Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

I/O
(Part 3)

Devin J. Pohly <djpohly@cse.psu.edu>

Two styles

- High-level, library-managed I/O
  - Uses `FILE *`
  - Requires `<stdio.h>`
  - Implemented by the library on top of low-level I/O
- Low-level, kernel-managed I/O
  - Uses integer file descriptor
  - Requires several headers:
    - `<sys/types.h>`
    - `<sys/stat.h>`
    - `<fcntl.h>`
    - `<unistd.h>`

High- vs. low-level

- Key differences between high-level and low-level I/O
  - High-level I/O provides you with buffering I/O at the library level, and it may or may not turn out to be faster than the low-level functions.
    - Depends on application, workload, and lower layers
    - Kernel also does buffering for both, unless you request unbuffered
  - High-level I/O does line-ending translation if the file is not opened in binary mode, which can be helpful (for text!) if your program is ported to a non-Unix environment.
  - High-level I/O gives you the ability to parse formatted text using `fscanf` and similar stdio functions.

- General rule: use high-level I/O for ASCII/text processing, and low-level I/O for binary data.

The file abstraction

- Remember our abstract definition of a "file" for Unix-like systems?

The file abstraction

- Remember our abstract definition of a "file" for Unix-like systems?
  - All of the low-level I/O functions operate on this file abstraction
    - Sockets
    - Pipes
    - POSIX message queues
  - So you will use a file descriptor for all of these

Reading from a file

- To read a block of data from a file, use `read`:

```
ssize_t read(int fd, void *buf, size_t count);
```

  - `fd`: file descriptor of the file to be read
  - `buf`: buffer (array of bytes) into which to read the data from the file
  - `count`: how many bytes to try to read
- Return value: the number of bytes that were actually read
  - Be sure to check the result: -1 indicates an error, and short reads (`rv < count`) are possible.
    - Function sets the `errno` value on error, so you can use `perror` to print the corresponding error message
  - 0 means EOF or equivalent (e.g., connection closed).
  - You are responsible for supplying a large enough buffer!

Writing to a file

- To write a block of data to a file, use `write`:

```
ssize_t write(int fd, const void *buf, size_t count);
```

  - `fd`: file descriptor of the file to be written to
  - `buf`: buffer (array of bytes) containing the data to write to the file
  - `count`: how many bytes to try to write
- Return value: the number of bytes that were actually written
  - Again, check the result: -1 indicates an error, and short writes are possible.
    - Function sets the `errno` value on error, so you can use `perror` to print the corresponding error message
  - 0 is less common than with `read` but can happen in similar circumstances.

Seeking

- You can change the current position for reads and writes (known as *seeking*):

```
off_t lseek(int fd, off_t offset, int whence)
```

  - `fd`: file descriptor of the file to seek
  - `ofs`: how many bytes to seek...
  - `whence`: ... and from where in the file
    - `SEEK_SET`: beginning of the file
    - `SEEK_CUR`: current offset
    - `SEEK_END`: end of the file
- Return value: the new offset
  - Or -1 for error, sets `errno`
  - How could you use this to query the current offset?

Read/write with offset

- Functions for random-access reads and writes:

```
pread(fd, buf, count, ofs)
pwrite(fd, buf, count, ofs)
```

- Same arguments as `read`/`write`, but with offset parameter
  - Offset from the beginning of the file
  - Does not change the current offset permanently, only for this read or write
    - I.e., not equivalent to `lseek` + `read`/`write`

Demo time

```c
int show_open(void) {
    // Set up variables
    char *filename = "/tmp/open.dat";
    uint32_t vals[1000] = { [0...999] = 0xff }, vals2[1000];
    int fd, flags;
    mode_t mode;

    // Open and create the file
    flags = O_WRONLY|O_CREAT|O_EXCL;  // New file, don't overwrite
    mode = S_IRUSR|S_IWUSR|S_IRGRP;  // User can read/write, group read
    fd = open(filename, flags, mode);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    // Now write the array to the file
    if (write(fd, vals, sizeof(vals)) != sizeof(vals)) {
        perror("write");
        return 1;
    }

    close(fd);
    fd = -1;

    // continued on next slide...
```

Demo time

```
    // Open the file for reading
    flags = O_RDONLY;
    fd = open(filename, flags, 0);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    // Now read the array from the file
    if (read(fd, vals2, sizeof(vals2)) != sizeof(vals2)) {
        perror("read");
        return 1;
    }

    close(fd);
    fd = -1;

    return 0;
}
```

```
$ ./show_open
$ od -t x1 /tmp/open.dat
0000000 ff 00 00 00 ff 00 00 00 ff 00 00 00 ff 00 00 00
*
0000400
```

Bridging the gap

- You can get a file descriptor from a `FILE *`, and (sometimes) vice versa:

```
int fileno(FILE *stream)
FILE *fdopen(int fd, const char *mode)
```

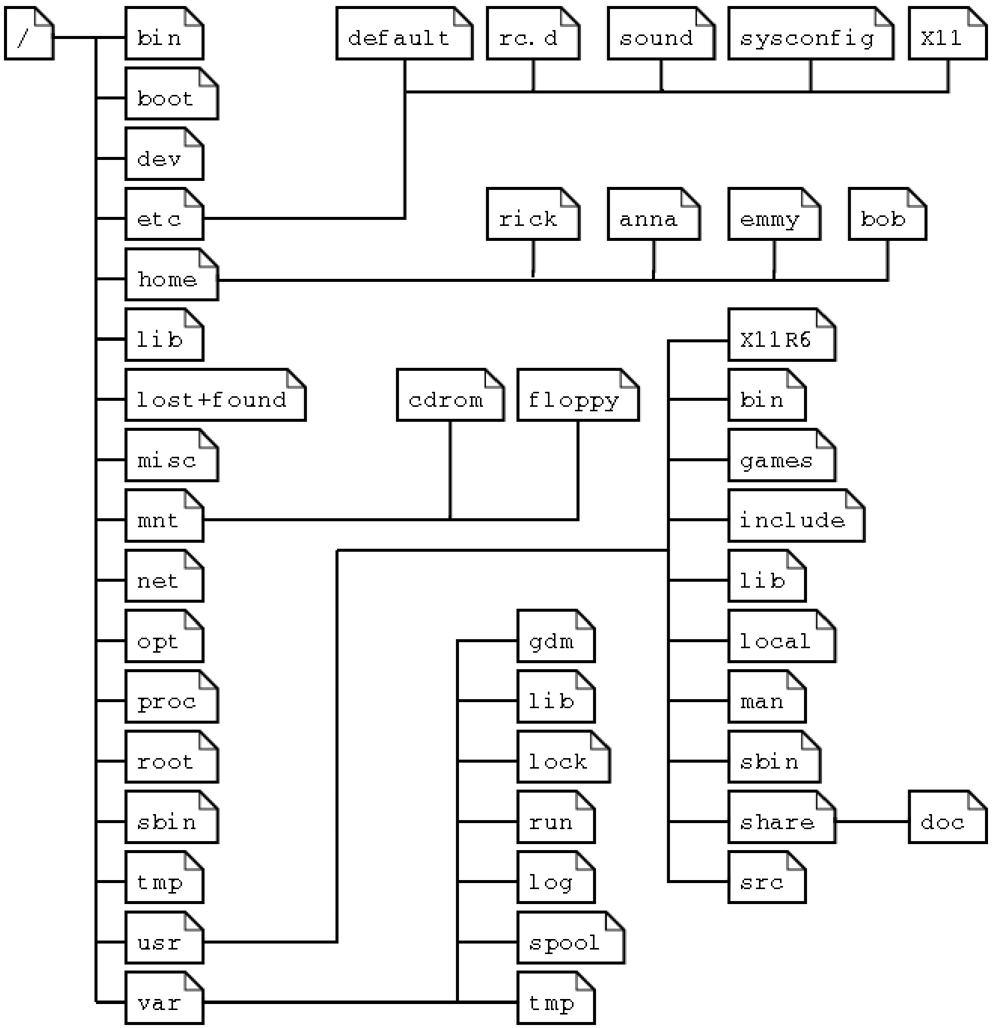
  - The `mode` parameter is the same as in the `fopen` function
- Don't mix and match I/O styles – stick to one or the other!
  - Buffer/flushing issues, and be sure you only close the file once...

Some stream equivalents

- There are `fread` and `fwrite` functions which work similarly to `read` and `write`.
- The `fseek` function works like `lseek`, except with a `FILE *` instead of file descriptor and a different return value.
  - Can't get current offset this way; use `ftell` instead.

Filesystem functions

- POSIX also provides functions for manipulating the Unix filesystem
  - I.e., files themselves, not their contents
  - Equivalents for most of the core command-line utilities



Basic file operations

- We already learned to create files using `open` and `O_CREAT`.
- To rename an existing file:

```
int rename(const char *old, const char *new)
```

- To delete an existing file:

```
int unlink(const char *path)
```

- Both return 0 for success and nonzero for error, and set `errno` accordingly
- Wait a second... "unlink"??
  - Any idea why the `unlink` function is called that?

Links

- Unix systems allow you to create “links” to files
  - *Symbolic links*, or *symlinks*, just point to a file’s path
    - Like Windows shortcuts
    - Underlying file can move or disappear (“broken link”)
    - Symlink doesn’t count as a reference
  - *Hard links* are multiple paths that refer to the same underlying file data
    - Feature of Unix filesystems
    - One file with several equivalent paths
    - Hard link counts as a reference to the file – only deleted when none left

Link commands

- You can create and remove links from the command line
- Create links:

  `ln SRC DEST`

  - Hard links by default
  - Give `-s` flag for symlinks
- Remove links:
  - This is actually what the `rm` command does!

On the C side

```
int link(const char *src, const char *dest)
int symlink(const char *src, const char *dest)
int rename(const char *src, const char *dest)
```

- All three take an existing file as the source and manipulate the links in the filesystem
- Return value for all three: 0 for success, nonzero for error
  - All three also set `errno` on error
- What exactly does `rename` do?
  - Try to think in terms of filesystem links
  - How does this work for moving files between directories?