UNIX File I/O

File I/O system calls

open

Example (taken from `man open` in Linux):

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *pathname = "/tmp/file";
...
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

Another example – creating a lock file:

```
fd = open("/var/run/myprog.pid", O_WRONLY | O_CREAT | O_EXCL, 0644);
```

creat

Redundant: `creat(path, mode)` is equivalent to `open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)`

And it should have been called `create`, says Ken Thompson

close

```
int close(int fildes);
```

lseek

```
off_t lseek(int fildes, off_t offset, int whence);
```

- If whence is SEEK_SET, the file offset shall be set to offset bytes.
- If whence is SEEK_CUR, the file offset shall be set to its current location plus offset.
- If whence is SEEK_END, the file offset shall be set to the size of the file plus offset.

read

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- returns number of bytes read, 0 if end of file, –1 on error

Note:

- Number of bytes read may be less than the requested nbyte
- `read()` may block forever on a “slow” read from pipes, FIFOs (aka named pipes), sockets, or keyboard
- For sockets, `read(socket, buf, nbyte)` is equivalent to `recv(socket, buf, nbyte, 0)`

  ```
  recv(int socket, void *buffer, size_t length, int flags)
  ```

  - normally, recv() blocks until it has received at least 1 byte
  - returns num bytes received, 0 if connection closed, –1 if error

write

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- returns number of bytes written, –1 on error

Note:

- Number of bytes written may be less than the requested nbyte – ex) filling up a disk
- `write()` may block forever on a “slow” write into pipes, FIFOs, or sockets
- For sockets, `write(socket, buf, nbyte)` is equivalent to `send(socket, buf, nbyte, 0)`

  ```
  send(int socket, const void *buffer, size_t length, int flags)
  ```

  - normally, send() blocks until it sends all bytes requested
  - returns num bytes sent or –1 for error
- If the file was opened with `O_APPEND` flag, the file offset gets set to the end of the file prior to each write
  - setting of the offset and writing happen in an atomic operation

Signals and system calls (revisited)

Recall `shell2.c` from previous lecture:

```
#include "apue.h"
#include <sys/wait.h>

static void sig_int(int);       /* our signal-catching function */

int
main(void)
{
        char    buf[MAXLINE];   /* from apue.h */
        pid_t   pid;
        int     status;

        if (signal(SIGINT, sig_int) == SIG_ERR)
                err_sys("signal error");

        printf("%% ");  /* print prompt (printf requires %% to print %) */
        while (fgets(buf, MAXLINE, stdin) != NULL) {
                if (buf[strlen(buf) - 1] == '\n')
                        buf[strlen(buf) - 1] = 0; /* replace newline with null */

                if ((pid = fork()) < 0) {
                        err_sys("fork error");
                } else if (pid == 0) {      /* child */
                        execlp(buf, buf, (char *)0);
                        err_ret("couldn't execute: %s", buf);
                        exit(127);
                }

                /* parent */
                if ((pid = waitpid(pid, &status, 0)) < 0)
                        err_sys("waitpid error");
                printf("%% ");
        }
        exit(0);
}

void
sig_int(int signo)
{
        printf("interrupt\n%% ");
}
```

Three problems:

1. “Slow” system calls may get interrupted on signals
   - `errno` set to `EINTR`
2. Signals get lost
   - signal handling resets after each signal

3. Signal handler calls non-reentrant functions
    - `malloc()`, `free()`, standard I/O functions

Solutions:

- Don't use `signal()`; use `sigaction()` instead
- From a signal handler, call only *async-signal-safe* functions (see `man 7 signal`)

File sharing

1. Kernel data structures for open files

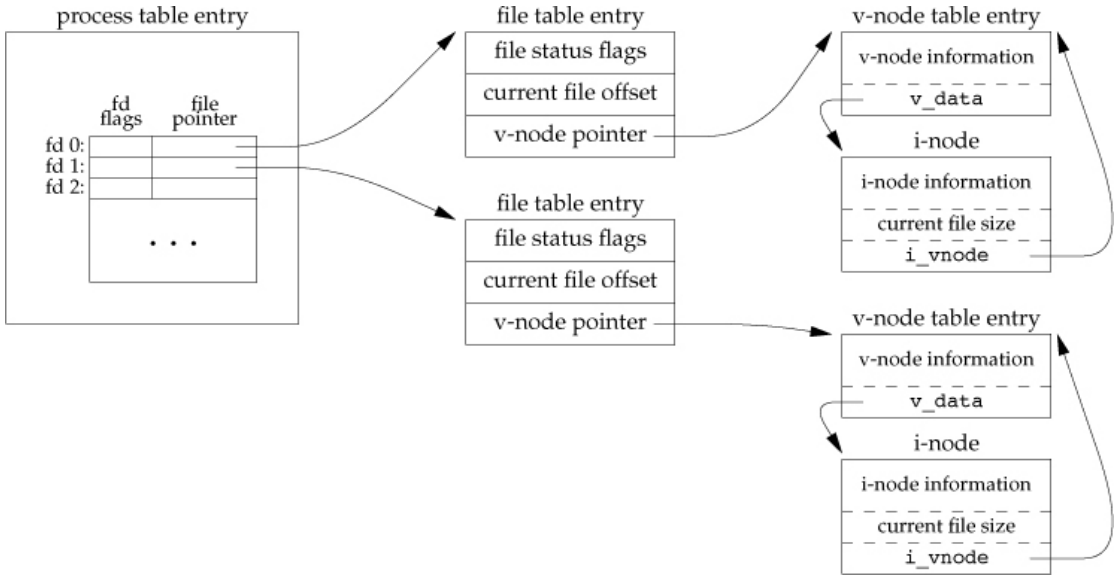

Figure 3.7, APUE

2. Two independent processes with the same file open

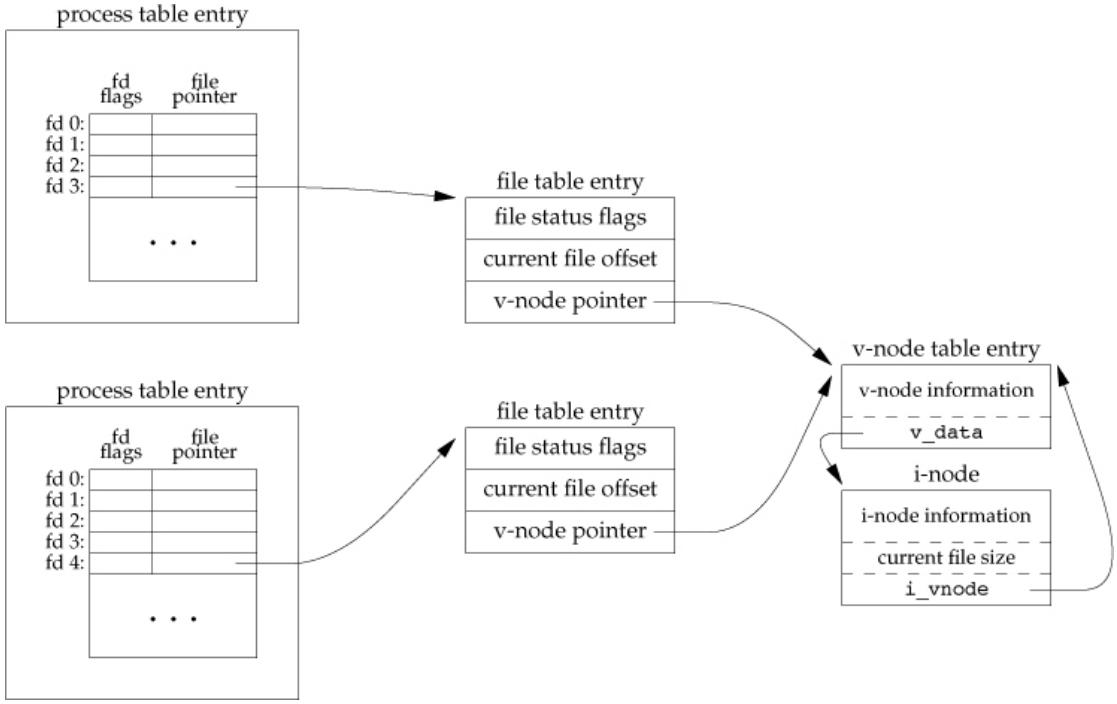

Figure 3.8, APUE

3. Kernel data structures after `dup(1)`

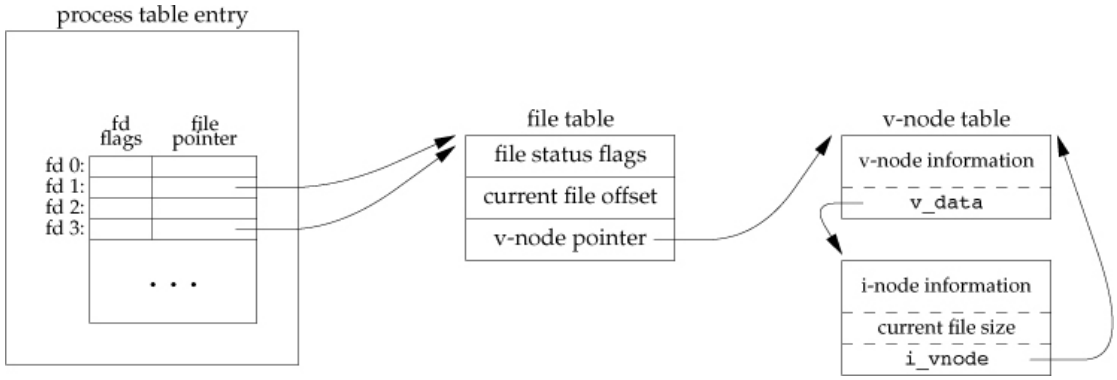

Figure 3.9, APUE

4. Sharing of open files between parent and child after `fork`

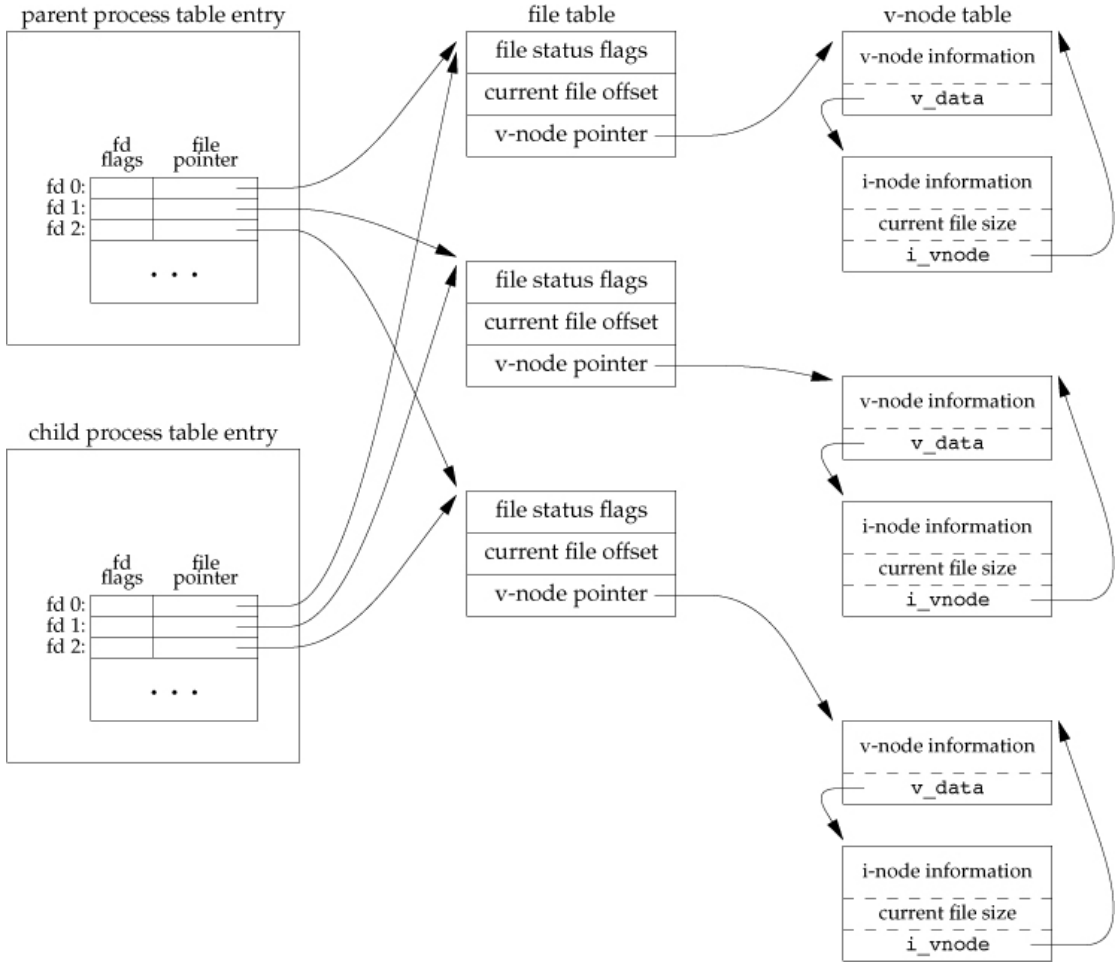

Figure 8.2, APUE

Last updated: 2015–01–30