Design Document: dog.cpp

Cristian C. Castillo
CruzID: ccarri11
Baubak Saadat
CruzID: saadat

Goal

The objective of this assignment is to emulate the cat's (Unix) program's basic functionality without advocating flags of any sort. From the command line, files may be present as arguments to the dog file. However, if no files exist as arguments, dog, and files with the character named (-) dash, shall echo the user's entry. The dog file shall contain code that invokes a series of system calls that utilize a file descriptor (Unix methodology file invocation derived from operating system hierarchy) to read, write, and open files. The yearned result is that dog shall copy data from each file provided as an argument via the command line to standard output. Dog's performance will replicate the exact cat commands in terms of performance and functionality.

Assumptions

My presumed assumption is arguments shall be read from int main's argv[]. Standard input (0)/outputs(1) shall use special system calls to read/write in lieu of the input/output through the iostream's norms (e.g. cout in C++); imitating the library in its entirety.

Description of data structures

The *argv[] is declared within the int main() parameters and is used to help access the files and keep track of the number of inputs given by the user in this program. *argv[] is an array containing the names of all the inputs made through the terminal when executing the ./dog program

The code must use fixed-size buffers , and may allocate no more than 32KiB of Memory for each file read(either via malloc() or as a direct variable declaration).

Assignment Question

The code differs in user privileges. Essentially with system calls we are accessing the Kernel Operating System, enabling max privilege on a monolithic system and direct access to lower levels, e.g. disk, process control, device management, etc. By interacting at such a low level (practically at the intersection where hardware/software meet to link and transmit - powerful but dangerous) we can leverage control to request services, memory allocation/management of resources, and privileges to optimize performance when using such system calls. Whereas standard handling file is less privileged (user mode - safe), and a derivative of such stated; a wrapper that cannot tap into the realm of Kernel mode to complete and request such services.

From the user mode, one would use a library to access a defined keyword (wrapper) to output, read, and open files. Such complexity involves layering, modularization, abstraction, and a clear specification at the level of indirection. Though these predefined libraries exist to use with ease, the creation of such libraries exhibit mass complexity. For example by handling a regular file to output content we would simply call upon the iostream library, and cout/cin the content we want written/read to our terminal. This simplicity relies on layering/modularization, whereas the user only needs to know the specification of the module. Once the spec is known, we become aware of the robustness that this black box allows. In detail the black box is the abstraction, where one only needs to know what goes in, and what comes out (knowing the internals do not matter). Such abstraction is achieved by breaking the components/modules into logical small black boxes, allowing the development of smaller units, and the ability to interchange them out amongst other coexisting components/modules. By modularization (putting up walls in the code) we constraint the complexity involved, direct flow, and decrease debugging time. Note that modularization only creates the interconnections (arrows), it is layering responsibility to structure how those arrows are structured amongst the components/modules, enabling an abstraction where the user only needs to know the top of hierarchy and nothing more under the hood. The interaction of requirements also comes into play, as we note the cout/cin only has one job, meaning that the generality of such a tool is specialized; resulting in less complexity, whereas iostream (hierarchy of abstraction not visible to the common user) would contain many other embedded tools, increasing generality and resulting in an increase of complexity. Should the generality of such a tool increase, so does complexity. All such techniques mentioned Modularization, Hierarchy, Layering, Abstraction, are ways to limit complexity, which are ironically similar. However, Naming is also notable here and without proper Naming, code would not bind accordingly or swap out the appropriate module/components in memory through late binding. Nevertheless, all techniques that constrain and limit complexity as shown to handle the file inputs are solutions that mitigated the cognitive burden on complexity which allows mere mortals such as you and I to comprehend such systems, resulting in the iterations of development of more complex systems by following such techniques. In regards to my implementation of handling inputs via system calls, these techniques too were used, and are clearly defined. E.g. the ./dog with zero arguments segment of my code is a module/component that was constructed to read/write system calls upon user input via terminal (see Module Box 1 & Module Box of 1), was the same component packaged into a box, modularized, and reimplemented for the dash argument by altering only the robustness at the abstraction level (redefining tolerable inputs to my black box at the spec and looking for concise inputs by maintaining a margin of safety with flags). By doing this, the dash is none the wiser, resulting in a decrease of complexity by layering the components accordingly as a feasible solution.

Design

The program shall have various phases, but the initial step is checking the amount of arguments passed to the program dog. If only ./dog is present, argc shall equal one, hence, if n amount of arguments are passed, e.g./dog foo2 foo3… then the program shall equal argc amount of n arguments, in this case n is 3. However, the program shall first check for the file via a system call (if the file is present in one’s directory read, else create), and in return store such call in a file descriptor to enable data to be read from an already 32Kib initialized buffer as instructed; along with checking for directory/files, outputting an error message should the arguments be of any other type then a file. If the file descriptor is valid, and only ./dog is present, ./dog shall replicate the cat command by receiving standard input from the user (reading the data), and on “Enter”, the buffer shall standard out the content as is; promptly awaiting another standard in user entry. Just like cat, ./dog is only able to escape from the infinite loop via ctrl+d, as there is no break statement for the cat command, also applies to ./dog with one dash argument in the command line. Shall ./dog not open successfully, a warning followed by an exit 1 shall be asynchronously called upon. If ./dog receives more arguments than itself, the program will check immediately upon a “-”, and compare the string to pointers argv[], and if recognized as the first argument, shall also replicate the ./dog program with no arguments. Should “-” be the third, fourth, nth argument, any argument file (argc) shall standard output via a system call, and then begin replicating the ./dog argument procedure imitating the exact process of the cat command.

This program will only adhere to one dash, additional dashes will simply be ignored following the output of any file arguments, should they be provided; checking if each is of file type. Lastly, if no dash is present in n amount of arguments greater than 1, then the order that the file arguments were first receive will open, check if the file descriptor is valid (non valid will throw a warn error and exit 1, indicating issue by exiting the program), and if valid, the files will write it's content in the order it was received line by line (via standard output); ending the program via the terminal.

Test Case

Valgrind was used to check for memory leaks via the terminal. Please note that since an infinite loop is present in this program, valgrind will halt at a pause unless a break is present in the code. However, in this case the loop is exited with ctrl+d indicating EOF, this same interaction (ctrl+d) can be used on valgrind to unpause testing. The stressed results exhibit zero leaks.

Functions

- stat (const char *path, struct stat *buf) : A unix system call that returns file attributes about an inode (a data structure that stores varios info about a file in Linux, such as access mode (read,write,execute permissions), ownership, file type, file size, group, number of links, etc. Needless to say, stat's st_mode was accessed in dog_definition.cpp as struct stat &buf (provided second argument), along with the first argument const char *path as the file arguments. By tapping into st_mode we enabled features that allowed us to utilize S_ISREG/S_ISDIR, to distinguish between a file and a directory. Stat contains many other features that when provided the prompted arguments give access to the following features in its struct:

    - st_dev /* An ID of device containing file */
    - st_ino /* inode number */
    - st_mode /* protection */
    - st_nlink /* number of hard links */
    - st_uid /* user ID of owner */
    - st_uid /* group ID of owner */

  And much more. See man 2 stat for further description.

- S_ISREG(m) : A macro used to interpret the values in a stat-struct, as return from the system call stat, which evaluates true if the argument (the st_mode member in struct stat) is a regular file. However, one must first ensure to set stat with its appropriate arguments for correct usage. S_ISREG argument m is represented as struct stat buf, which when passed as an argument, buf.st_mode (which is in reference to POSIX: bits corresponding to S_INFMT - masked values that are defined for file types e.g 0170000, essentially a bit mask for the file type bit field). In our definitions dog this feature simply writes out the contents of a valid file.

- S_ISDIR(m) : A macro used to interpret the values in a stat-struct, as return from the system call stat, which evaluates true if the argument (the st_mode member in struct stat) is a directory. However, one must first ensure to set stat with its appropriate arguments for correct usage. S_ISDIR argument m is represented as struct stat buf, which when passed as an argument, buf.st_mode (which is in reference to POSIX: bits corresponding to S_IFDIR - masked values that are defined for file types e.g 0040000, essentially a bit mask for the file type bit field).

- Memset(void *ptr, provided value, size) : used to fill a block of memory with a particular value. Where the starting pointer to the address of memory is to be filled with the provided value, given the size. Memset was declared in our dog definitions to re-initiate the state of our buffer; setting all bits to zero by using '\0' for the second argument. The third argument is simply the size for this buffer in question; 4Kib as stated in requirements.

- strcmp(const char * str1, const char *str2): Takes two string arguments, compares them, if equal, returns integer value 0 indicating equivalence. In our driver program dog.cpp, this feature was implemented as the first argument corresponds to the file arguments, and the second argument a string dash argument.

- strstr(const char *haystack, const char *needle): finds first occurrence of the substring needle in the string haystack. The terminating '\0' characters are not compared however. The first argument in our driver program dog.cpp is provided as an argument to ./dog, and the second argument is a string dash. The strstr is then assigned to dash for later future comparisons with future file arguments.

- user_input_output(int fd, const char *[]): Takes the file descriptor and argv files as arguments, prompts user for input via system standard input (0), and outputs standard output (1) upon enter. Exit’s out of loop when the user inputs ctrl+d.

- dash_input_output(): Prompts user for input via system standard input (0), and outputs standard output (1) upon enter. Exit’s out of loop when the user inputs ctrl+d.

- check_format: Checks if of file type (and or exist) or directory. Outputs depend on type; note, warn is used to output the error message that corresponds to the file in question.

- check_warn: Signals an interruption via warn; example would state something along the lines of not found.

Pseudocode

```
procedure Dog.cpp
    include “dog_definition.cpp” file

    declare fd
    fd <- open(dog.cpp,read only)
    check_warn(fd,argv)
    close(fd)

      if argc=1 then
          fd <- open(dog.cpp, read or write)
          user_input_output(fd,argv)
      else
        close(fd)
        declare char pointer dash
        declare bool flag = false

        for i 1 to argc do
            if stringCompare(“-” = argv[i]) then
                dash = “-”
                if argv[i] == dash and flag = false then
                    dash_input_output()
```

```
                         flag = true
                    else
                         fd <- open(argv[i], read only)
                         check_format(fd,argv,i)
return 0
```
end procedure

procedure dog_definition.cpp
```
define MAX_BUFFER 4Kib
define Enter 0x0A
define STD_IN 0
define STD_OUT 1

/* Module box 1*/
void function user_input_output(fd,argv)
          declare buffer[MAX_BUFFER]
          Loop (std_in read buffer until EOF):
                  if fd = -1 then
                     warn("%s",argv[0]);
                      Exit
                  if "Enter" then
                      write(stdin,buffer,read(stdout,buffer,size of buffer max))
                      memset(reset buffer and fill with '\0')
                  close(fd);
                  Exit


/* Module box of 1 minor config */
void function dash_input_output()
          declare buffer[MAX_BUFFER]
          Loop (std_in read buffer until EOF):
                  if fd = -1 then
                       warn("%s",argv[0]);
                        Exit;
                   if "Enter" then
                        write(stdin,buffer,read(stdout, buffer, buffer max size))
                        memset(reset buffer and fill with '\0')
```

```
/* Checks for file or directory */
void function check_format(fd,argv,i)
        declare buffer[MAX_BUFFER]
        if fd = -1 then
            fprintf(stderr, ”strings”,argv[i], strerror(errno))
            close(fd)

        declare struct stat buf
        stat(argv[i],&buf)

        if regularFile(buf.st_mode) then
            write(stdout, buffer, read(fd,buffer,max buffer size))
            close(fd)

        if isDirectory(buf.st_mode) then
            fprintf(stderr,”strings”,argv[i],strerror(errno))
            close(fd)

/* file interruption */
void function check_warn(fd,argv)
      if fd = -1 then
          warn(“%s”,argv[0]);
          Exit
```

end procedure

procedure dog_header.h

```
Include guard

declare function user_input_output(integer field descriptor, const char *argv[])
declare function dash_input_output()
declare check_format(integer field descriptor, const char *argv[], int of current file arg)
declare check_warn(int field descriptor, const char *argv[])

End include guard
```

end procedure