Many of the shell scripts have command sequences that look like the following (though the actual commands tend to be more proprietary):

(sort < a | cat b - | tr A-Z a-z > c) 2>> d

This command invokes three subcommands. The first runs the command sort with standard input being the file a and standard output being a unnamed pipe 1. The second runs the command cat b - with standard input being pipe 1 and standard output being pipe 2. The third runs the command tr A-Z a-z with standard input being pipe 2 and standard output being the file c. All three commands have standard error sent, via the same file descriptor, to file d in append-only mode.

BDSI's developers have several complaints about these shell scripts:

* The shell script syntax does not let them open files with special flags available at the system call level. For example, there is no way to open a file with the O\_DSYNC flag of the [open](http://pubs.opengroup.org/onlinepubs/9699919799/functions/open.html) system call.
* The developers want to be able to create arbitrary directed graphs of processes connected via pipes, but the shell syntax forces them into pipelines.
* The developers want a simpler way to invoke the shell, one that is more easily generated from their programs and scripts, one that does not require that they write a shell script. They do not mind if the simpler shell is harder for humans to use, because it's not intended to be used directly by programmers.

## Basic idea

To address these issues, your boss proposes a new program simpsh, short for "SIMPleton SHell", a very simple, stripped down shell. simpsh does not use a scripting language at all, and you do not interact with it at a terminal or give it a script to run. Instead, developers invoke the simpsh command by passing it arguments telling it which files to access, which pipes to create, and which subcommands to invoke. It then creates or accesses all the files and creates all the pipes processes needed to run the subcommands, and reports the processes's exit statuses as they exit.

For example, the abovementioned command in the standard shell could be run using the following simpsh command. This invocation uses standard shell syntax, because it is invoking simpsh from the standard shell; the command itself, though, is just an array of strings and simpsh interprets this array and executes the same three subcommands that the abovementioned shell command does.

simpsh \

--rdonly a \

--pipe \

--pipe \

--creat --trunc --wronly c \

--creat --append --wronly d \

--command 3 5 6 tr A-Z a-z \

--command 0 2 6 sort \

--command 1 4 6 cat b - \

--close 2 \

--close 4 \

--wait

This example invocation creates seven file descriptors:

1. A read only descriptor for the file a, created by the --rdonly option.
2. The read end of the first pipe, created by the first --pipe option.
3. The write end of the first pipe, also created by the first --pipe option.
4. The read end of the second pipe, created by the second --pipe option.
5. The write end of the second pipe, also created by the second --pipe option.
6. A write only descriptor for the file c, created by the first --wronly option as modified by the preceding --creat and --trunc.
7. A write only, append only descriptor for the file d, created by the --wronly option as modified by the preceding --creat and --append options.

It then creates three subprocesses:

* A subprocess with standard input, output, and error being the file descriptors numbered 3, 5, and 6 above. This subprocess runs the command tr with the two arguments A-Z and a-z.
* A subprocess with standard input, standard output, and standard error being the file descriptors numbered 0, 2, and 6 above, respectively. This subprocess runs the command sort with no arguments
* A subprocess with standard input, output, and error being the file descriptors numbered 1, 4, and 6 above. This subprocess runs the command cat with the two arguments b and -.

It then closes the write ends of the pipes, and waits for all three subprocesses to finish. As each finishes, it outputs "exit N" if it exited with status N or "signal S" if it terminated with signal number S, followed by the command and arguments. The output might look like this:

exit 0 sort

exit 0 cat b -

exit 0 tr A-Z a-z

although not necessarily in that order, depending on which order the subprocesses finished.

## simpsh options

Here is a detailed list of the command-line options that simpsh should support. Each option should be executed in sequence, left to right.

First are the file flags. These flags affect the next file that is opened. They are ignored if no later file is opened. Each file flag corresponds to an oflag value of [open](http://pubs.opengroup.org/onlinepubs/9699919799/functions/open.html); the corresponding oflag value is listed after the option. Also see [Opening and Closing Files](http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html) and [Open-time Flags](http://www.gnu.org/software/libc/manual/html_node/Open_002dtime-Flags.html).

--append

O\_APPEND

--cloexec

O\_CLOEXEC

--creat

O\_CREAT

--directory

O\_DIRECTORY

--dsync

O\_DSYNC

--excl

O\_EXCL

--nofollow

O\_NOFOLLOW

--nonblock

O\_NONBLOCK

--rsync

O\_RSYNC

--sync

O\_SYNC

--trunc

O\_TRUNC

Second are the file-opening options. These flags open files. Each file-opening option also corresponds to an oflag value, listed after the option. Each opened file is given a file number; file numbers start at 0 and increment after each file-opening option. Normally they increment by 1, but the --pipe option causes them to increment by 2.

--rdonly f

O\_RDONLY. Open the file f for reading only.

--rdwr f

O\_RDWR. Open the file f for reading and writing.

--wronly f

O\_WRONLY. Open the file f for writing only.

--pipe

Open a pipe. Unlike the other file options, this option does not take an argument. Also, it consumes two file numbers, not just one.

Third is the subcommand options:

--command i o e cmd args

Execute a command with standard input i, standard output o and standard error e; these values should correspond to earlier file or pipe options. The executable for the command is cmd and it has zero or more arguments args. None of the cmd and args operands begin with the two characters "--".

--wait

Wait for all commands to finish. As each finishes, output its exit status or signal number as described above, and a copy of the command (with spaces separating arguments) to standard output.

Finally, there are some miscellaneous options:

--close N

Close the Nth file that was opened by a file-opening option. For a pipe, this closes just one end of the pipe. Once file N is closed, it is an error to access it, just as it is an error to access any file number that has never been opened. File numbers are not reused by later file-opening options.

--verbose

Just before executing an option, output a line to standard output containing the option. If the option has operands, list them separated by spaces. Ensure that the line is actually output, and is not merely sitting in a buffer somewhere.

--profile

Just after executing an option, output a line to standard output containing the resources used. Use [getrusage](http://pubs.opengroup.org/onlinepubs/9699919799/functions/getrusage.html) and output a line containing as much useful information as you can glean from it.

--abort

Crash the shell. The shell itself should immediately dump core, via a segmentation violation.

--catch N

Catch signal N, where N is a decimal integer, with a handler that outputs the diagnostic N caught to stderr, and exits with status N. This exits the entire shell. N uses the same numbering as your system; for example, on GNU/Linux, a segmentation violation is signal 11.

--ignore N

Ignore signal N.

--default N

Use the default behavior for signal N.

--pause

Pause, waiting for a signal to arrive.

When there is a syntax error in an option (e.g., a missing operand), or where a file cannot be opened, or where is some other error in a system call, simpsh should report a diagnostic to standard error and should continue to the next option. However, simpsh should ignore any write errors to standard error, so that it does not get into an infinite loop outputting write-error diagnostics.

When simpsh finishes other than in response to a signal, it should finish with the worst result of all the subcommands that it ran and successfull waited for. By "worst" is meant the following: if any subcommand terminated with a signal, simpsh should terminate with the highest-numbered signal that a subcommand terminated with; otherwise simpshshould exit with status equal to the maximum of all the exit statuses of all the subcommands. If there are no such subcommands, or if no subcommands terminated with a signal and the maximum subcommand exit status is zero, simpsh should exit with status 0 if all options succeeded, and with status 1 one of them failed. For example, if a file could not be opened, simpsh must exit with nonzero status.

//lab1c.c

// lab1b.c

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <getopt.h>

#include <stdio.h>

#include <signal.h>

#include <errno.h>

#include <string.h>

#include <stddef.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <sys/mman.h>

#include <sys/time.h>

#include <sys/resource.h>

#define BUFSIZE 100

static int err = 0;

static struct rusage usage;

static struct rusage end\_usage;

/\* return true if the argument is a long option \*/

int is\_long\_option(char\* opt) {

if (strlen(opt) >= 2 && opt[0] == '-' && opt[1] == '-')

return 1;

return 0;

}

void sig\_handler(int sig) {

fprintf(stderr, "%d caught.\n", sig);

exit(sig);

}

void profile(int pflag, struct rusage\* start, char\* option) {

if (pflag == 1) {

if (getrusage(RUSAGE\_SELF, &end\_usage) < 0) {

fprintf(stderr, "getrusage() fails: %s\n", strerror(errno));

if (err < 1) err = 1;

}

else if (start == NULL) {

fprintf(stderr, "can't get starting time\n");

if (err < 1) err = 1;

}

else {

struct timeval usertime;

struct timeval systime;

timersub(&(end\_usage.ru\_utime), &(start->ru\_utime), &usertime);

timersub(&(end\_usage.ru\_stime), &(start->ru\_stime), &systime);

fprintf(stdout, "\'%s\'\t", option);

fprintf(stdout, "user time: %ld.%06lds\t", usertime.tv\_sec, usertime.tv\_usec);

fprintf(stdout, "system time: %ld.%06lds\n", systime.tv\_sec, systime.tv\_usec);

fflush(stdout);

}

}

}

struct rusage start\_profile(int pflag) {

static struct rusage start\_usage;

if (pflag == 1) {

if (getrusage(RUSAGE\_SELF, &start\_usage) < 0) {

fprintf(stderr, "getrusage() fails: %s\n", strerror(errno));

if (err < 1) err = 1;

}

}

return start\_usage;

}

int main(int argc, char\* argv[]) {

/\* declare options and flags \*/

static struct option longopts[] = {

{ "rdonly", required\_argument, NULL, O\_RDONLY },

{ "rdwr", required\_argument, NULL, O\_RDWR },

{ "wronly", required\_argument, NULL, O\_WRONLY },

{ "pipe", no\_argument, NULL, 'p' },

{ "command", required\_argument, NULL, 'c' },

{ "wait", no\_argument, NULL, 't' },

{ "close", required\_argument, NULL, 's' },

{ "verbose", no\_argument, NULL, 'v' },

{ "profile", no\_argument, NULL, 'o' },

{ "abort", no\_argument, NULL, 'a' },

{ "catch", required\_argument, NULL, 'h' },

{ "ignore", required\_argument, NULL, 'i' },

{ "default", required\_argument, NULL, 'f' },

{ "pause", no\_argument, NULL, 'u' },

// flags

{ "append", no\_argument, NULL, O\_APPEND },

{ "cloexec", no\_argument, NULL, O\_CLOEXEC },

{ "creat", no\_argument, NULL, O\_CREAT },

{ "directory", no\_argument, NULL, O\_DIRECTORY },

{ "dsync", no\_argument, NULL, O\_DSYNC },

{ "excl", no\_argument, NULL, O\_EXCL },

{ "nofollow", no\_argument, NULL, O\_NOFOLLOW },

{ "nonblock", no\_argument, NULL, O\_NONBLOCK },

{ "rsync", no\_argument, NULL, O\_RSYNC },

{ "sync", no\_argument, NULL, O\_SYNC },

{ "trunc", no\_argument, NULL, O\_TRUNC },

{ 0,0,0,0 }

};

/\* declare variables \*/

int opt = 0;

int brk = 0;

int vflag = 0;

int crtflag = 0;

int pflag = 0;

int cmdind = 0;

int flag = 0;

int sig = 0;

int ifd = 0;

/\* file descriptors \*/

int f\_ind = 0;

int\* fds;

int NFILES = 20;

fds = malloc(NFILES\*sizeof(int));

if (!fds) {

fprintf(stderr, "fds[] malloc() fails: %s\n", strerror(errno));

exit(-1);

}

int\* file\_valid;

file\_valid = malloc(NFILES\*sizeof(int));

if (!file\_valid) {

fprintf(stderr, "fds[] malloc() fails: %s\n", strerror(errno));

exit(-1);

}

for (int i = 0; i < NFILES; i++) {

file\_valid[i] = 1;

}

/\* file numbers for subcommand \*/

char\* subcmd;

int\* subcmd\_files;

subcmd\_files = malloc(3\*sizeof(int));

if (!subcmd\_files) {

fprintf(stderr, "subcmd\_files[] malloc() fails: %s\n", strerror(errno));

exit(-1);

}

char\*\* subarg\_buf = malloc(sizeof(char\*));

if (!subarg\_buf) {

fprintf(stderr, "subarg\_buf[] malloc() fails: %s\n", strerror(errno));

exit(-1);

}

/\* processes \*/

int nProcesses = 0;

int\* processes = malloc(sizeof(int));

int\* processes\_args\_ind = malloc(sizeof(int));

if (!processes) {

fprintf(stderr, "processes[] malloc() fails: %s\n", strerror(errno));

exit(-1);

}

if (!processes\_args\_ind) {

fprintf(stderr, "processes[] malloc() fails: %s\n", strerror(errno));

exit(-1);

}

/\* handling errors \*/

//int err = 0;

int status = 0;

int exit\_status = 0;

/\* creating pipes \*/

int pipefd[2];

/\* usage \*/

//struct rusage usage;

while(1) {

opt = getopt\_long(argc, argv, "", longopts, NULL);

if (opt == -1)

break;

switch(opt) {

/\* --verbose \*/

case 'v':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s\n", argv[optind-1]);

fflush(stdout);

}

vflag = 1;

profile(pflag, &usage, argv[optind-1]);

break;

/\* flags \*/

case O\_CREAT: // mode must be supplied when O\_CREAT or O\_TMPFILE is specified in flags

crtflag = 1;

case O\_APPEND:

case O\_CLOEXEC: // close after execution

case O\_DIRECTORY: // fail if not a directory

case O\_DSYNC:

case O\_EXCL: // use with O\_CREAT, not overwrite existing file

case O\_NOFOLLOW: //If pathname is a symbolic link, then the open fails, with the error ELOOP.

case O\_NONBLOCK:

//case O\_RSYNC:

case O\_SYNC:

case O\_TRUNC:

if (vflag){

fprintf(stdout, "%s %s\n", argv[optind-1], optarg);

fflush(stdout);

}

flag = flag | opt;

break;

/\* --rdonly, --wronly, --rdwr \*/

case O\_RDONLY:

case O\_WRONLY:

case O\_RDWR:

usage = start\_profile(pflag);

if (vflag){

fprintf(stdout, "%s %s\n", argv[optind-2], optarg);

fflush(stdout);

}

/\* resize fd arrays \*/

if (f\_ind == NFILES) {

NFILES \*= 2;

fds = realloc(fds, NFILES \* sizeof(int));

file\_valid = realloc(file\_valid, NFILES \* sizeof(int));

if (!fds || !file\_valid) {

fprintf(stderr, "fds[] or file\_valid[] realloc() fails: %s\n", strerror(errno));

exit(-1);

}

for (int i = NFILES/2; i < NFILES; i++) {

file\_valid[i] = 1;

}

}

/\* open file with the correct flag \*/

flag = (flag | opt);

if (crtflag == 1)

ifd = open(optarg, flag, 0644);

else

ifd = open(optarg, flag);

/\* store the file descriptor \*/

file\_valid[f\_ind] = 1;

fds[f\_ind++] = ifd;

if (!optarg || ifd < 0) {

fprintf(stderr, "\'%s\': can't open file %s. ", argv[optind-2], optarg);

fprintf(stderr, "%s\n", strerror(errno));

if (err < 1) err = 1;

file\_valid[f\_ind] = 0;

}

/\* reset flag \*/

flag = 0;

// if (pflag) {

// if (getrusage(RUSAGE\_SELF, &usage) < 0) {

// fprintf(stderr, "getrusage() fails: %s\n", strerror(errno));

// if (err < 1) err = 1;

// }

// else {

// //nd-2]);

// fprintf(stdout, "user time: %ld.%06ld\t", usage.ru\_utime.tv\_sec, usage.ru\_utime.tv\_usec);

// fprintf(stdout, "system time: %ld.%06ld\n", usage.ru\_stime.tv\_sec, usage.ru\_stime.tv\_usec);

// fflush(stdout);

// }

// }

profile(pflag, &usage, argv[optind-2]);

break;

/\* --pipe \*/

case 'p':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s\n", argv[optind-1]);

fflush(stdout);

}

// open two file numbers: read end and write end

// data written to the write end of the pipe is buffered by the kernel

// until it is read from the read end of the pipe.

if (pipe(pipefd) < 0) {

fprintf(stderr, "pipe() fails: %s\n", strerror(errno));

// max error

if (err < 1) err = 1;

}

else {

if (f\_ind == NFILES) {

NFILES \*= 2;

fds = realloc(fds, NFILES \* sizeof(int));

if (!fds) {

fprintf(stderr, "fds[] realloc() fails: %s\n", strerror(errno));

exit(-1);

}

}

/\* increment two file descriptors \*/

file\_valid[f\_ind] = 1;

fds[f\_ind++] = pipefd[0];

file\_valid[f\_ind] = 1;

fds[f\_ind++] = pipefd[1];

}

/\* output profile \*/

profile(pflag, &usage, argv[optind-1]);

break;

/\* --command i o e cmd agrs \*/

case 'c':

usage = start\_profile(pflag);

// find command index

cmdind = optind-2;

// move optind to the next long option

while ((optind < argc) && (is\_long\_option(argv[optind]) == 0))

optind++;

// --verbose

if (vflag) {

fprintf(stdout, "--command ");

for (int i = cmdind+1; (i < argc) && is\_long\_option(argv[i]) == 0; i++) {

fprintf(stdout, "%s ", argv[i]);

}

fprintf(stdout, "\n");

// always flush stdout to avoid stdout hanging in buffer

fflush(stdout);

}

/\* check if --command is followed by sufficient arguments \*/

int i = cmdind+1;

int cmdargc = 0;

while ((i < argc) && (is\_long\_option(argv[i]) == 0)) {

i++;

cmdargc++;

}

if (cmdargc < 4) {

fprintf(stderr, "No enough arguments for --command: %d\n", cmdargc);

if (err < 1) err = 1;

/\* output profile \*/

profile(pflag, &usage, argv[cmdind]);

break;

}

/\* identify 3 file descriptors, continue to the next option if invalid file no. \*/

char\* endptr = NULL;

for (int i = 1; i < 4; i++) {

subcmd\_files[i-1] = (int) strtol(argv[cmdind+i], &endptr, 10);

if (endptr == argv[cmdind+i]) {

fprintf(stderr, "Please provide a valid file. %s is not a number.\n", argv[cmdind+i]);

if (err < 1) err = 1;

brk = 1;;

}

}

if (brk == 1) {

/\* output profile \*/

profile(pflag, &usage, argv[cmdind]);

break;

}

for (int i = 0; i < 3; i++) {

if (subcmd\_files[i] >= f\_ind || subcmd\_files[i] < 0) {

fprintf(stderr, "Please provide a valid file. %d is not a correct index.\n", subcmd\_files[i]);

if (err < 1) err = 1;

brk = 1;

}

}

if (brk == 1) {

/\* output profile \*/

profile(pflag, &usage, argv[cmdind]);

break;

}

/\* check if the 3 files are accessible \*/

for (int i = 0; i < 3; i++) {

if (file\_valid[subcmd\_files[i]] == 0) {

fprintf(stderr, "File %d is not valid.\n", subcmd\_files[i]);

err = 1;

}

}

/\* identify subcommand \*/

subcmd = argv[cmdind+4];

int si = cmdind+5;

int subargc = 1;

subarg\_buf[0] = subcmd;

/\* store subcommand arguments into a buffer \*/

while ((si < argc)) {

if (is\_long\_option(argv[si]) == 0) {

subarg\_buf = realloc(subarg\_buf, (subargc+1) \* sizeof(char\*));

if (!subarg\_buf) {

fprintf(stderr, "subarg\_buf[] realloc() fails: %s\n", strerror(errno));

exit(-1);

}

subarg\_buf[subargc++] = argv[si++];

}

else

break;

}

subarg\_buf = realloc(subarg\_buf, (subargc+1) \* sizeof(char\*));

if (!subarg\_buf) {

fprintf(stderr, "subarg\_buf[] realloc() fails: %s\n", strerror(errno));

exit(-1);

}

subarg\_buf[subargc] = NULL;

/\* start a new process to execute subcommand \*/

if (subcmd && subarg\_buf) {

pid\_t pid = fork();

if (pid < 0) {

fprintf(stderr, "Can't create a new process\n");

if (err < 1) err = 1;

/\* output profile \*/

profile(pflag, &usage, argv[cmdind]);

break;

}

/\* Child process \*/

else if (pid == 0) {

// fprintf(stderr, "subcmd\_files[0]: %d\n", subcmd\_files[0]);

// fprintf(stderr, "subcmd\_files[1]: %d\n", subcmd\_files[1]);

// fprintf(stderr, "subcmd\_files[2]: %d\n", subcmd\_files[2]);

/\* redirect i/o \*/

if (dup2(fds[subcmd\_files[0]], 0) < 0) {

fprintf(stderr, "dup() --command input failed. File index: %d\n", subcmd\_files[0]);

}

if (dup2(fds[subcmd\_files[1]], 1) < 0) {

fprintf(stderr, "dup() --command output failed. File index: %d\n", subcmd\_files[1]);

}

if (dup2(fds[subcmd\_files[2]], 2) < 0) {

fprintf(stderr, "dup() --command error failed. File index: %d\n", subcmd\_files[2]);

}

/\* close files to avoid hanging \*/

for (int i = 0; i < f\_ind; i++) {

close(fds[i]);

file\_valid[i] = 0;

}

/\* execute subcommand \*/

if (execvp(subarg\_buf[0], subarg\_buf) < 0) {

fprintf(stderr, "%s error: %s\n", subarg\_buf[0], strerror(errno));

}

}

/\* Parent process \*/

else {

/\* store processes pids and arguments for each pid \*/

processes = realloc(processes, (nProcesses+1)\*sizeof(int));

processes\_args\_ind = realloc(processes\_args\_ind, (nProcesses+1)\*sizeof(int\*));

if (!processes || !processes\_args\_ind) {

fprintf(stderr, "processes[] realloc() fails: %s\n", strerror(errno));

exit(-1);

}

processes[nProcesses] = pid;

processes\_args\_ind[nProcesses++] = cmdind+4;

}

}

/\* output profile \*/

profile(pflag, &usage, argv[cmdind]);

break;

/\* --wait \*/

case 't':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s\n", argv[optind-1]);

fflush(stdout);

}

fflush(stdout);

i = 0;

while(1) {

/\* wait for all child processes to terminate \*/

pid\_t p = waitpid(-1, &status, 0);

if (i >= nProcesses || p < 0)

break;

// returns true if the child terminated normally

if (WIFEXITED(status)) {

// returns the exit status of the child

exit\_status = WEXITSTATUS(status);

fprintf(stdout, "exit %d ", exit\_status);

}

// returns true if the child terminated with a signal

else if (WIFSIGNALED(status)) {

// returns the signal of the child

exit\_status = WTERMSIG(status);

fprintf(stdout, "signal %d ", exit\_status);

// exit status is (128 + signal)

exit\_status += 128;

}

else {

fprintf(stderr, "Child exit status is unknown\n");

/\* output profile \*/

profile(pflag, &usage, argv[optind-1]);

break;

}

// highest exit code

if (exit\_status > err)

err = exit\_status;

/\* print the argumetns of the child process program \*/

int pro\_ind = 0;

for (int i = 0; i < nProcesses; i++) {

if (processes[i] == p) {

// process index to search for argument position in argv[]:

pro\_ind = i;

break;

}

}

for (int j = processes\_args\_ind[pro\_ind]; j < argc; j++) {

if (is\_long\_option(argv[j]))

break;

fprintf(stdout, "%s ", argv[j]);

}

fprintf(stdout, "\n");

fflush(stdout);

i++;

}

/\* output profile \*/

if (pflag == 1) {

struct rusage child\_u;

if (getrusage(RUSAGE\_CHILDREN, &child\_u) < 0) {

fprintf(stderr, "getrusage() fails: %s\n", strerror(errno));

if (err < 1) err = 1;

}

else {

fprintf(stdout, "children sum: \t");

fprintf(stdout, "user time: %ld.%06lds\t", child\_u.ru\_utime.tv\_sec, child\_u.ru\_utime.tv\_usec);

fprintf(stdout, "system time: %ld.%06lds\n", child\_u.ru\_stime.tv\_sec, child\_u.ru\_stime.tv\_usec);

fflush(stdout);

}

}

profile(pflag, &usage, argv[optind-1]);

break;

/\* --close N \*/

case 's':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s %s\n", argv[optind-2], optarg);

fflush(stdout);

}

if (!optarg) {

fprintf(stderr, "Please provide a valid file index\n");

if (err < 1) err = 1;

break;

}

/\* convert string to int \*/

endptr = NULL;

int n = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg) {

fprintf(stderr, "Please provide a valid file index. %s is not a number.\n", optarg);

if (err < 1) err = 1;

break;

}

/\* file index needs to be in range \*/

if (n < f\_ind && n >= 0) {

close(fds[n]);

// IMPORTANT: invalidate the file

file\_valid[n] = 0;

}

else {

fprintf(stderr, "Please provide a valid file index. %s is out of range.\n", optarg);

if (err < 1) err = 1;

}

/\* output profile \*/

profile(pflag, &usage, argv[optind-2]);

break;

/\* --profile \*/

case 'o':

if (vflag) {

fprintf(stdout, "%s %s\n", argv[optind-2], optarg);

fflush(stdout);

}

pflag = 1;

break;

/\* --abort \*/

case 'a':

if (vflag) {

fprintf(stdout, "%s\n", argv[optind-1]);

fflush(stdout);

}

fflush(stdout);

/\* force a segmentation fault \*/

char\* a = NULL;

\*a = 0;

break;

/\* --catch N \*/

case 'h':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s %s\n", argv[optind-2], optarg);

fflush(stdout);

}

if (!optarg) {

fprintf(stderr, "Please provide a valid file index\n");

if (err < 1) err = 1;

break;

}

/\* convert string to int \*/

endptr = NULL;

sig = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg) {

fprintf(stderr, "Please provide a valid signal. %s is not a number.\n", optarg);

if (err < 1) err = 1;

break;

}

signal(sig, sig\_handler);

/\* output profile \*/

profile(pflag, &usage, argv[optind-2]);

break;

/\* --ignore N \*/

case 'i':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s %s\n", argv[optind-2], optarg);

fflush(stdout);

}

if (!optarg) {

fprintf(stderr, "Please provide a valid file index\n");

if (err < 1) err = 1;

break;

}

/\* convert string to int \*/

endptr = NULL;

sig = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg) {

fprintf(stderr, "Please provide a valid signal. %s is not a number.\n", optarg);

if (err < 1) err = 1;

break;

}

signal(sig, SIG\_IGN);

/\* output profile \*/

profile(pflag, &usage, argv[optind-2]);

break;

/\* --default N \*/

case 'f':

usage = start\_profile(pflag);

if (vflag) {

fprintf(stdout, "%s %s\n", argv[optind-2], optarg);

fflush(stdout);

}

if (!optarg) {

fprintf(stderr, "Please provide a valid file index\n");

if (err < 1) err = 1;

break;

}

/\* convert string to int \*/

endptr = NULL;

sig = (int) strtol(optarg, &endptr, 10);

if (endptr == optarg) {

fprintf(stderr, "Please provide a valid signal. %s is not a number.\n", optarg);

if (err < 1) err = 1;

break;

}

signal(sig, SIG\_DFL);

/\* output profile \*/

profile(pflag, &usage, argv[optind-2]);

break;

/\* --pause \*/

case 'u':

usage = start\_profile(pflag);

pause();

/\* output profile \*/

profile(pflag, &usage, argv[optind-1]);

break;

default:

fprintf(stderr, "Error: unrecognized option \'%s\'\n", argv[optind-1]);

if (err < 1) err = 1;

break;

}

}

exit(err);

}

LAB0 PROJECT DESCRIPTION:

1. (if you do not already have one) bring up or obtain access to a GNU/Linux development environment. Your development environment should include (at least):

* gcc
* libc (e.g., glibc or libc6-dev)
* make
* gdb

1. (if you are not already familiar with them) study the following manual sections:

* POSIX file operations ... open(2), creat(2), close(2), dup(2), read(2), write(2), exit(2), signal(2), and this brief tutorial on [file descriptor manipulation](http://web.cs.ucla.edu/classes/winter19/cs111/labs/fd_juggling.html).
* strerror(3) ... function that interprets the error codes returned from failed system calls.
* getopt(3) ... the framework we will use for argument handling in all projects for this course.
* tar(1) (and the -z option) ... program for archiving files in a tarball.
* gdb(1) (and the run, bt, list, print and break commands in particular) ... a GNU/Linux debugger for C/C++ programs.
* You will probably find understanding getopt(3) to be the most difficult part of this project. Feel free to seek out other examples/tutorials for these functions, but make sure you cite those sources in your README.

1. write a program that copies its standard input to its standard output by read(2)-ing from file descriptor 0 (until encountering an end of file) and write(2)-ing to file descriptor 1. If no errors (other than EOF) are encountered, your program should exit(2) with status 0.
   * + Your program executable should be called lab0, and accept the following (optional) command line arguments (in any combination or order; duplicate uses of the same option with the same or different option-arguments are allowed):
     + --input=filename ... use the specified file as standard input (instead of what is already being used as standard input)   
       If you are unable to open the specified input file, report the failure (on stderr, file descriptor 2) using fprintf(3), and exit(2) with status 1.
     + --output=filename ... create the specified file and use it as standard output (instead of whatever is already being used as standard output)   
       If the file already exists, truncate it to zero size. If you are unable to create or truncate the specified output file, report the failure (on stderr, file descriptor 2) using fprintf(3), and exit(2) with status 2.
     + --segfault ... force a segmentation fault (e.g., by calling a subroutine that sets a char \* pointer to NULL and then stores through the null pointer). If this argument is specified, do it immediately, and do not copy from stdin to stdout.
     + --catch ... use signal(2) to register a SIGSEGV handler that catches the segmentation fault, logs an error message (on stderr, file descriptor 2) and exit(2) with status 4.
     + --dump-core ... dump core on segmentation faults.
     + When you print out an error message (e.g., because an open failed), your message should include enough information to enable a user to understand not merely the nature of the problem but its cause ... for example:
     + which argument caused the problem ... e.g., --input
     + which file could not be opened ... e.g., myfile.txt
     + the reason it could not be opened ... e.g., no such file
     + Do your argument parsing with getopt\_long(3). This is, for historical reasons, a somewhat convoluted API, but ...
     + it is very similar APIs are used in many other languages and systems.
     + I want you to gain experience with the very common trial-and-error process of learning how to use a non-trivial API.
     + If you encounter an unrecognized argument you should print out an error message including a correct usage line, and exit(2) with status 3.
     + To ensure that operations are performed in the right order when multiple arguments are specified, it is suggested that you:
     + First check all arguments for argument syntax.
     + Then carry out all actions in the same order as the arguments.
     + It is relatively easy to generate primitive error messages with perror(3), but if you study the documentation you will see how to get access to the underlying error descriptions, which you could then use with fprintf(stderr,... to generate better formatted error messages to the correct file descriptor.
     + Note that to use the advanced debugging features of gdb(1) you will need to compile your program with the -g option, which adds debugging symbol table information to your program.

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <getopt.h>

#include <stdio.h>

#include <signal.h>

#include <errno.h>

#include <string.h>

#define SIZE 10000

static int dcflag = 0;

static int catchflag = 0;

// register a SIGSEGV handler that catches the segmentation fault

void sigsegv\_handler() {

// catch a seg fault when no core dumped

if (catchflag == 1 && dcflag == 0) {

fprintf(stderr, "SIGSEGV caught\n");

exit(4);

}

// force a seg fault otherwise

if (dcflag == 1) {

char\* s1 = NULL;

char s2 = \*s1;

s2 = s2;

}

}

int main(int argc, char\* argv[]) {

/\*

struct option {

const char \*name;

int has\_arg; -> no\_argument, required\_argument, optional argument

int \*flag; -> NULL returns val, otherwise 0

int val;

}

\*/

static struct option longopts[] = {

{ "input", required\_argument, NULL, 'i' },

{ "output", required\_argument, NULL, 'o' },

{ "segfault", no\_argument, NULL, 's' },

{ "catch", no\_argument, NULL, 'c' },

{ "dump-core", no\_argument, NULL, 'd'},

{ 0, 0, 0, 0 }

};

/\* declare varaibles \*/

int ifd = 0;

int ofd = 0;

int sfflag = 0;

char opt = '0';

char buffer[SIZE];

char\* seg1 = NULL;

char seg2 = '0';

while (1) {

opt = getopt\_long(argc, argv, "i:o:scd", longopts, NULL);

if (opt == -1)

break;

switch(opt) {

case 'i':

case 'o':

case 'c':

case 'd':

case 's':

break;

default:

fprintf(stderr, "Usage: %s\n", argv[0]);

fprintf(stderr, "--input=[input\_file] : specify an input file\n");

fprintf(stderr, "--output=[output\_file] : specify an output file\n");

fprintf(stderr, "--segfault : force a segmentation fault\n");

fprintf(stderr, "--catch : catch a sementation fault\n");

fprintf(stderr, "--dump-core : dump core on segmentation faults\n");

exit(1);

}

}

// there are arguments left which are neither option nor option arguments

if (argv[optind] != NULL) {

fprintf(stderr, "Syntax error: '%s'\n", argv[optind]);

fprintf(stderr, "--input=[input\_file] : specify an input file\n");

fprintf(stderr, "--output=[output\_file] : specify an output file\n");

fprintf(stderr, "--segfault : force a segmentation fault\n");

fprintf(stderr, "--catch : catch a sementation fault\n");

fprintf(stderr, "--dump-core : dump core on segmentation faults\n");

exit(1);

}

optind = 1;

while (1) {

opt = getopt\_long(argc, argv, "i:o:scd", longopts, NULL);

if (opt == -1)

break;

switch (opt) {

case 'i':

ifd = open(optarg, O\_RDONLY);

if (!optarg) {

fprintf(stderr, "--input error: can't open file %s. ", optarg);

fprintf(stderr, "%s\n", strerror(errno));

exit(2);

}

// direct to the input file normally

if (ifd >= 0) {

close(0);

dup(ifd);

close(ifd);

}

// If you are unable to open the specified input file, exit(2) with status 1.

else {

fprintf(stderr, "--input error: can't open file %s. ", optarg);

fprintf(stderr, "%s\n", strerror(errno));

exit(2);

}

break;

case 'o':

// If the file already exists, truncate it to zero size.

ofd = creat(optarg, 0666); //rw-rw-rw-

// ofd = open(optarg, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

if (!optarg) {

fprintf(stderr, "--output error: can't open file %s. ", optarg);

fprintf(stderr, "%s\n", strerror(errno));

exit(3);

}

// direct to the output normally

if (ofd >= 0) {

close(1);

dup(ofd);

close(ofd);

}

// If you are unable to create or truncate the specified output file, exit with status 2.

else {

fprintf(stderr, "--output error: can't create the output file %s. ", optarg);

fprintf(stderr, "%s\n", strerror(errno));

exit(3);

}

break;

case 's':

// force a segfault

seg2 = \*seg1;

seg2 = seg2;

sfflag = 1;

break;

case 'c':

// catch a segfault

catchflag = 1;

dcflag = 0;

// catch action is executed in the end

signal(SIGSEGV, sigsegv\_handler);

break;

case 'd':

catchflag = 0;

dcflag = 1;

break;

case ':':

printf("Missing arg\n");

break;

default:

// unrecognized argument

fprintf(stderr, "Usage: %s\n", argv[0]);

fprintf(stderr, "--input=[input\_file] : specify an input file\n");

fprintf(stderr, "--output=[output\_file] : specify an output file\n");

fprintf(stderr, "--segfault : force a segmentation fault\n");

fprintf(stderr, "--catch : catch a sementation fault\n");

fprintf(stderr, "--dump-core : dump core on segmentation faults\n");

exit(1);

}

}

if (sfflag == 1) {

seg2 = \*seg1;

seg2 = seg2;

}

else {

while(1) {

int r = read(0, &buffer, SIZE);

if (r < 0) {

fprintf(stderr, "Error: can't read from the input\n");

exit(errno);

}

else if (r == 0)

exit(0);

else {

int w = write(1, &buffer, r);

if (w < 0) {

fprintf(stderr, "Error: can't write to the output\n");

exit(errno);

}

}

}

}

exit(0);

}