# Syscalls: Files 2: Basic I/O Syscalls

1. Library vs. Syscall I/O
2. **Basic I/O Syscalls** (part a)
   - **open**
   - **close**
3. Basic I/O Syscalls (part b)
4. File Offset
5. Filesystem Syscalls

# The Basic I/O System Calls

The set of basic I/O-related syscalls is small:

- **open()** – open a file, possibly creating it
- **close()** – close an open file
- **read()** – read from an open file
- **write()** – write to an open file

While much of the time these four syscalls will be all that are needed, there are a few additional I/O syscalls.

The additional calls can read/write at specific positions in files, deal with particular special files like sockets, etc.

# open() System Call

`open()` is the syscall to *open* a file, making it possible to read from and/or write to the file.

It can potentially also cause the file to be *created*.

It has two possible syntax variations:

```
int open(const char *pathname, int flags)
int open(const char *pathname, int flags, mode_t mode)
```

- `pathname` is the file path as a C string
- `flags` specifies the file **access mode** and possibly other options, such as file creation
- `mode` specifies the file *permissions*, and should be provided if the file might be created by the call (`O_CREAT` flag)
- returns a *file descriptor* handle for the open file, else -1 on *error*

# open() System Call (contd.)

`flags` *must include one* of the following three **access mode** specs, which determine what operations can be done to the file:

- **O_RDONLY** – allow only reading
- **O_WRONLY** – allow only writing
- **O_RDWR** – allow both reading and writing

Optional flags may be included by *bitwise-or'ing* them with the access mode spec using the C bitwise-or operator ("|"):

```
O_WRONLY | O_CREAT | O_EXCL
```

# open() System Call (contd.)

The most useful/common `open()` flags are:

- **O_APPEND** – open file in **append mode**, meaning any writen data is appended to the end of the file automatically
- **O_CREAT** – create the file if it does not already exist
- **O_EXCL** – in conjunction with `O_CREAT`, causes an error if the file already exists
- **O_TRUNC** – if the file exists (and the access mode includes writing), file will be **truncated** (to length zero) when opened
- **O_CLOEXEC** – close the open file if an exec call is made
- **O_NONBLOCK** – open file in **nonblocking mode**, meaning a `read()` will immediately return even if there is no data to be read, etc

# open() System Call (contd.)

If the `O_CREAT` flag is supplied so that the file may be *created*, the `mode` argument *should be supplied* (otherwise the file mode will end up being random, which can cause access problems later).

`mode` is often supplied as an *octal* value such as would be used with the `chmod` command:

```
open("test",O_WRONLY|O_CREAT,0764)
```

[the leading `0` in `0764` tells C to interpret the number as *octal*]

There are also a set of *symbolic constants* that can be bitwise or'd together to specify the desired mode/permissions:

```
open("test",O_WRONLY|O_CREAT,S_IRWXU|S_IRGRP|S_IWGRP|S_IROTH)
```

Use "`man 2 open`" for a list of the flag/mode symbols.

# open() System Call (contd.)

`open()` always returns the *lowest-numbered* file descriptor not currently active for the process.

`open()` returns -1 in case of an error, plus `errno` will be set ($> 0$).

There are a number of reasons why `open()` can encounter an *error* and fail.

`errno`'s value can be checked to determine the particular error.

Among the most common `open()` errors are:

- **EACCES** – requested access denied due to permissions (user does not have permission for requested access mode)
- **EEXIST** – file exists and `O_CREAT` and `O_EXCL` were used
- **ENOENT** – file does not exist and `O_CREAT` was not used

# open() System Call (contd.)

Example of opening an existing file for reading:

```
int infd;
if ((infd = open("test.text",O_RDONLY)) == -1) {
  perror("Opening file test.text failed");
  exit(EXIT_FAILURE); }
...
```

Notes:

- The file descriptor return will virtually always need to be *captured* since this is the handle to access the opened file.
- The return should *always* be checked to see if an error occurred because errors can definitely occur with `open()`.
- The common C idiom is to do both of the above together inside the `if` condition.

# open() System Call (contd.)

Example of creating a *new* file and opening it for writing:

```
int outfd;
if ((outfd = open("newfile.text",O_WRONLY|O_CREAT|O_EXCL,0640)) == -1) {
  perror("Creating file newfile.text failed");
  exit(EXIT_FAILURE); }
...
```

Example of opening a file for reading and writing, *creating* the file if it does not exist or *truncating* it if it does:

```
int outfd;
if ((outfd = open("maybenewfile.text",O_RDWR|O_CREAT|O_TRUNC,0640)) == -1) {
  perror("Creating file maybenewfile.text failed");
  exit(EXIT_FAILURE); }
...
```

# close() System Call

`close()` is the syscall that closes an open file:

```
int close(int fd)
```

- `fd` is the open file's file descriptor
- returns 0 (zero) on success, else -1 on error

# close() System Call (contd.)

Example of closing an open file (descriptor):

```
int fd;
if ((fd = open(...)) == -1)
...
if (close(fd) == -1) {
  perror("Error closing file");
  exit(EXIT_FAILURE); }
```

Note that it is quite common to ignore the return from `close()` (not do any error checking on it).

Unless the file descriptor argument is wrong, an error is unlikely and nothing can be done about it anyway.

# close() System Call (contd.)

When a process terminates, Linux/UNIX systems automatically close all of the process' open files.

This means there is little point in making a bunch of `close()` calls just prior to exiting.

However, in long running processes such as servers, it can be critical to close open files as soon as the program is finished with them:

- the maximum number of per process file descriptors is a *resource* that can be limited by the sysadmin
- open files take up memory in the process' *file descriptor table*
- open files take up memory in the global *open files table*