Problem 1: System calls, error checking, and reporting

Caleb Zulawski

September 14, 2015

```c
/* copycat.c
 * Caleb Zulawski
 *
 * Entrance point of the program.
 */

#include "copycat.h"

int main(int argc, char* argv[]) {
    Options options;

    cc_parse_args(argc, argv, &options);
    cc_log(&options);
    cc_copy(&options);

    return 0;
}
```

```c
/* copycat.h
 * Caleb Zulawski
 *
 * Function and struct declarations, constants.
 */

#ifndef COPYCAT_H
#define COPYCAT_H

#define DEFAULT_BUFFER_SIZE 1024 // 1KB
#define DEFAULT_FILE_PERM   0664 // RW-RW-R--

// Command line options
typedef struct {
    char**        argv;
```

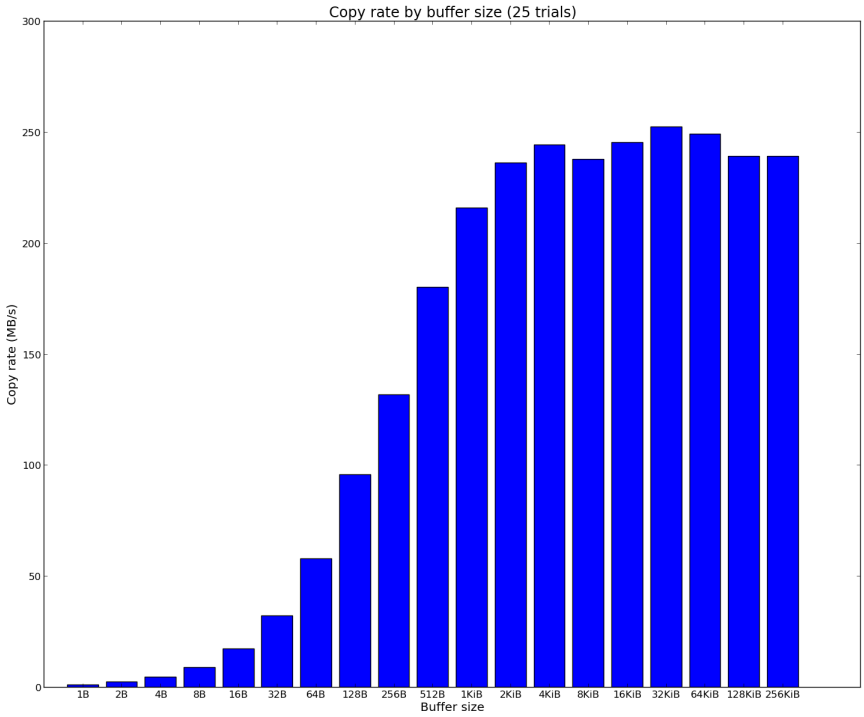

Figure 1: Performance

```
    int           argc;
    unsigned int  buffersize;
    int           outfile_index;
    int           infiles_index;
    int           verbose;
    unsigned int  mode;
} Options;

// Non-read/write error conditions
typedef enum {
    CC_NONE,
    CC_USAGE,
    CC_MALLOC_FAIL
} cc_error_t;

// Read/write error conditions
typedef enum {
    CC_F_NONE,
    CC_F_OPEN_RD,
    CC_F_OPEN_WR,
    CC_F_READ,
    CC_F_WRITE
} cc_file_error_t;

// Function declarations
cc_error_t      cc_parse_args(int argc, char* argv[], Options* options);

void            cc_error(cc_error_t e);

void            cc_error_f(cc_file_error_t e, int err, char filename[]);

void            cc_log(Options* options);

void            cc_copy(Options* options);

cc_file_error_t cc_copy_file(const int          fo,
                             const int          fi,
                             const unsigned int buf_len,
                             char*              buf,
                             int*               err
                            );

#endif /* COPYCAT_H */

/* copycat_cli.c
 * Caleb Zulawski
```

```
 *
 * Handles command line argument parsing, logging,
 * and displaying errors.
 */

#include "copycat.h"

/* Library functions */
#include <stdio.h>      // sscanf, printf
#include <getopt.h>     // getopt
#include <stdlib.h>     // exit
#include <string.h>     // strerror

/* Parses command line arguments and stores them in a struct */
cc_error_t cc_parse_args(int argc, char* argv[], Options* options) {
    options->buffersize    = DEFAULT_BUFFER_SIZE;
    options->outfile_index = -1;
    options->argc          = argc;
    options->argv          = argv;
    options->mode          = DEFAULT_FILE_PERM;

    int c;
    while ((c = getopt(argc, argv, "+b:m:o:vh")) != -1) {
        switch (c) {
            case 'b':
                if ( !sscanf(optarg, "%u", &options->buffersize) )
                    cc_error(CC_USAGE);
                break;
            case 'm':
                if ( !sscanf(optarg, "%o", &options->mode) )
                    cc_error(CC_USAGE);
                break;
            case 'o':
                options->outfile_index = optind-1;
                break;
            case 'v':
                options->verbose = 1;
                break;
            case 'h':
                cc_error(CC_USAGE);
                break;
            case '?':
                cc_error(CC_USAGE);
                break;
        }
    }
```

```
    options->infiles_index = optind;

    return CC_NONE;
}

/* Logs some information if the -v verbose flag is set */
void cc_log(Options* options) {
    if (options->verbose) {

        // OUTPUT FILE
        if (options->outfile_index != -1)
            printf("Output file:\t%s\n", options->argv[options->outfile_index]);
        else
            printf("Printing to standard output stream\n");

        // INPUT FILES
        for (int i = options->infiles_index; i < options->argc; i++) {
            printf("Input file:\t%s\n", options->argv[i]);
        }
        if (options->infiles_index == options->argc)
            printf("Reading from standard input stream\n");

        // BUFFER SIZE
        printf("Buffer size:\t%u\n", options->buffersize);

        // OUTPUT FILE PERMISSIONS
        printf("Output mode:\t%o\n", options->mode);
    }
}

/* Handles errors other than read/write associated */
void cc_error(cc_error_t e) {
    switch (e) {
        case CC_NONE:
            return;
        case CC_USAGE:
            printf("Usage: copycat [OPTION]... [FILE]...\n");
            printf("Concatenate FILE(s), or standard input, to standard output.\
            printf("Similar to GNU cat.\n\n");
            printf("  -v             print diagnostic messages to standard error\n
            printf("  -b SIZE        size of internal copy buffer, in bytes\n");
            printf("  -m MODE        file mode, in octal\n");
            printf("  -o FILE        output to FILE instead of standard output\n")
            printf("  -h             display this help and exit\n");
            exit(-1);
```

```
        case CC_MALLOC_FAIL:
            printf("Could not allocate buffer!\n");
            exit(-1);
    }
}

/* Handles read/write errors */
void cc_error_f(cc_file_error_t e, int err, char filename[]) {
    switch (e) {
        case CC_F_NONE:
            return;
        case CC_F_OPEN_RD:
        case CC_F_OPEN_WR:
            fprintf(stderr, "Error opening file %s: %s\n", filename, strerror(er
            exit(-1);
        case CC_F_READ:
            fprintf(stderr, "Error reading from file %s: %s\n", filename, strerr
            exit(-1);
        case CC_F_WRITE:
            fprintf(stderr, "Error writing to file %s: %s\n", filename, strerror
    }
}

/* copycat_copy.c
 * Caleb Zulawski
 *
 * Contains the functions for copying data from
 * the specified files.  Reports errors.
 */

#include "copycat.h"

/* Library functions */
#include <stdlib.h>     // malloc
#include <string.h>     // strcmp

/* Linux system calls */
#include <unistd.h>     // read, close
#include <errno.h>      // errno
#include <sys/types.h> // open, write
#include <sys/stat.h>  // open, write
#include <fcntl.h>      // open, write

/* Calls appropriate copy functions and handles errors */
void cc_copy(Options* options) {
    int  /*fee*/ fi, fo /*fum*/;
```

```
char**        argv           = options->argv;
int           argc           = options->argc;
int           outfile_index  = options->outfile_index;
int           infiles_index  = options->infiles_index;
mode_t        mode           = options->mode;
unsigned int  buffersize     = options->buffersize;

const int w_flags = O_WRONLY // Read only
                  | O_CREAT  // Create file if it doesn't exist
                  | O_TRUNC; // Truncate the file if it does

const int r_flags = O_RDONLY; // Read only

// Open output file
if (outfile_index != -1) {
    fo = open(argv[outfile_index], w_flags, (mode_t) mode);
    if (fo == -1)
        cc_error_f(CC_F_OPEN_WR, errno, argv[outfile_index]);
} else {
    fo = STDOUT_FILENO;
}

// Acquire buffer for copying
char* buffer = malloc(buffersize);
if (buffer == NULL)
    cc_error(CC_MALLOC_FAIL);

if (infiles_index >= argc) {
    // If no files were supplied, use STDIN
    fi = STDIN_FILENO;
    int err = 0;
    cc_file_error_t status = cc_copy_file(fi, fo, buffersize, buffer, &err);
    close(fi);
    if (status == CC_F_READ) {
        close(fo);
        cc_error_f(status, err, "STDIN");
    } else if (status == CC_F_WRITE) {
        close(fo);
        cc_error_f(status, err, argv[outfile_index]);
    }
} else {
    // If files were supplied, copy each in order
    for (int i = infiles_index; i < argc; i++) {
        if (!strcmp(argv[i], "-")) {
            fi = STDIN_FILENO;
        } else {
```

```
                fi = open(argv[i], r_flags);
                if (fi == -1)
                    cc_error_f(CC_F_OPEN_RD, errno, argv[i]);
            }

            int err = 0;
            cc_file_error_t status = cc_copy_file(fi, fo, buffersize, buffer, &e
            close(fi);
            if (status == CC_F_READ) {
                close(fo);
                cc_error_f(status, err, argv[i]);
            } else if (status == CC_F_WRITE) {
                close(fo);
                cc_error_f(status, err, argv[outfile_index]);
            }
        }
    }
    close(fo);
}

cc_file_error_t cc_copy_file(const int          fi,
                             const int          fo,
                             const unsigned int buf_len,
                             char*              buf,
                             int*               err)
{
    ssize_t bytes_read, bytes_written;

    while ((bytes_read = read(fi, buf, (size_t) buf_len))) {
        if (bytes_read == -1) {
            *err = errno;
            return CC_F_READ;
        }

        do {
            if (!(bytes_written = write(fo, buf, bytes_read))) {
                *err = errno;
                return CC_F_WRITE;
            }
        } while (bytes_written < bytes_read); // Handle partial write
    }
    return CC_F_NONE;
}
```