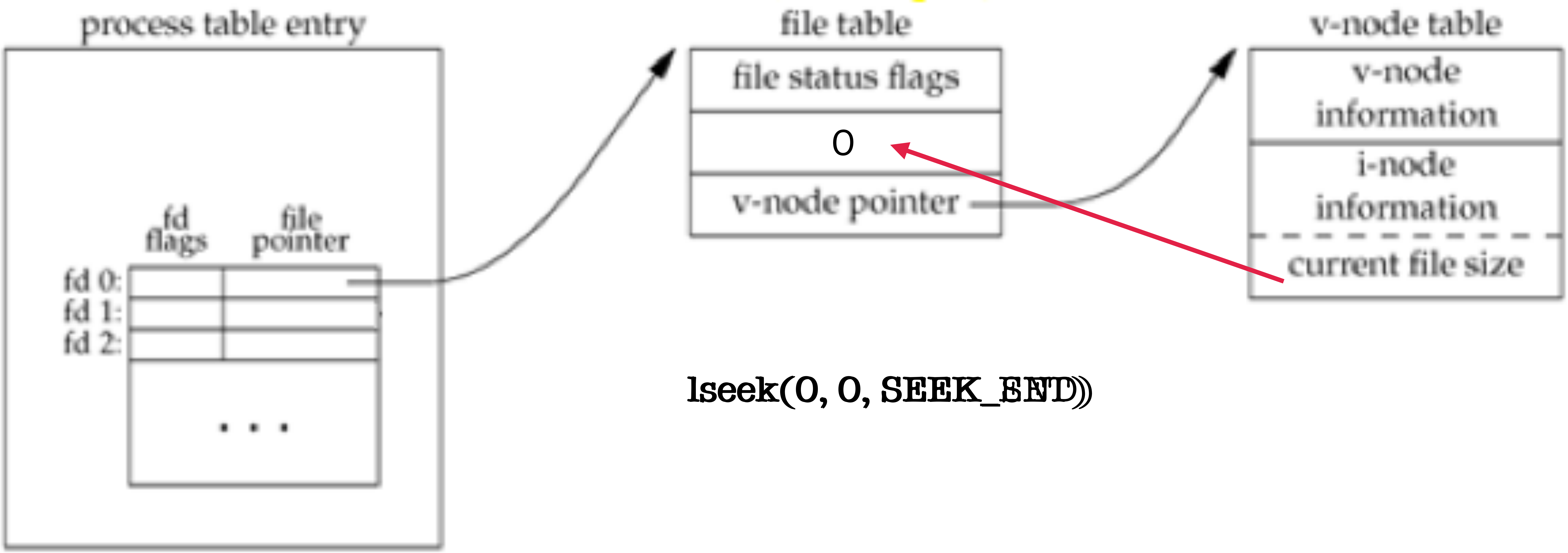
After each write(2) completes, the current file offset in the file table entry is incremented.

If the current file offset is larger than the current file size, we change the current file size in i-node table entry.



If file was opened with O_APPEND, set corresponding flag in file status flags in file table.

For each write, the current file offset is first set to the current file size from the i-node entry.



lseek(2) merely adjusts the current file offset in file table entry.

To seek to the end of a file, just copy current file size into current file offset.

To seek to the beginning of the file, simply set the offset to 0.

Atomic Operations

Early versions of Unix didn't support `O_APPEND`. Instead, you had to:

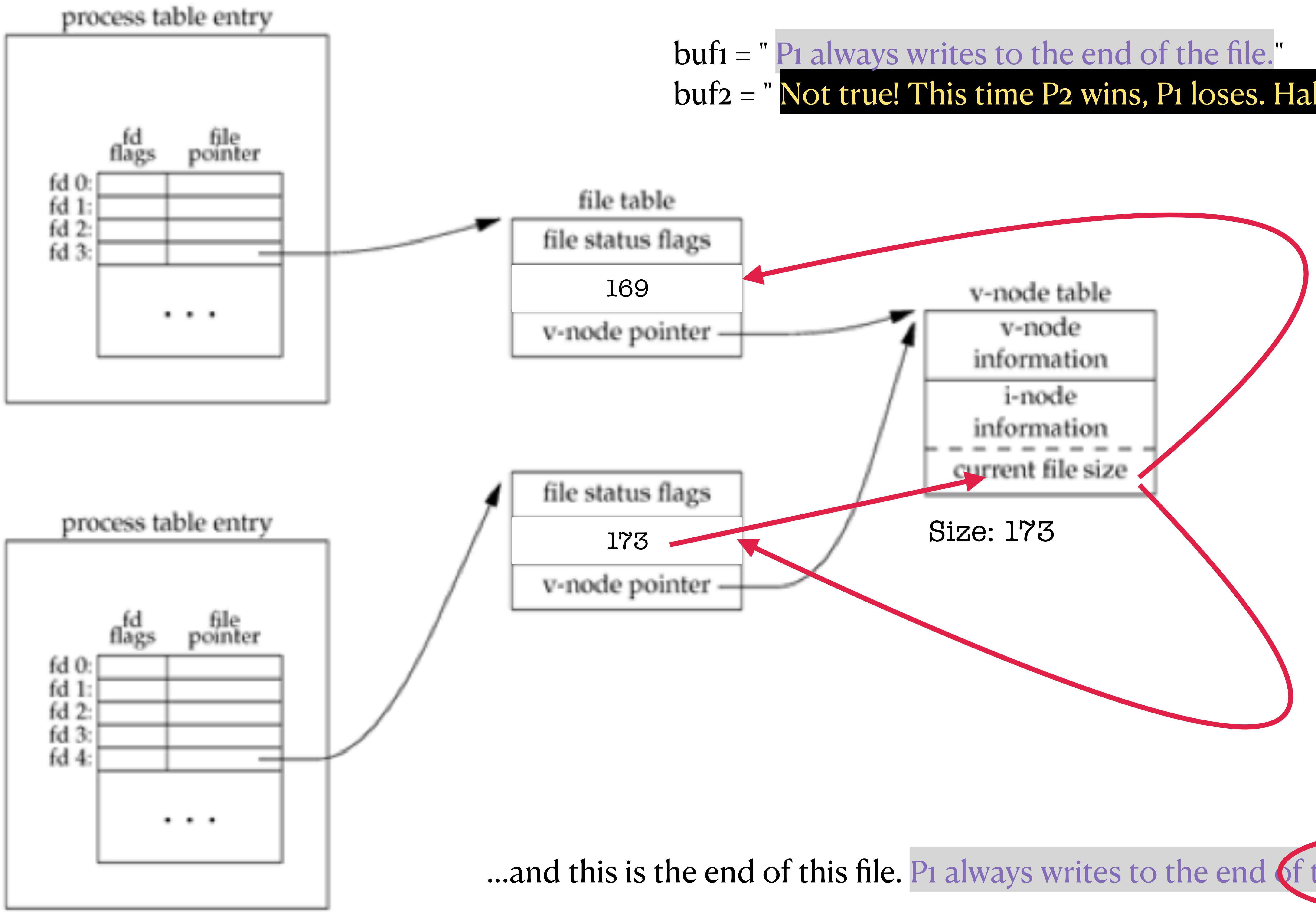
```
if (lseek(fd, 0, SEEK_END) < 0) {
     /* error */
}

if (write(fd, but, len) != len) {
     /* error */
}
```

Atomic Operations

What if another process did the same thing?

buf1 = " P1 always writes to the end of the file."
buf2 = " Not true! This time P2 wins, P1 loses. Haha!"



...and this is the end of this file. P1 always writes to the end of the file.aha!

P1:
lseek(fd, 0, SEEK_END)

current offset = 128

write(fd, buf1, 41)

new offset = 128 + 41 = 169

P2:
lseek(fd, 0, SEEK_END)

current offset = 128

write(fd, buf2, 45)

new offset = 128 + 45 = 173

size = 173

Atomic Operations

O_APPEND solves the case for writing to the end, but what if we want to write atomically anywhere else in the file?

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t num, off_t offset);

ssize_t pwrite(int fd, void *buf, size_t num, off_t offset);

                    Returns: number of bytes read/written, -1 on error
```

Note: current offset is not changed.

Terminal — 80×24

```
apue$
```

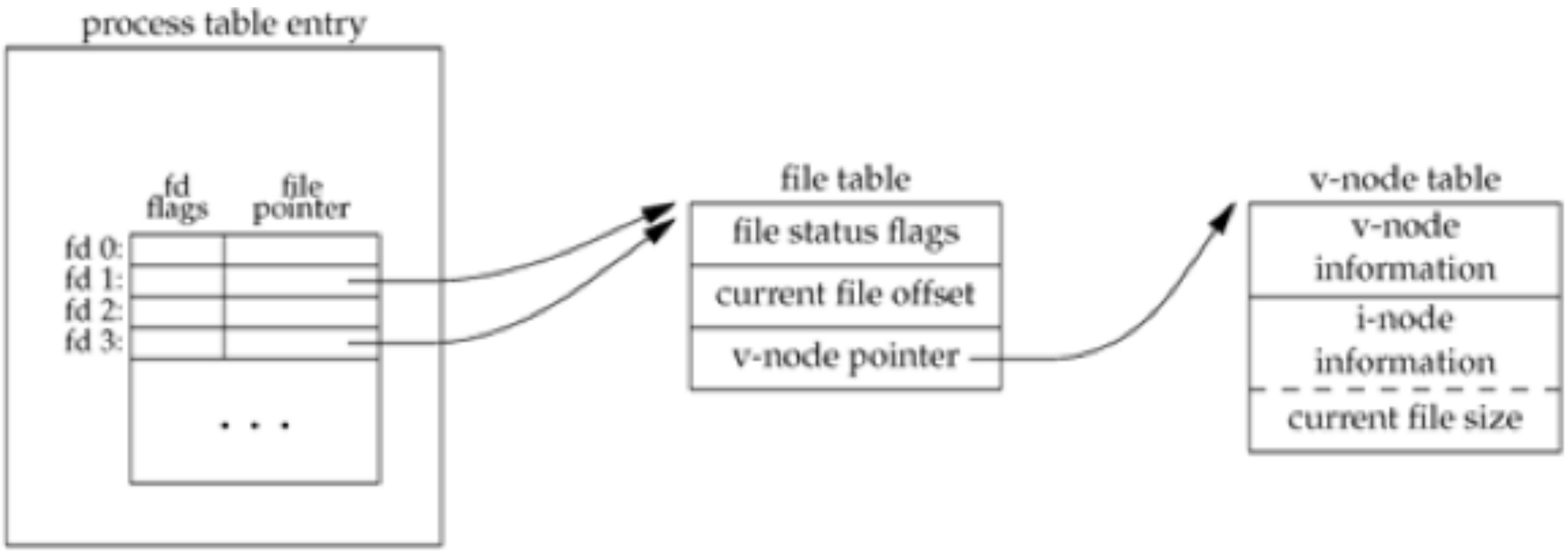
File Descriptor Duplication

```
#include <unistd.h>

int dup(int oldfd);

int dup2(int oldfd, int newfd);
```

Returns: newfd, -1 on error



Terminal — 80×24

```
$ ssh apue
Last login: Sun Sep  6 18:26:58 2020 from 10.0.2.2
NetBSD 9.0 (GENERIC) #0: Fri Feb 14 00:06:28 UTC 2020

Welcome to NetBSD!

apue$
```

File Descriptor Control

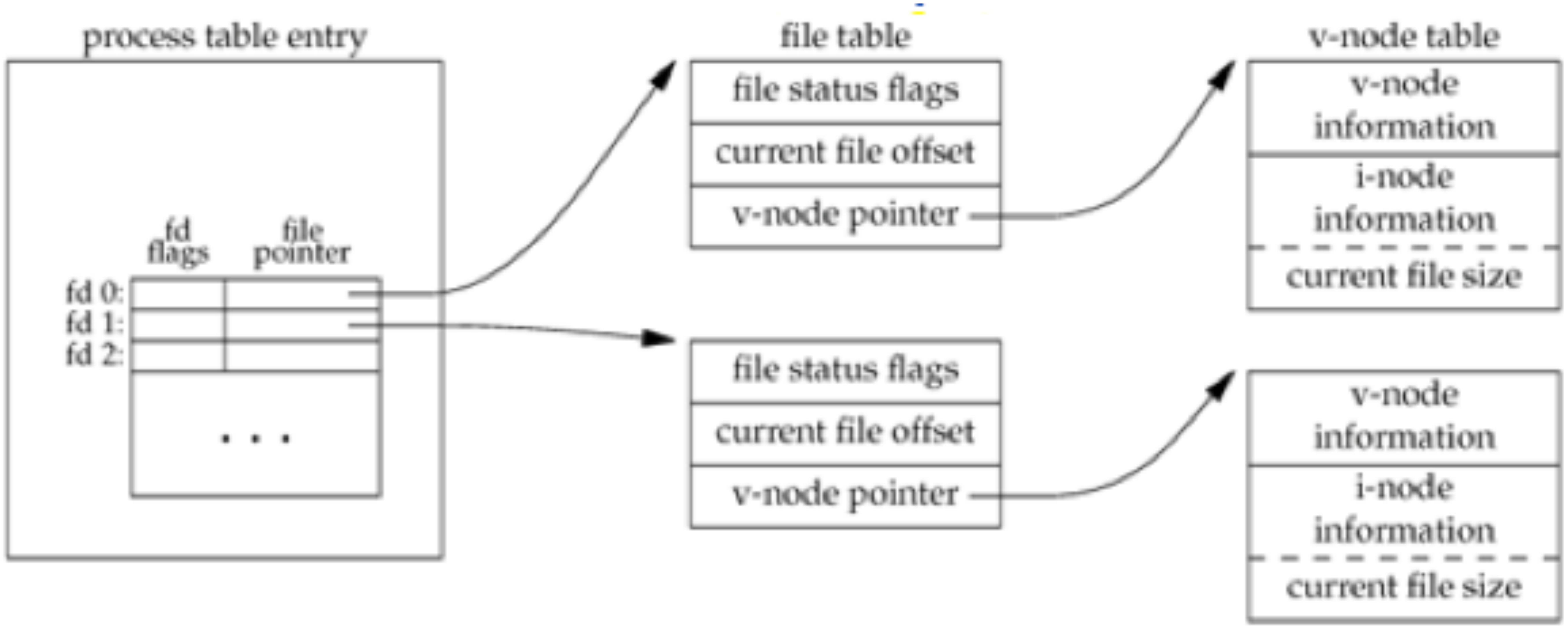
```
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
```

Returns: depends on *cmd*, -1 on error

fcntl(2) is one of those "catch-all" functions with a myriad of purposes. Here, they all relate to changing properties of an already open file. Some of them are:

- F_DUPFD - duplicate file descriptors
- F_GETFD - get file descriptor flags
- F_SETFD - set file descriptor flags
- F_GETFL - get file status flags
- F_SETFL - set file status flags
- ...



Terminal — 80×24

```
$ ssh apue
Last login: Sun Sep  6 20:25:42 2020 from 10.0.2.2
NetBSD 9.0 (GENERIC) #0: Fri Feb 14 00:06:28 UTC 2020

Welcome to NetBSD!

apue$ 
```

File Descriptor Control

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

Returns: depends on *request*, -1 on error

Another catch-all function, this one is designed to handle device specifics that can't be specified via any of the previous function calls.

Examples include terminal I/O, magtape access, socket I/O, etc.

Mentioned here mostly for completeness's sake.

See also: tty(4), ioctl_list(2) (Linux)

Terminal — 80×24

```
apue$
```

Conclusion

We've covered the five syscalls on which all basic Unix I/O is based:

`open(2)`, `close(2)`, `read(2)`, `write(2)`, and `lseek(2)`

We've looked at the kernel structures used to implement these calls and discussed the impact of multiple, simultaneous processes accessing the same files.

We've seen a bunch of odd (or at least surprising) things and have written some code to clarify our understanding.

Let's put all this knowledge into practice:

https://stevens.netmeister.org/631/f20-hw1.html

Additional Reading

- https://en.wikipedia.org/wiki/File_descriptor
- https://en.wikipedia.org/wiki/Sparse_file
- https://tldp.org/LDP/abs/html/io-redirection.html
- https://unix.stackexchange.com/questions/98958/linux-nuisance-dev-stdin-doesnt-work-with-sockets
- https://marc.info/?l=ast-users&m=120978595414990&w=2