dup(2) — Linux manual page

DUP(2) Linux Programmer's Manual DUP(2)

NAME top

dup, dup2, dup3 - duplicate a file descriptor

SYNOPSIS top

```
#include <unistd.h>

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

#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <fcntl.h>              /* Definition of O_* constants */
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

DESCRIPTION top

The **dup**() system call allocates a new file descriptor that refers to the same open file description as the descriptor *oldfd*. (For an explanation of open file descriptions, see open(2).) The new file descriptor number is guaranteed to be the lowest-numbered file descriptor that was unused in the calling process.

After a successful return, the old and new file descriptors may be used interchangeably. Since the two file descriptors refer to the same open file description, they share file offset and file status flags; for example, if the file offset is modified by using lseek(2) on one of the file descriptors, the offset is also changed for the other file descriptor.

The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (**FD_CLOEXEC**; see fcntl(2)) for the duplicate descriptor is off.

dup2()

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it

uses the file descriptor number specified in *newfd*. In other words, the file descriptor *newfd* is adjusted so that it now refers to the same open file description as *oldfd*.

If the file descriptor *newfd* was previously open, it is closed before being reused; the close is performed silently (i.e., any errors during the close are not reported by **dup2**()).

The steps of closing and reusing the file descriptor *newfd* are performed *atomically*. This is important, because trying to implement equivalent functionality using close(2) and **dup**() would be subject to race conditions, whereby *newfd* might be reused between the two steps. Such reuse could happen because the main program is interrupted by a signal handler that allocates a file descriptor, or because a parallel thread allocates a file descriptor.

Note the following points:

* If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.

* If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2**() does nothing, and returns *newfd*.

dup3()

dup3() is the same as **dup2**(), except that:

* The caller can force the close-on-exec flag to be set for the new file descriptor by specifying **O_CLOEXEC** in *flags*. See the description of the same flag in open(2) for reasons why this may be useful.

* If *oldfd* equals *newfd*, then **dup3**() fails with the error **EINVAL**.

RETURN VALUE top

On success, these system calls return the new file descriptor. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS top

EBADF *oldfd* isn't an open file descriptor.

EBADF *newfd* is out of the allowed range for file descriptors (see the discussion of **RLIMIT_NOFILE** in getrlimit(2)).

EBUSY (Linux only) This may be returned by **dup2**() or **dup3**() during a race condition with open(2) and **dup**().

EINTR The **dup2**() or **dup3**() call was interrupted by a signal; see signal(7).

EINVAL (**dup3**()) *flags* contain an invalid value.

EINVAL (**dup3**()) *oldfd* was equal to *newfd*.

EMFILE The per-process limit on the number of open file descriptors has been reached (see the discussion of **RLIMIT_NOFILE** in getrlimit(2)).

VERSIONS top

dup3() was added to Linux in version 2.6.27; glibc support is available starting with version 2.9.

CONFORMING TO top

dup(), **dup2**(): POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

dup3() is Linux-specific.

NOTES top

The error returned by **dup2**() is different from that returned by **fcntl(..., F_DUPFD, ...)** when *newfd* is out of range. On some systems, **dup2**() also sometimes returns **EINVAL** like **F_DUPFD**.

If *newfd* was open, any errors that would have been reported at close(2) time are lost. If this is of concern, then—unless the program is single-threaded and does not allocate file descriptors in signal handlers—the correct approach is *not* to close *newfd* before calling **dup2**(), because of the race condition described above. Instead, code something like the following could be used:

```
/* Obtain a duplicate of 'newfd' that can subsequently
   be used to check for close() errors; an EBADF error
   means that 'newfd' was not open. */

tmpfd = dup(newfd);
if (tmpfd == -1 && errno != EBADF) {
    /* Handle unexpected dup() error. */
}

/* Atomically duplicate 'oldfd' on 'newfd'. */

if (dup2(oldfd, newfd) == -1) {
    /* Handle dup2() error. */
}

/* Now check for close() errors on the file originally
   referred to by 'newfd'. */

if (tmpfd != -1) {
    if (close(tmpfd) == -1) {
        /* Handle errors from close. */
```

```
        }
    }
```

SEE ALSO top

close(2), fcntl(2), open(2), pidfd_getfd(2)

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Linux **2021-03-22** **DUP(2)**

Pages that refer to this page: bpf(2), fcntl(2), flock(2), getrlimit(2), kcmp(2), lseek(2), open(2), pidfd_getfd(2), syscalls(2), fileno(3), getdtablesize(3), posix_spawn(3), epoll(7), pipe(7), signal-safety(7), unix(7)
