Un Known · Follow
Last edited by Un Known on Mar 4, 2024
Contributed by

3/1/24 Discussion Notes

Final `cat.c`:

```c
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

#define BUF_SZ 4096

int main(int argc, char* argv[]) {
    int opt;
    int verbose = 0;
    while ((opt = getopt(argc, argv, "ve:")) != -1) {
        switch (opt) {
            case 'v':
                verbose = 1;
                break;
            case 'e':
                printf("%s\n", optarg);
                break;
            default:
                fprintf(stderr, "Unknown argument %c\n", opt);
                return 1;
        }
    }
    for (int i = optind; i < argc; i++) {
        char* filepath = argv[i];
        if (verbose) {
            fprintf(stderr, "Opening file %s...\n", filepath);
        }
        int fd = open(filepath, O_RDONLY);
        if (fd < 0) {
            perror("open");
            return 1;
        }
        char buf[BUF_SZ];
        while (1) {
            ssize_t n = read(fd, buf, BUF_SZ);
            if (n < 0) {
                perror("read");
                return 1;
            }
            if (n == 0) {
                break;
            }
            ssize_t written = 0;
            while (written < n) {
                ssize_t w = write(STDOUT_FILENO, buf + written, n - written);
                if (w < 0) {
                    perror("write");
                    return 1;
                }
                written += w;
            }
        }
        close(fd);
    }
}
```

Compile with `gcc cat.c -o cat`. Flags of interest:

- `-Wall` enables most warnings (I recommend always using this)
- `-g` enables debugging information
- `-fsanitize=address` enables AddressSanitizer which catches many types of undefined behavior (I recommend using this and `-g` during development; don't use these flags for anything you're turning in)

Example usage:

- `./cat test.txt test2.txt` will concatenate 2 files
- `./cat -v test.txt test2.txt` concat 2 files but print filename first
- `./cat test.txt -v test2.txt` same as above, but demonstration of how flags can go anywhere
- `./cat -e 'line 1' -e 'line 2' test.txt` example of how to take an argument for a flag

Low-Level Programming Notes

- Standard process of opening the file, reading it, using the read data (in this case writing it to somewhere else), and then closing it
- `argv` contains the arguments, where `argv[0]` is the name of the program being run, so `argv[1]` is the actual first argument
- `argc` contains the number of arguments (`argv[0]` is counted as an argument)

Initial code:

```c
#define BUF_SZ 4096

// O_RDONLY opens a file readonly, as opposed to O_WRONLY for writeonly or O_RDWR
int fd = open(filepath, O_RDONLY);
char buf[BUF_SZ];
read(fd, buf, BUF_SZ);
// STDOUT_FILENO is aliased to 1 on Linux
write(STDOUT_FILENO, buf, BUF_SZ);
close(fd);
```

A couple of issues:

- If read is underfull (saw EOF before BUF_SZ characters), the write will write garbage past the end of the read
- If read is overfull (didn't see EOF after BUF_SZ characters), the read will cut off early
- write also doesn't have to write the entire buffer at once
- Not checking for errors on anything

Fix for the the first issue: `read` returns the number of bytes read, so just use that instead:

```c
ssize_t n = read(fd, buf, BUF_SZ);
write(STDOUT_FILENO, buf, n);
```

Fix for the next two: loop read and write until read returns 0 (EOF). This works because files have internal pointers to their data, so consecutive read/writes will keep advancing the file pointer instead of re-reading/writing the same thing.

```c
char buf[BUF_SZ];
while (1) {
    ssize_t n = read(fd, buf, BUF_SZ);
    if (n == 0) {
        break;
    }
    ssize_t written = 0;
    while (written < n) {
        // we need to advance buf manually with buf+written
        // so we don't write the start of the buffer every time
        ssize_t w = write(STDOUT_FILENO, buf + written, n - written);
        written += w;
    }
}
```

Fix for error checking: all of these functions return -1 on error, and then set `errno` to some number indicating the error code. A couple ways of working with `errno`:

- use it directly (not very meaningful)
- pass it to `strerror` to get a string error message (good for custom error messages)
- use `perror` to print the error message for you:
  - `perror("abc")` is equivalent to `fprintf(stderr, "abc: %s\n", strerror(errno))`

Argument Parsing Notes

- `getopt` in C is basically a really bad version of Python's `argparse`
- Pass in `argc`, `argv`, and an argument specifier
  - Argument specifier is a string of flags, where each character is a flag
  - Put a `:` after a flag to make it take an argument
- Loop `getopt` until it returns `-1`
- `getopt` will return a `?` on invalid parse
- A couple of special values:
  - `optarg` is a string representing the current argument
  - `optind` is the index of the first non-flag argument (aka first positional argument)
  - If flags come in the middle of the command, `getopt` will rearrange them to come before positional arguments
- Example argument parsing code:

```c
int opt;
int verbose = 0;
while ((opt = getopt(argc, argv, "ve:")) != -1) {
    switch (opt) {
        case 'v':
            verbose = 1;
            break;
        case 'e':
            printf("%s\n", optarg);
            break;
        case '?':
            return 1;
    }
}
```