CptS 360 (System Programming)
Unit 5: Low-Level I/O

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- *All* file and device (and some network) input/output ultimately uses these routines.
- Most of them map one-to-one to kernel system calls.
- This is the functionality all I/O drivers have to provide.

References

- Stevens & Rago Ch. 3

Introduction

- Recall system/utility library interface diagram.
- These all work with *file descriptors* (aka "`fd`s").
  - Q: What is the C type used for a file descriptor?
  - Q: What are the typical values of file descriptors?
- All processes start with 3 open file descriptors: 0, 1, and 2.
  - This is a UNIX tradition.
  - Q: What are they and what do they correspond to?

open(2) and *creat(2)*

To get access to a file[1]:

- *open(2)*
  - Opens a file for reading and/or writing.
  - What it does depends on the `flags` argument (below).
  - It can create a new file.
  - It returns a file descriptor (or ?) to that file.
- *creat(2)*

  Creates a file. It's equivalent to calling `open()` with `flags` set to "`O_CREAT|O_WRONLY|O_TRUNC`".

[1] From now on, we'll use the term "file" to refer to anything that has a name in the filesystem: regular files, directories, devices, sockets, FIFOs, etc.

Common *open(2)* Flags

- `O_RDONLY`, `O_WRONLY`, or `O_RDWR`
  read-only, write-only, read/write (choose one)
- `O_APPEND`
  append (always write to the end of the file)[2]
- `O_CREAT`
  create file if it doesn't exist
- `O_EXCL`
  exclusive access (with `O_CREAT`, make sure file is created by caller)
- `O_TRUNC`
  truncate to length 0

If necessary, combine these with the "|" (the bitwise "or" operator).

[2] This is especially useful for log files.

Less-Used *open(2)* Flags

- `O_NOCTTY`
  if opening a device, don't make it the control tty (later)
- `O_NONBLOCK` or `O_NDELAY`
  nonblocking open/access – caller will never wait
- `O_SYNC`
  sync (force physical disk I/O) on every call
  (generally not a good idea – see why?)

The `mode` Argument to *open(2)* and *creat(2)*

- Remember "ugo" for mode flags.
- Primitive (early UNIX) security notion divides the world into three classes:
  - user
  - members of the user's group
  - "other"
- `mode` is only used when `O_CREAT` is one of the `flags`.
  - It's affected by *umask(2)* (below).

`mode` is a mask (in octal[3])

			also set in
00400	user read	`S_IRUSR`	`S_IRWXU`
00200	user write	`S_IWUSR`	`S_IRWXU`
00100	user execute	`S_IXUSR`	`S_IRWXU`
00040	group read	`S_IRGRP`	`S_IRWXG`
00020	group write	`S_IWGRP`	`S_IRWXG`
00010	group execute	`S_IXGRP`	`S_IRWXG`
00004	others read	`S_IROTH`	`S_IRWXO`
00002	others write	`S_IWOTH`	`S_IRWXO`
00001	others execute	`S_IXOTH`	`S_IRWXO`

(See the `demos/d`*n*`_octal` demo.)

[3] base-8. This is one of the few times this base is still used in computing.

umask(2)

umask(2) sets the calling process's "file creation mask".

- The `mask` argument becomes a process attribute.
- Whenever set, it affects all later *open(2)* (and *creat(2)*) calls.
- The umask bitmask says: "Turn these mode bits off, even if the `mode` argument says to turn them on."
- The mask of the created file is `(mode & ~umask)` where `mode` is the argument to *open(2)* (or *creat(2)*).

read(2) and *write(2)*

These are the primary low-level I/O functions.

- *read(2)*
  reads from a file.
- *write(2)*
  writes to a file.
- Q: How do you know if the operation fails?

Note: In both cases, the returned value is usually equal to `count`, but might not be even if the operation was successful (see the man pages).

I/O and Program Efficiency

Q: What's are the advantages of these routines over those in the Standard I/O library (TBD)? Vice Versa?

- These are *system calls*: Each one asks for kernel service.
- This is "raw" I/O: There's no buffering in user memory (unlike Standard I/O).
- Performance could go either way.
- You have the option to do "no wait" I/O and treat these calls like "requests".
- You can exclusively lock regions of the file.

We'll study this issue in an upcoming lab.

lseek(2)

lseek(2) makes the next *read(2)* or *write(2)* operation start at any byte location in the file. This is what we call this "random access" (cf. "RAM").

- Q: What major application area depends on the ability to skip around from any location to any other in a (possibly) huge file?
- Not every "file" permits this. (Q: Name one that doesn't.)
- Q: Can you `lseek()`
  - standard input (`fd` == 0)?
  - standard output (`fd` == 1)?
- `whence` should be one of:
  - `SEEK_SET`
  - `SEEK_CUR`
  - `SEEK_END`
- Take a look at *lseek64(3)*.

close(2)

close(2) tells the OS you no longer want to access `fd`.

- Calling *close(2)* is optional, but couth. (Exception: You usually don't close `fd`'s 0, 1, or 2.)
- When your process exits, the OS will close any open files you don't close.

File Sharing

- Two processes on the same system can *open(2)* and share the same file.
- This is okay for one writer, any number of readers.
- This is not okay for > 1 writer.

Simultaneous Operations

There are some tricky issues with multiple processes. Suppose two processes are running at the same time.

- If they open the same file for writing and do an *lseek(2)* to EOF, followed by *write(2)*.
  - Both writes succeed, but the later overwrites the earlier. (a classic *race condition*)
  - Solution: Both processes call *open(2)* with the `O_APPEND` flag. (Both appends work in chronological order.)
- If they both try to create the same file if the file doesn't already exist...
  - Both calls succeed, but the earlier process's file will vanish when it's closed(!)
  - Solution: Both processes call *open(2)* with the `O_CREAT` and `O_EXCL` flags. (The earlier process succeeds. The later one fails.)

dup(2) and *dup2(2)*

dup(2) and *dup2(2)* duplicate one file descriptor to another. Write to either of them go to the same place.

- `dup2(foo, bar);`
  closes "bar" before dupping it
- Handy: Lets you direct two already-open files (e.g. standard output and standard error) to the same place.
- This is how

  ```
  $ myprog 2>&1 | less
  ```

  paginates both `myprog`'s standard output and standard error.
- This is useful for interprocess communcation (later).

fcntl(2)

fcntl(2) performs miscellaneous "file control" functions:

- duplication (redundant with *dup(2)*)
- retrieve the `fd`'s *file descriptor flags*
- modify (some of) or get the `fd`'s `mode` and `flags`
- exclusively lock and unlock byte ranges of the file (advisory)
- manage signals affecting `fd`
- set the "close on exec" flag on `fd` (later)
- lots of other stuff

ioctl(2)

ioctl(2) is a "catch-all" like *fcntl(2)*, but is more device-related to control stuff like:

- disk labels
- device control (e.g., CD/DVD eject)
- mag tape control (e.g., setting tape density – bits per inch)
- socket (later) I/O
- terminal I/O
  esp. character-at-a-time I/O (e.g. for *stty(1)*)

Summary: What's in a Device Driver?

Supporting a new device in the kernel means writing a *device driver*, code for the kernel which implements these functions (some may be trivial):

- `open()`
  connect to the device
- `read()`
  read data from the device, if it is capable of reading
- `write()`
  write data to the device, if it is capable of writing
- `lseek()`
  reposition the device, if it is capable of repositioning
- `close()`
  disconnect from the device
- `fcntl()`
  perform any miscellaneous file-like operations on the device
- `ioctl()`
  perform any miscellaneous device-like operations on the device