CS631 - Advanced Programming in the UNIX Environment

File I/O, File Sharing

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/

Recall `simple-cat.c` from last week...

```
int main(int argc, char **argv) {
        int n;
        char buf[BUFFSIZE];

        while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0) {
                if (write(STDOUT_FILENO, buf, n) != n) {
                        fprintf(stderr, "write error\n");
                        exit(1);
                }
        }
        if (n < 0) {
                fprintf(stderr, "read error\n");
                exit(1);
        }

        return(0);
}
```

Warm-up exercise

Write a program that:

- prints the value of `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`
- prints the value of the file descriptors referenced via the `stdin`, `stdout`, `stderr` streams
- `open(2)`'s a file, then prints the value of that file descriptor
- `fopen(3)`'s a file, then prints the value of the file descriptor referenced via that stream

What results do you expect?

Let's look at the file descriptors.

`fds.c`

File Descriptors

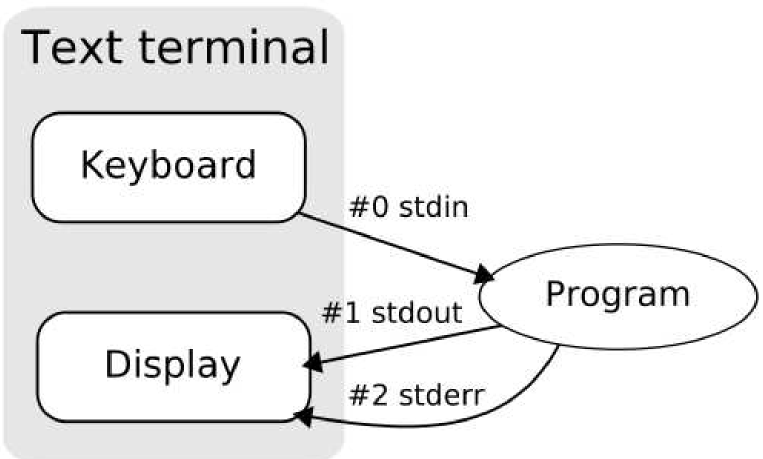
- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.

File Descriptors

- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.

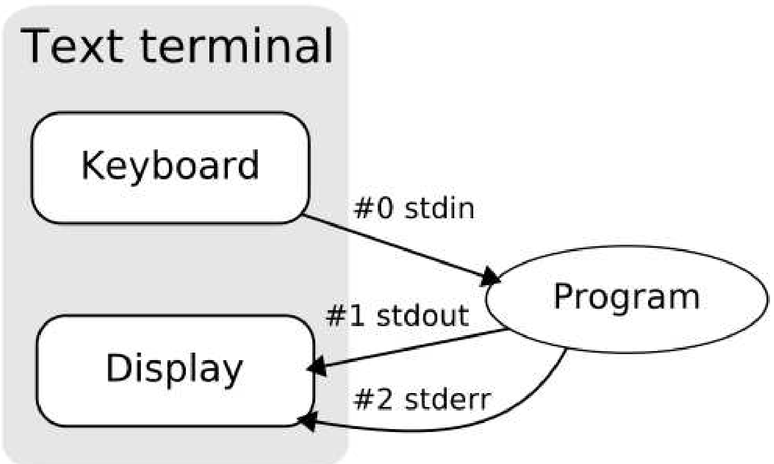
File Descriptors

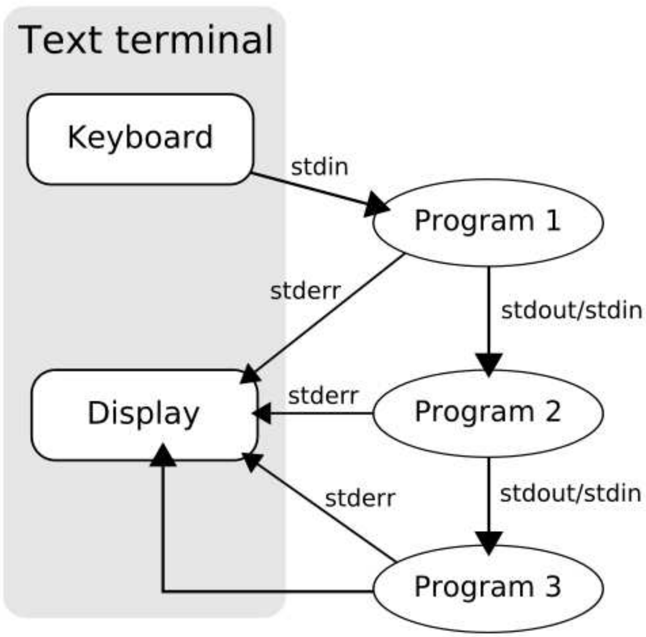
- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.



File Descriptors

- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.





File Descriptors

- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.
- Relying on "magic numbers" is Bad™. Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.

File Descriptors

- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.
- Relying on "magic numbers" is Bad™. Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.
- All resources are finite. So are file descriptors. So... how many can you have?

File Descriptors

- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.
- Relying on "magic numbers" is Bad™. Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.
- All resources are finite. So are file descriptors. So... how many can you have?

`openmax.c`

See also: `https://en.wikipedia.org/wiki/File_descriptor`

Standard I/O

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

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

Processes may want to share recources. This requires us to look at:

- atomicity of these operations
- file sharing
- manipulation of file descriptors

creat(2)

```
#include <fcntl.h>

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

Returns: file descriptor if OK, -1 on error

https://is.gd/x4KPa2

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)`.

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 to end of file for each write
- `O_CREAT` – Create the file if it doesn't exist. Requires *mode* argument
- `O_EXCL` – Generate error if `O_CREAT` and file already exists. (atomic)
- `O_TRUNC` – If file exists and successfully open in `O_WRONLY` or `O_RDWR`, make length = 0
- `O_NOCTTY` – If pathname refers to a terminal device, do not allocate the device as a controlling terminal
- `O_NONBLOCK` – If pathname refers to a FIFO, block special, or char special, set nonblocking mode (open and I/O)
- `O_SYNC` – Each write waits for physical I/O to complete

`open(2)` variants

```
#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

On some platforms additional *oflag*s may be supported:

- `O_EXEC` – Open for execute only
- `O_SEARCH` – Open for search only (applies to directories)
- `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_RSYNC` – Block read operations on any pending writes.
- `O_PATH` – Obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)

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

`openat(2)`

POSIX (`https://is.gd/3hZ4EZ`) says:

The purpose of the `openat()` *function is to enable opening files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to* `open()`*, resulting in unspecified behavior. By opening a file descriptor for the target directory and using the openat() function it can be guaranteed that the opened file is located relative to the desired directory. Some implementations use the* `openat()` *function for other purposes as well.*

Think of *specific* examples how this defeats TOCTOU problems; write a Proof-of-Concept program to illustrate.

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

`open(2)` and `close(2)`

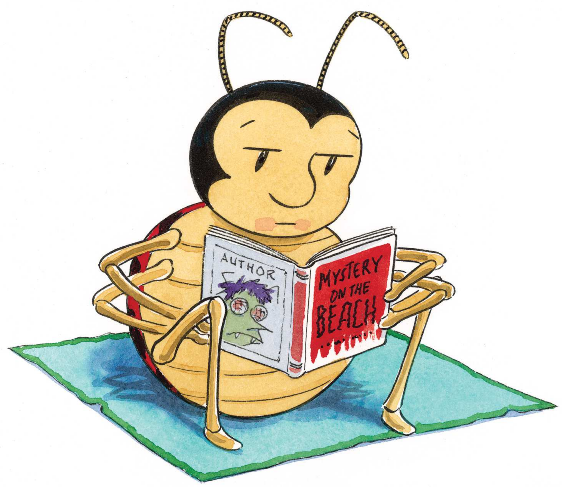
`openex.c`

read(2)

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes );
```

Returns: number of bytes read, 0 if end of file, -1 on error



https://is.gd/qI5r8E

read(2)

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes );
```

Returns: number of bytes read, 0 if end of file, -1 on error

There can be several cases where `read` returns less than the number of bytes requested. For example:

- EOF reached before requested number of bytes have been read
- Reading from a terminal device, one ”line” read at a time
- Reading from a network, buffering can cause delays in arrival of data
- Record-oriented devices (magtape) may return data one record at a time
- Interruption by a signal

`read` begins reading at the current offset, and increments the offset by the number of bytes actually read.

write(2)

```
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes );
```

Returns: number of bytes written if OK, -1 on error

- `write` returns `nbytes` or an error has occurred
- for regular files, `write` begins writing at the current offset (unless `O_APPEND` has been specified, in which case the offset is first set to the end of the file)
- after the write, the offset is adjusted by the number of bytes actually written

`write(2)`

Some manual pages note:

If the real user is not the super-user, then `write()` *clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-id file owned by the super-user.*

Think of *specific* examples for this behaviour. Write a program that attempts to exploit a scenario where `write(2)` does *not* clear the setuid bit, then verify that your evil plan will be foiled.

`read(2)` and `write(2)`

`rwex.c`

lseek(2)

```
#include <sys/types.h>
#include <fcntl.h>

off_t lseek(int filedes, off_t offset, int whence );
```

Returns: new file offset if OK, -1 on error

http://is.gd/3fp5Vx

lseek(2)

```
#include <sys/types.h>
#include <fcntl.h>

off_t lseek(int filedes, off_t offset, int whence );
```

Returns: new file offset if OK, -1 on error

The value of whence determines how offset is used:

- SEEK_SET bytes from the beginning of the file
- SEEK_CUR bytes from the current file position
- SEEK_END bytes from the end of the file

"Weird" things you can do using `lseek(2)`:

- seek to a negative offset
- seek 0 bytes from the current position
- seek past the end of the file

lseek(2)

```
$ cc -Wall lseek.c
$ ./a.out < lseek.c
seek OK
$ cat lseek.c | ./a.out
cannot seek
$ mkfifo fifo
$ ./a.out <fifo
```

lseek(2)

```
$ cc -Wall hole.c
$ ./a.out
$ ls -l file.hole
-rw-------  1 jschauma  wheel  10240020 Sep 18 17:20 file.hole
$ hexdump -c file.hole
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000010  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
09c4000  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0   A   B   C   D   E   F
09c4010   G   H   I   J
09c4014
$ cat file.hole > file.nohole
$ ls -ls file.*
   96 -rw-------  1 jschauma  wheel  10240020 Sep 18 17:20 file.hole
20064 -rw-r--r--  1 jschauma  wheel  10240020 Sep 18 17:21 file.nohole
```

https://en.wikipedia.org/wiki/Sparse_file (not on e.g. HFS+)

I/O Efficiency

Reviewing the program `simple-cat.c` from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately

I/O Efficiency

Reviewing the program `simple-cat.c` from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately
- works for “text” and “binary” files since there is no such distinction in the UNIX kernel

I/O Efficiency

Reviewing the program `simple-cat.c` from the last class:

- assumes that *stdin* and *stdout* have been set up appropriately
- works for "text" and "binary" files since there is no such distinction in the UNIX kernel
- how do we know the optimal `BUFFSIZE`?

I/O Efficiency

```
$ make tmpfiles

$ for n in $(seq 10); do
        dd if=/dev/urandom of=tmp/file$n count=204800
done

$ i=1; for n in 1048576 32768 16384 4096 512 256 128 64 1 ; do
        cc -Wall -DBUFFSIZE=$n simple-cat.c;
        i=$(( $i + 1 ));
        time ./a.out <tmp/file$i >tmp/file$i.copy;
done

$ make catio

$ stat -f "%k" tmp/file1 # stat -c "%o" tmp/file1
```

Note: results vary depending on OS/filesystem.

So far, so good...

What questions do you have?

Hooray!

5 Minute Break

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (See: Stevens, pp 75 ff)

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (See: Stevens, pp 75 ff)

- each process table entry has a table of file descriptors, which contain
  - the file descriptor flags (e.g. `FD_CLOEXEC`, see `fcntl(2)`)
  - a pointer to a file table entry

File Sharing

Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (See: Stevens, pp 75 ff)

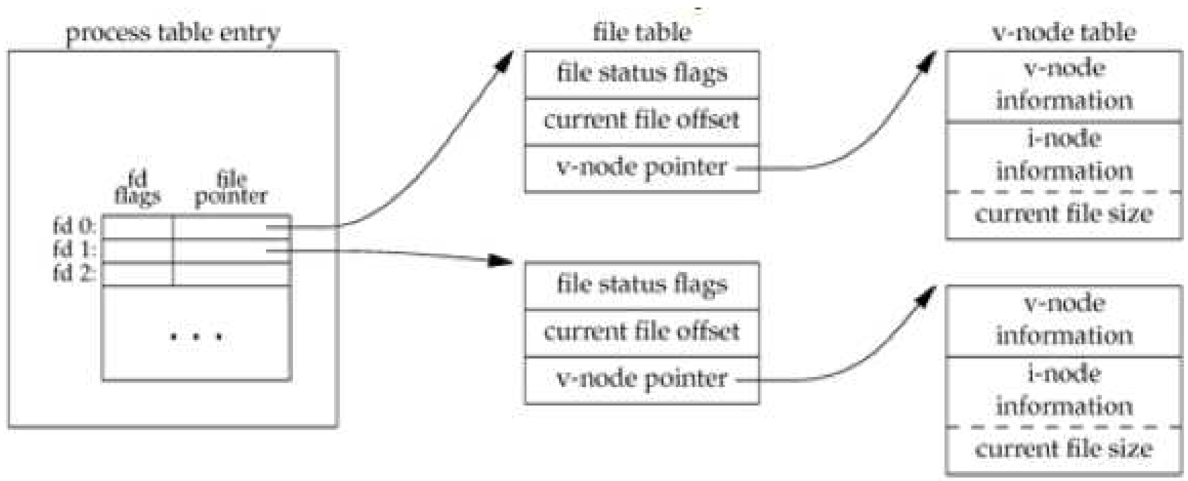
- each process table entry has a table of file descriptors, which contain
  - the file descriptor flags (e.g. `FD_CLOEXEC`, see `fcntl(2)`)
  - a pointer to a file table entry
- the kernel maintains a file table; each entry contains
  - file status flags (`O_APPEND`, `O_SYNC`, `O_RDONLY`, etc.)
  - current offset
  - pointer to a vnode table entry

File Sharing

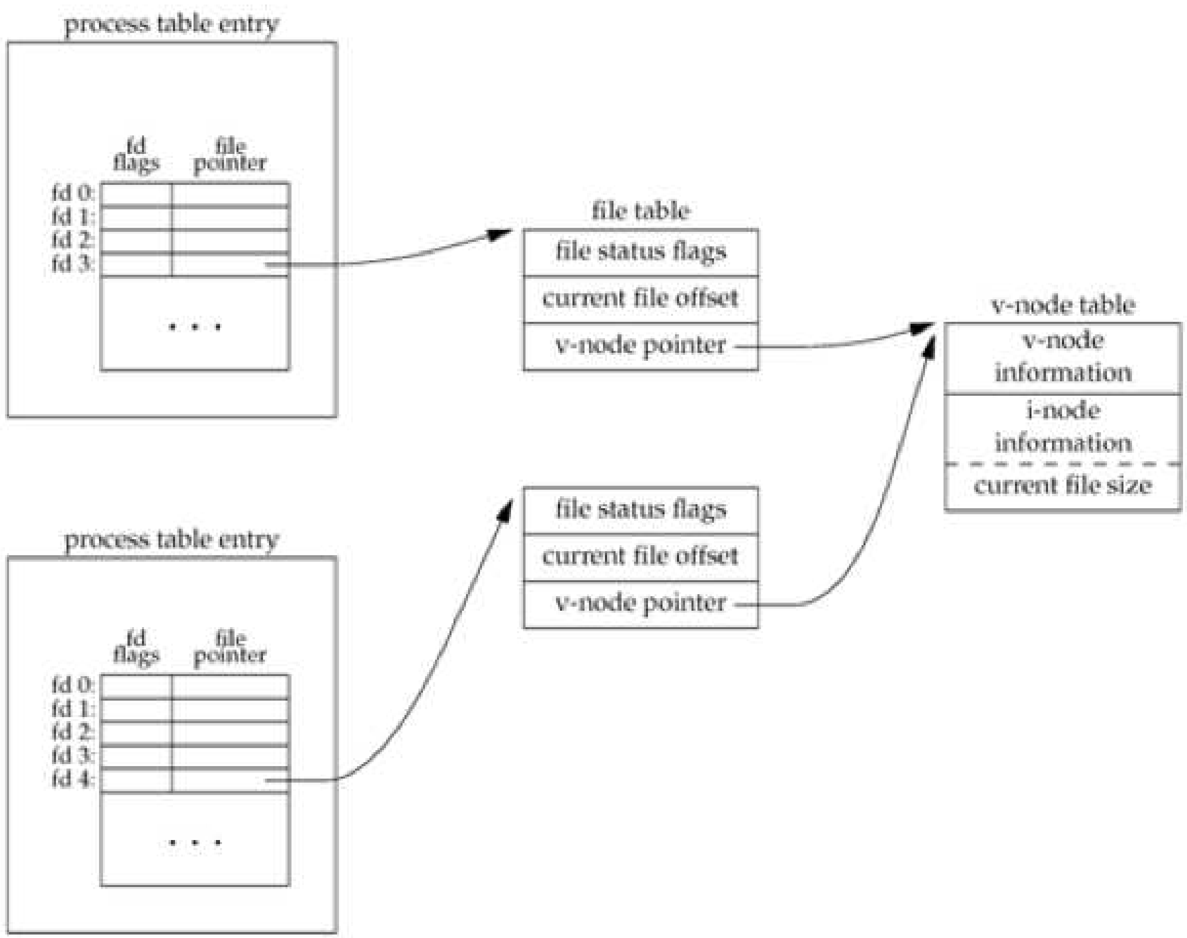
Since UNIX is a multi-user/multi-tasking system, it is conceivable (and useful) if more than one process can act on a single file simultaneously. In order to understand how this is accomplished, we need to examine some kernel data structures which relate to files. (See: Stevens, pp 75 ff)

- each process table entry has a table of file descriptors, which contain
  - the file descriptor flags (e.g. `FD_CLOEXEC`, see `fcntl(2)`)
  - a pointer to a file table entry
- the kernel maintains a file table; each entry contains
  - file status flags (`O_APPEND`, `O_SYNC`, `O_RDONLY`, etc.)
  - current offset
  - pointer to a vnode table entry
- a vnode structure contains
  - vnode information
  - inode information (such as current file size)

File Sharing



File Sharing



File Sharing

Knowing this, here's what happens with each of the calls we discussed earlier:

- after each `write` completes, the current file offset in the file table entry is incremented. (If current_file_offset $>$ current_file_size, change current file size in i-node table entry.)
- If file was opened `O_APPEND` set corresponding flag in file status flags in file table. For each `write`, current file offset is first set to current file size from the i-node entry.
- `lseek` simply adjusts current file offset in file table entry
- to `lseek` to the end of a file, just copy current file size into current file offset.

Atomic Operations

In order to ensure consistency across multiple writes, we require *atomicity* in some operations. An operation is atomic if either *all* of the steps are performed or *none* of the steps are performed.

Suppose UNIX didn't have `O_APPEND` (early versions didn't). To append, you'd have to do this:

```
if (lseek(fd, 0L, 2) < 0) {        /* position to EOF */
    fprintf(stderr, "lseek error\n");
    exit(1);
}

if (write(fd, buff, 100) != 100) { /* ...and write */
    fprintf(stderr, "write error\n");
    exit(1);
}
```

What if another process was doing the same thing to the same file? Recall `rwex.c`.

pread(2) and pwrite(2)

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, void *buf, size_t count, off_t offset);
```

Both return number of bytes read/written, -1 on error

Atomic read/write at offset without invoking `lseek(2)`.
Current offset is *not* updated.

`dup(2)` and `dup2(2)`

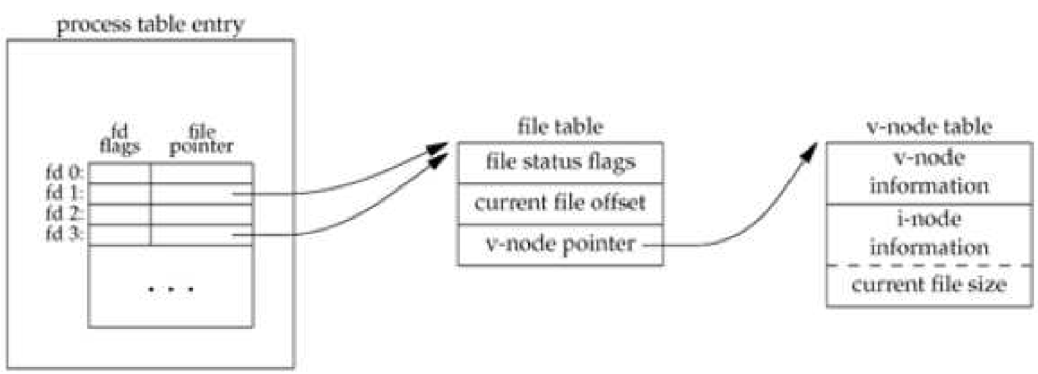
```
#include <unistd.h>

int dup(int oldd);
int dup2(int oldd, int newd);
```

Both return new file descriptor if OK, -1 on error

An existing file descriptor can be duplicated with `dup(2)` or duplicated to a particular file descriptor value with `dup2(2)`. As with `open(2)`, `dup(2)` returns the lowest numbered unused file descriptor.

Note the difference in scope of the file *descriptor* flags and the file *status* flags compared to distinct processes.



fcntl(2)

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int filedes, int cmd, ...  /* int arg */);
```

Returns: depend on *cmd* if OK, -1 on error

`fcntl(2)` is on of those "catch-all" functions with a myriad of purposes. Here, they all relate to changing properties of an already open file. It can:

cmd	effect	return value
`F_DUPFD`	duplicate *filedes* (`FD_CLOEXEC` file descriptor flag is cleared	new filedes
`F_GETFD`	get the file descriptor flags for *filedes*	descriptor flags
`F_SETFD`	set the file descriptor flags to the value of the third argument	not -1
`F_GETFL`	get the file status flags	status flags
`F_SETFL`	set the file status flags	not -1

...as well as several other functions.

fcntl(2)

```
$ cc -Wall sync-cat.c -o scat
$ sed -e 's/\(.*O_SYNC.*\)/\/\/\1/' sync-cat.c > async-cat.c
$ cc -Wall async-cat.c -o ascat
$ time ./scat <file >out

$ time ./ascat <file >out

$ make sync async
```

Note: results will differ depending on the filesystem (-options).

ioctl(2)

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* 4.3+BSD */

int ioctl(int filedes, int request, ...);
```

Returns: -1 on error, something else if OK

Another catch-all function, this one is designed to handle device specifics that can't be specified via any of the previous function calls. For example, terminal I/O, magtape access, socket I/O, etc.

Mentioned here mostly for completeness's sake.

/dev/fd

```
$ bash
$ ls -l /dev/stdin /dev/stdout /dev/stderr
lr-xr-xr-x  1 root  wheel  0 Sep  7 13:56 /dev/stderr -> fd/2
lr-xr-xr-x  1 root  wheel  0 Sep  7 13:56 /dev/stdin -> fd/0
lr-xr-xr-x  1 root  wheel  0 Sep  7 13:56 /dev/stdout -> fd/1
$ ls -l /dev/fd/
total 0
crw--w----   1 jschaumann  tty     16,   4 Sep  8 21:48 0
crw--w----   1 jschaumann  tty     16,   4 Sep  8 21:48 1
crw--w----   1 jschaumann  tty     16,   4 Sep  8 21:48 2
drw-r--r--  93 jschaumann  staff      3162 Sep  8 21:40 3
dr--r--r--   1 root        wheel         0 Sep  7 13:56 4
$ echo first >file1
$ echo third >file2
$ echo second | cat file1 /dev/fd/0 file2
```

Note: https://marc.info/?l=ast-users&m=120978595414990&w=2

Homework

- Reading:
  - manual pages for the functions covered
  - Stevens Chap. 3, 4
- Thinking:
  - Stevens # 3.4
  - Stevens # 3.5 (bourne shell syntax "$> \&$")
  - Use `openat(2)` to protect against TOCTOU issues.
  - Confirm that `write(2)` clearing the setuid bit foils your evil attempts to root the system.
  - Determine the optimal file I/O size on different systems via the benchmark example.
- Coding: `https://stevens.netmeister.org/631/f18-hw1.html`