WIKIPEDIA

File descriptor

In Unix and related computer operating systems, a **file descriptor** (**FD**, less frequently **fildes**) is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket. File descriptors form part of the POSIX application programming interface. A file descriptor is a non-negative integer, generally represented in the C programming language as the type `int` (negative values being reserved to indicate "no value" or an error condition).

Each Unix process (except perhaps a daemon) should expect to have three standard POSIX file descriptors, corresponding to the three standard streams:

Integer value	Name	<unistd.h> symbolic constant[1]	<stdio.h> file stream[2]
0	Standard input	`STDIN_FILENO`	`stdin`
1	Standard output	`STDOUT_FILENO`	`stdout`
2	Standard error	`STDERR_FILENO`	`stderr`

Contents

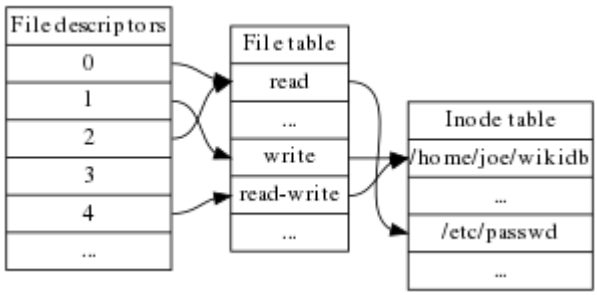
Overview

In the traditional implementation of Unix, file descriptors index into a per-process **file descriptor table** maintained by the kernel, that in turn indexes into a system-wide table of files opened by all processes, called the **file table**. This table records the *mode* with which the file (or other resource) has been opened: for reading, writing, appending, and possibly other modes. It also indexes into a third table called the inode table that describes the actual underlying files.[3] To perform input or output, the process passes the file descriptor to the kernel through a system call, and the kernel will access the file on behalf of the process. The process does not have direct access to the file or inode tables.

On Linux, the set of file descriptors open in a process can be accessed under the path `/proc/PID/fd/`, where PID is the process identifier.

In Unix-like systems, file descriptors can refer to any Unix file type named in a file system. As well as regular files, this includes directories, block and character devices (also called "special files"), Unix domain sockets, and named pipes. File descriptors can also refer to other objects that do not normally exist in the file system, such as anonymous pipes and network sockets.

The FILE data structure in the C standard I/O library usually includes a low level file descriptor for the object in question on Unix-like systems. The overall data structure provides additional abstraction and is instead known as a *file handle*.



File descriptors for a single process, file table and inode table. Note that multiple file descriptors can refer to the same file table entry (e.g., as a result of the dup system call[3]:104 and that multiple file table entries can in turn refer to the same inode (if it has been opened multiple times; the table is still simplified because it represents inodes by file names, even though an inode can have multiple names). File descriptor 3 does not refer to anything in the file table, signifying that it has been closed.

Operations on file descriptors

The following lists typical operations on file descriptors on modern Unix-like systems. Most of these functions are declared in the `<unistd.h>` header, but some are in the `<fcntl.h>` header instead.

Creating file descriptors

- `open()`
- `creat()`
- `socket()`
- `accept()`
- `socketpair()`
- `pipe()`
- `opendir()`

Deriving file descriptors

- `dirfd()`
- `fileno()`

Operations on a single file descriptor

- `read()`, `write()`
- `readv()`, `writev()`
- `pread()`, `pwrite()`
- `recv()`, `send()`
- `recvmsg()`, `sendmsg()` (including allowing sending FDs)
- `sendfile()`
- `lseek()`
- `fstat()`
- `fchmod()`
- `fchown()`
- `fdopen()`
- `ftruncate()`
- `fsync()`

- `fdatasync()`
- `fstatvfs()`
- `dprintf()`

Operations on multiple file descriptors

- `select()`, `pselect()`
- `poll()`
- `epoll()` (for Linux)
- `kqueue()` (for BSD-based systems).

Operations on the file descriptor table

The `fcntl()` function is used to perform various operations on a file descriptor, depending on the command argument passed to it. There are commands to get and set attributes associated with a file descriptor, including `F_GETFD, F_SETFD, F_GETFL` and `F_SETFL`.

- `close()`
- `closefrom()` (BSD and Solaris only; deletes all file descriptors greater than or equal to specified number)
- `dup()` (duplicates an existing file descriptor guaranteeing to be the lowest number available file descriptor)
- `dup2()` (the new file descriptor will have the value passed as an argument)
- `fcntl (F_DUPFD)`

Operations that modify process state

- `fchdir()` (sets the process's current working directory based on a directory file descriptor)
- `mmap()` (maps ranges of a file into the process's address space)

File locking

- `flock()`
- `fcntl() (F_GETLK, F_SETLK)` and `F_SETLKW`
- `lockf()`

Sockets

- `connect()`
- `bind()`
- `listen()`
- `accept()` (creates a new file descriptor for an incoming connection)
- `getsockname()`
- `getpeername()`
- `getsockopt()`
- `setsockopt()`
- `shutdown()` (shuts down one or both halves of a full duplex connection)

Miscellaneous

- `ioctl()` (a large collection of miscellaneous operations on a single file descriptor, often associated with a device)

Upcoming operations

A series of new operations on file descriptors has been added to many modern Unix-like systems, as well as numerous C libraries, to be standardized in a future version of POSIX.[4] The `at` suffix signifies that the function takes an additional first argument supplying a file descriptor from which relative paths are resolved, the forms lacking the `at` suffix thus becoming equivalent to passing a file descriptor corresponding to the current working directory. The purpose of these new operations is to defend against a certain class of TOCTTOU attacks.

- `openat()`
- `faccessat()`
- `fchmodat()`
- `fchownat()`
- `fstatat()`
- `futimesat()`
- `linkat()`
- `mkdirat()`
- `mknodat()`
- `readlinkat()`
- `renameat()`
- `symlinkat()`
- `unlinkat()`
- `mkfifoat()`
- `fdopendir()`

File descriptors as capabilities

Unix file descriptors behave in many ways as capabilities. They can be passed between processes across Unix domain sockets using the `sendmsg()` system call. Note, however, that what is actually passed is a reference to an "open file description" that has mutable state (the file offset, and the file status and access flags). This complicates the secure use of file descriptors as capabilities, since when programs share access to the same open file description, they can interfere with each other's use of it by changing its offset or whether it is blocking or non-blocking, for example.[5][6] In operating systems that are specifically designed as capability systems, there is very rarely any mutable state associated with a capability itself.

A Unix process' file descriptor table is an example of a C-list.

See also

- fuser (Unix)
- lsof

References

1. The Open Group. "The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008, 2016 Edition" (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/unistd.h.html). Retrieved 2017-09-21.
2. The Open Group. "The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008, 2016 Edition" (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdio.h.html). <stdio.h>. Retrieved 2017-09-21.
3. Bach, Maurice J. (1986). *The Design of the UNIX Operating System* (8 ed.). Prentice-Hall. pp. 92–96. ISBN 9780132017992.
4. *Extended API Set, Part 2* (https://www2.opengroup.org/ogsys/catalog/c063). The Open Group. October 2006. ISBN 1931624674.
5. Brinkmann, Marcus (2009-02-04). "Building a bridge: library API's and file descriptors?" (http://www.eros-os.org/pipermail/cap-talk/2009-February/012137.html). *cap-talk*. Retrieved 2017-09-21.
6. de Boyne Pollard, Jonathan (2007). "Don't set shared file descriptors to non-blocking I/O mode" (http://jdebp.eu/FGA/dont-set-shared-file-descriptors-to-non-blocking-mode.html). Retrieved 2017-09-21.