

Advanced Programming in the UNIX Environment — *File I/O*

Hop Lee
hoplee@bupt.edu.cn

Contents

1	File Descriptors	1
2	open and openat Functions	2
3	creat Function	3
4	close Function	3
5	lseek Function	3
6	read Function	4
7	write Function	4
8	File I/O Efficiency	5
9	File Sharing	5
10	Atomic Operations	6
10.1	Appending to a File	6
10.2	pread and pwrite Functions	7
10.3	Creating a File	7
11	dup and dup2 Functions	7
12	sync, fsync, and fdatasync Functions	8
13	fcntl and ioctl Functions	8
14	Summary	10

1 File Descriptors

- To the kernel all open files are referred to by **file descriptors**. A file descriptor is a non-negative integer.
- When we open an existing file or create a new file, the kernel returns a file descriptor to the process.
- When we want to read or write a file, we identify the file with the file descriptor that was returned by **open** or **creat**.

- The magic numbers 0, 1, and 2 should be replaced in POSIX.1 applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`. These macros are defined in the `unistd.h`.
- File descriptors range from 0 through `OPEN_MAX`.

2 open and openat Functions

- A file is opened or created by calling the `open` function.

```
1 int open(const char *path, int oflag, mode_t mode);
2 int openat(int fd, const char *path, int oflag, mode_t mode);
```

- For these functions, the last argument is used only when a new file is being created.
- The `path` parameter is the name of the file to open or create.
- The parameter `oflag` used to control the operate mode of the file need to be opened. It is formed by OR'ing together one or more of the following constants (from the `fcntl.h` header):

`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 for search only (applies to dirs)

- One and only one of the previous five constants must be specified. The following constants are optional:

`O_APPEND` Append to the end of file on writing

`O_CREAT` Create the file if doesn't exist

`O_EXCL` Generate an error if `O_CREAT` is also specified and the file already exists.

`O_TRUNC` If the file exist, and if the file is successfully opened for either write-only or read-write, truncate its length to 0.

`O_NOCTTY` If the `pathname` refers to a terminal device, do not allocate the device as the controlling terminal for this process.

`O_NONBLOCK` If the `pathname` refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O.

`O_SYNC` Have each `write` wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the `write`.

- The file descriptor returned by `open` is guaranteed to be the lowest numbered unused descriptor.
- The `fd` parameter distinguishes the `openat` function from the `open` function. There are three possibilities:
 1. The `path` parameter specifies an absolute pathname. In this case, the `fd` parameter is ignored and the `openat` function behaves like the `open` function.
 2. The `path` parameter specifies a relative pathname and the `fd` parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The `fd` parameter is obtained by opening the directory where the relative pathname is to be evaluated.
 3. The `path` parameter specifies a relative pathname and the `fd` parameter has the special value `AT_FDCWD`. In this case, the pathname is evaluated starting in the current working directory and the `openat` function behaves like the `open` function.

- The `openat` function is one of a class of functions added to the latest version of POSIX.1 to address two problems. First, it gives threads a way to use relative pathnames to open files in directories other than the current working directory. As we'll see in Chapter 11, all threads in the same process share the same current working directory, so this makes it difficult for multiple threads in the same process to work in different directories at the same time. Second, it provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors.
- The basic idea behind TOCTTOU errors is that a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call. Because the two calls are not atomic, the file can change between the two calls, thereby invalidating the results of the first call, leading to a program error.

3 `creat` Function

- `creat` function can create a new file:

```
1 int creat(const char *pathname, mode_t mode);
```

- This function is equivalent to:

```
1 open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

- So this function is no longer needed.

4 `close` Function

- A open file is closed by `close` function:

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

- Closing a file also releases any record locks that the process may have on the file.
- When a process terminates, all open files are automatically closed by the kernel.

5 `lseek` Function

- Every open file has an associated “current file offset”. This is a nonnegative integer that measures the number of bytes from the beginning of the file.
- Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written.
- By default, this offset is initialized to 0 when a file is opened, unless the `O_APPEND` option is specified.

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

- The interpretation of the offset depends on the value of *whence* parameter.
 - If *whence* is `SEEK_SET`, the file's offset is set to *offset*(positive) bytes from the beginning of the file;
 - If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*(positive or negative).
 - If *whence* is `SEEK_END`, the file's offset is set to the size of the file plus the *offset*(positive or negative).
- Since a successful call to `lseek` returns the new file offset, we can seek zero bytes from the current position to determine the current offset: `off_t currpos = lseek(fd, 0, SEEK_CUR);`

- We can also call `lseek` function to determine if the referenced file is capable of seeking: `lseek` will return -1 and set `errno` to `EPIPE` when the file descriptor refers to a pipe or **FIFO**. See Example (Figure 3.1, `fileio/seek.c`).
- The file's offset can be greater than the file's current size, in which case the next write to the file will extend the file. See Example (Figure 3.2, `fileio/hole.c`).

6 read Function

- Data is read from an open file with `read` function.

```
1 ssize_t read(int fd, void *buf, size_t count);
```

- On success, the number of bytes read is returned; if the end of file is encountered, 0 is returned; on error, -1 is returned, and the `errno` is set appropriately.
- There are several cases in which the number of bytes actually read is less than the amount requested:
 - When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. The next time we call `read`, it will return 0 (EOF).
 - When reading from a terminal device. Normally, up to one line is read at a time. (We'll see how to change this default in Chapter 18.)
 - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
 - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, `read` will return only what is available.
 - When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.
 - When interrupted by a signal and a partial amount of data has already been read. We discuss this further in Section 10.5.
- The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

7 write Function

- Data is written to an open file by `write` function.

```
1 ssize_t write(int fd, const void *buf, size_t count);
```

- On success, the number of bytes written is returned (zero indicates nothing was written).
- On error, -1 is returned, and the `errno` is set appropriately.
- For a regular file, the write starts at the file's current offset. If the `O_APPEND` option was specified in the open function call, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

8 File I/O Efficiency

- In example (Figure 3.5, `fileio/mycat.c`), we redirect `STDIN` to a data file, and redirect `STDOUT` to `/dev/null`. The size of data file is 1,500,000 bytes.
- We show the time results for reading/writing the data file using 18 different buffer sizes in Figure 3.6.
- We'll return to this timing example later in the text. In Section 3.14, we show the effect of **synchronous writes**; in Section 5.8, we compare these **unbuffered I/O** times with the standard I/O library.

9 File Sharing

- UNIX supports the sharing of open files between different processes.
- There are three data structures used by kernel for all I/O, and the relationships among them determinate the effect on process has on another with regard to file sharing.
- There is a process table maintained by kernel. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, with one entry per descriptor. Associated with each file descriptor are:
 - The file descriptor flags
 - A pointer to a file table entry
- The kernel maintains a file table for all open files. Each file table entry contains:
 - The file status
 - The current file offset
 - A pointer to the **v-node** table entry
- Each open file has a v-node structure. It contains information about the file type and pointers to functions that operate on the file. For most files the v-node also contains the **i-node** for the file.

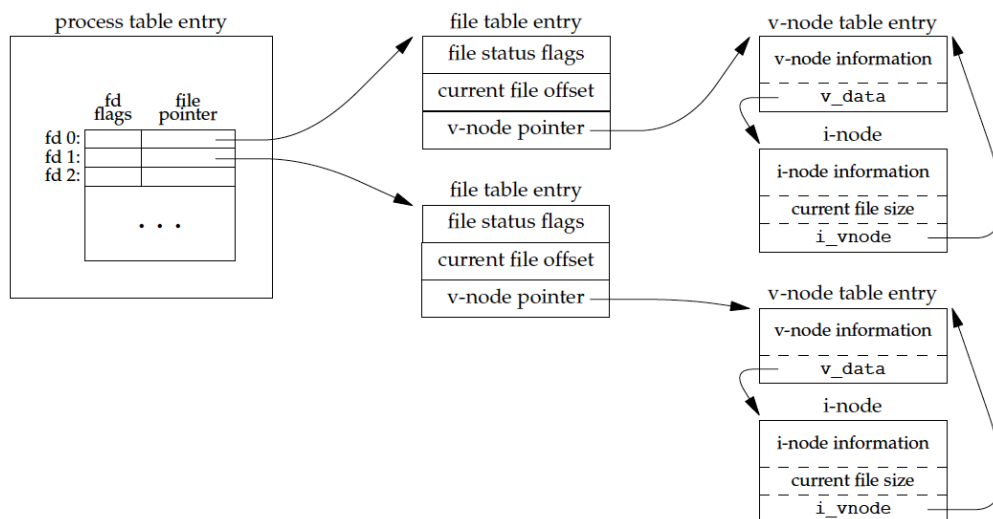


Figure 1: Kernel data structures for open files

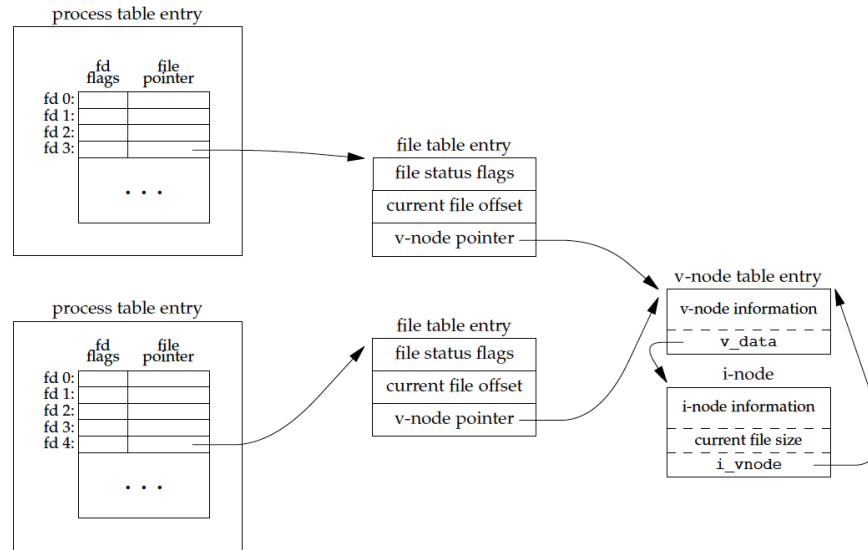


Figure 2: Two independent processes with the same file open

- If two different processes have the same file open. Each process that opens the file gets its own file table entry, but only a single v-node table entry is required for a given file.
- After each write is complete, the current file offset in the file table entry is incremented by the number of bytes written and the current file size in i-node will be changed according to the case.
- If a file is opened with the `O_APPEND` flag, a corresponding flag is set in the file status flags of the file table entry.
- The `lseek` function only modifies the current file offset in the file table entry. No I/O takes place.
- If a file is positioned to its current end of file using `lseek`, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry.

10 Atomic Operations

10.1 Appending to a File

- If two processes want to write to the end of a single file:

```

1 if (lseek(fd, 0L, 2) < 0)
2     err_sys("lseek error");
3 if (write(fd, buf, 100) != 100)
4     err_sys("write error");

```

- The problem here is that our logical operation of “position to the end of file” and “write” required two separate function calls.
- The solution is to have the positioning to the current end of file and write be an **atomic** operation with regard to other processes.
- The UNIX System provides an atomic way to do this operation if we set the `O_APPEND` flag when a file is opened. As we described in the previous section, this causes the kernel to position the file to its current end of file before each `write`. We no longer have to call `lseek` before each write.

- An atomic operation causes the kernel either perform all the steps or perform none.
- Any operation that requires more than one function call can *NOT* be atomic.

10.2 pread and pwrite Functions

- The Single UNIX Specification includes two functions that allow applications to seek and perform I/O atomically: `pread` and `pwrite`.

```
1 #include <unistd.h>
2 ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
3 ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```

- `pread` returns: number of bytes read, 0 if end of file, -1 on error
- `pwrite` returns: number of bytes written if OK, -1 on error
- Calling `pread` is equivalent to calling `lseek` followed by a call to `read`, with the following exceptions.
 - There is no way to interrupt the two operations that occur when we call `pread`.
 - The current file offset is not updated.
- Calling `pwrite` is equivalent to calling `lseek` followed by a call to `write`, with similar exceptions.

10.3 Creating a File

- We saw another example of an atomic operation when we described the `O_CREAT` and `O_EXCL` options for the `open` function. When both of these options are specified, the `open` will fail if the file already exists. We also said that the check for the existence of the file and the creation of the file was performed as an atomic operation. If we didn't have this atomic operation, we might try

```
1 if ((fd = open(path, O_WRONLY)) < 0) {
2     if (errno == ENOENT) {
3         if ((fd = creat(path, mode)) < 0)
4             err_sys("creat error");
5     } else {
6         err_sys("open error");
7     }
8 }
```

- The problem occurs if the file is created by another process between the `open` and the `creat`. If that situation happened, and if that other process writes something to the file, that data is erased when this `creat` is executed. Combining the test for existence and the creation into a single atomic operation avoids this problem.

11 dup and dup2 Functions

- An existing file descriptor is duplicated by these two function calls.

```
1 #include <unistd.h>
2 int dup(int fildes);
3 int dup2(int fildes, int fildes2);
```

- The new file descriptor returned by `dup` is guaranteed to be the lowest numbered available file descriptor.

- With `dup2` we specify the value of the new descriptor with `filedes2` parameter. If `filedes2` is already open, close it first. If `filedes` equals `filedes2`, then return `filedes2` and nothing happened.
- They are *atomic* operations.

12 sync, fsync, and fdatasync Functions

- Traditional UNIX implementations have a buffer cache in the kernel through which most disk I/O passes. When we write data to a file the data is normally copied by the kernel into one of its buffers and queued for I/O at some later time. This is called **delay write**.
- To ensure consistency of the actual filesystem on disk with the contents of the buffer cache, the `sync`, `fsync` and `fdatasync` functions are provided.

```
1 #include <unistd.h>
2 void sync (void);
3 int fsync (int fd);
4 int fdatasync (int fd);
```

- The function `sync` just queues all the modified block buffers for writing and returns, it does not wait for the actual I/O to take place.
- The function `sync` is normally called every 30 seconds from a system daemon (`updated`).
- The function `fsync` refers only to a single file, and waits for the I/O to complete before returning.
- The `fdatasync` function is similar to `fsync`, but it affects only the data portions of a file. With `fsync`, the file's attributes are also updated synchronously.

13 fcntl and ioctl Functions

- The `fcntl` function can change the properties of a open file.

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 int fcntl(int fd, int cmd, .../* int arg */);
```

- Its return value depends on `cmd`, 0 if success, -1 on error.
- The `fcntl` function is used for five different purposes:
 - duplicate an existing descriptor
 - get/set file descriptor flags
 - get/set file status flags
 - get/set asynchronous I/O ownership
 - get/set record locks
- `F_DUPFD`: Duplicate the file descriptor `filedes`;
- `F_GETFD`: Return the file descriptor flags;
- `F_SETFD`: Set the file descriptor flags;

- `F_GETFL`: Return the file status flags;
 - The access mode flags are not separate bits that can be tested. Therefore we must first use `O_ACCMODE` mask to obtain the access mode bits and then test them.
- `F_SETFL`: Set the file status flags. The only flags that can be changed are `O_APPEND`, `O_NONBLOCK`, `O_SYNC`, and `O_ASYNC`.
- `F_GETOWN`: Get the process ID or process group ID currently receiving the `SIGIO` and `SIGURG` signals.
- `F_SETOWN`: Set the process ID or process group ID to receive the `SIGIO` and `SIGURG` signals.
- Example (Figure 3.11, `fileio/fileflags.c`).

```

1  ./fileflags 0 < /dev/tty
2  ./fileflags 1 > temp.foo
3  ./fileflags 2 2>> temp.foo
4  ./fileflags 5 5<> temp.foo

```

- In all examples listed above, arguments are 0, 1, 2, and 5. The third command redirect `STDERR` append to a regular file `temp.foo`, so we can get its output from the terminal.
- “5<> temp.foo” was interpreted by shell. It means open file `temp.foo` for read and write with a file descriptor 5. There must *NO* space between 5 and <>.
- Example (Figure 3.12, `lib/setfl.c`).
- An example of `O_SYNC` flag. When we turn on the synchronous write flag, it will cause each write to wait for the data to be written to disk before returning. Normally in UNIX, a write only queues the data for writing, and the actual I/O operation is taking place sometime later.
- The `BUFFSIZE=8192`: with a 1.5M file, Figure 3.5’s load is: 0.014 0.000 0.014.
- If we write to a regular file instead of `/dev/null`, the load is: 0.036 0.004 0.031.
- If we add `O_SYNC` flag and write to a regular file, the load is: 0.063 0.004 0.045.
- The `ioctl` function has always been the catchall for I/O operations. Anything that couldn’t be expressed using one of the other functions in this chapter usually ended up being specified with an `ioctl`.

```

1  #include <unistd.h>    /* System V */
2  #include <sys/ioctl.h> /* BSD and Linux */
3  #include <stropts.h>    /* XSI STREAMS */
4  int ioctl(int filedes, int request, ...);

```

- The first parameter is an open file descriptor.
- The second parameter is a device-depend request code, and the third is an un-typed pointer to memory.
- Check “`man ioctl_list`” for a nearly complete list of `ioctl` system call’s usage.

14 Summary

- This chapter has described the basic I/O functions provided by the UNIX System. These are often called the unbuffered I/O functions because each `read` or `write` invokes a system call into the kernel. Using only `read` and `write`, we looked at the effect of various I/O sizes on the amount of time required to read a file. We also looked at several ways to flush written data to disk and their effect on application performance.
- Atomic operations were introduced when multiple processes append to the same file and when multiple processes create the same file. We also looked at the data structures used by the kernel to share information about open files. We'll return to these data structures later in the lecture.
- We also described the `ioctl` and `fcntl` functions. We return to both of these functions later in the lecture. In Chapter 14, we'll use `fcntl` for record locking. In Chapter 18 and Chapter 19, we'll use `ioctl` when we deal with terminal devices.

The End of Chapter 3.