

# **Advanced Programming in the UNIX Environment**

**Week 02, Segment 2: open(2) and close(3)**

**Department of Computer Science  
Stevens Institute of Technology**

**Jan Schaumann**

jschauma@stevens.edu

<https://stevens.netmeister.org/631/>

## Standard I/O

---

Basic File I/O: almost all UNIX file I/O can be performed using these five functions:

- `open(2)`
- `close(2)`
- `read(2)`
- `write(2)`
- `lseek(2)`

## **creat(2)**

---

```
#include <fcntl.h>
```

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

Returns: file descriptor if OK, -1 on error

`creat(2)` returns a file handle in write-only mode. To get a read-write file handle:

```
1 if ((fd = creat(path, mode) < 0 ) {  
2     /* error */  
3 }  
4 (void)close(fd);  
5 if ((fd = open(path, O_RDWR) < 0) {  
6     /* error */  
7 }  
8 /* do stuff with 'fd' ... */
```

## **creat(2)**

---

```
#include <fcntl.h>
```

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

Returns: file descriptor if OK, -1 on error

This interface is made obsolete by `open(2)`.

`creat()` is the same as:

```
open(path, O_CREAT | O_TRUNC | O_WRONLY, mode);
```

## open(2)

```
#include <fcntl.h>
```

```
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

*oflag* must be one (and only one) of:

- O\_RDONLY - open for reading only
- O\_WRONLY - open for writing only
- O\_RDWR - open for reading and writing

and may be OR'd with any of these:

- O\_APPEND – append on each write
- O\_CREAT – create the file if it doesn't exist; requires *mode* argument
- O\_EXCL – error if O\_CREAT and file already exists. (atomic)
- O\_TRUNC – truncate size to 0
- O\_NONBLOCK – do not block on open or for data to become available
- O\_SYNC – wait for physical I/O to complete

## open(2)

---

```
#include <fcntl.h>
```

```
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

Additional *oflags* may be supported on some platforms:

- O\_DIRECTORY – if path resolves to a non-directory file, fail and set errno to ENOTDIR
- O\_DSYNC – wait for physical I/O for data, except file attributes
- O\_EXEC – open file for execute only, fail if it is a directory
- O\_NOFOLLOW – do not follow symlinks
- O\_PATH – obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)
- O\_RSYNC – block read operations on any pending writes
- O\_SEARCH – open for search only, fail if it is a regular file
- ...

## **openat(2)**

---

```
#include <fcntl.h>
```

```
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

```
int openat(int dirfd, const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

`openat(2)` is used to handle relative pathnames from different working directories in an atomic fashion.

Here, *pathname* is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory.



## open(2)

```
#include <fcntl.h>
```

```
int open(const char *pathname, int oflag, ... /* mode_t mode */);
```

Returns: file descriptor if OK, -1 on error

open(2) may fail for a surprising number of reasons. Some of the more common ones include:

- **EEXIST**: **O\_CREAT** | **O\_EXCL** was specified, but the file exists
- **EMFILE**: process has already reached max number of open file descriptors
- **ENOENT**: file does not exist
- **EPERM**: lack of permissions
- ...

```
1 fd = open(path, O_RDWR);
2 /* do stuff with fd */
```

```
1 if ((fd = open(path, O_RDWR) < 0) {
2     /* error */
3 }
4 /* do stuff with fd */
```



## **close(2)**

---

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, -1 on error

- closing a filedescriptor releases any record locks on that file (more on that in future lectures)
- file descriptors not explicitly closed are closed by the kernel when the process terminates.
- to avoid leaking file descriptors, always close(2) them within the same scope

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
```

```
int
main() {
    /* imagine a few dozen lines of code here */
}
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
~
```

```
"f.c" 8L, 120C written
```

```
6,8
```

```
71
```

```
75%
```

## **close(2)**

---

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, -1 on error

- closing a filedescriptor releases any record locks on that file (more on that in future lectures)
- file descriptors not explicitly closed are closed by the kernel when the process terminates.
- to avoid leaking file descriptors, always `close(2)` them within the same scope

```
1 (void)close(fd);
2 /* you can't do stuff with fd here either way */
```



Trying to create './newfile' with O\_RDONLY | O\_CREAT...

'./newfile' created. File descriptor is: 4

-rw----- 1 jschauma users 0 Sep 4 19:51 newfile

Checking if './newfile' exists...

-rw----- 1 jschauma users 0 Sep 4 19:51 ./newfile

Trying to create './newfile' with O\_RDONLY | O\_CREAT | O\_EXCL...

Unable to create './newfile': File exists

Closing failed: Bad file descriptor

Trying to open './openex.c' with O\_RDONLY...

'./openex.c' opened. File descriptor is: 5

'./openex.c' closed again

Trying to open (non-existent) './nosuchfile' with O\_RDONLY...

Unable to open './nosuchfile': No such file or directory

Copied 'openex.c' to 'newfile'.

-rw----- 1 jschauma users 3192 Sep 4 19:52 newfile

Trying to open './newfile' with O\_RDONLY | O\_TRUNC...

'./newfile' opened. File descriptor is: 5

'./newfile' truncated -- see 'ls -l newfile'

-rw----- 1 jschauma users 0 Sep 4 19:52 newfile

apue\$

## In our next segment...

---

- `read(2)`
- `write(2)`
- `lseek(2)`

Can you go backwards on a pipe?

What happens when you try to write data way beyond the end of a file?

How efficient is our `simple-cat.c` program?