*The Concatenate Program*

In theory, the `cat(1)` program takes zero to $n$ arguments which are interpreted as file names and concatenates their contents, printing the result to the console:

```
$ cat foo.txt bar.txt baz.txt
I'm a line from foo.txt
I'm two lines ...
... from bar.txt
And another line from baz.txt
```

Our simplified version of cat will not take any program arguments. In that case, both your and the original program, will read the consoles interactive text input, and 'echo' it back to the user. To get a feel of what we mean, try to replicate the given session.

```
$ cat
I write a line and press enter
I write a line and press enter
The first line was my input, the second by cat(1)
The first line was my input, the second by cat(1)
I can also write a line and press Ctrl-DI can also write
a line and press Ctrl-DThis way, no line break is issued
This way, no line break is issued
To terminate input, we need to end the 'input file'
To terminate input, we need to end the 'input file'
We do that by pressing Ctrl-D twice
```

**Listing 1** Interactive Input/Output with `cat(1)`

UNIX/Linux programs see interactive console input as one 'file'. What the `cat(1)` program does is simply read a chunk of that file, write that to the console, read another chunk, and so on.

If you type input in the console it will not be *written* to that 'file' immediately though, but only after you 'flush the buffer'. This is done by sending the special ASCII symbol `EOT` (End-Of-Transmission) which can be entered by typing `Ctrl-D`. Alternatively, you can press the `ENTER` key which will first append a newline (`'\n'`)

at the end of the input and then send `EOT`. The effect can be observed in the **Listing 1**.

In order for `cat(1)` to terminate, we need to end the file. A file has ended, when there's 'no more data to read'. We can signal this condition, also called `EOF` (End-Of-File) by flushing the buffer when there's been no input. This may be directly after program startup, but also, after just having flushed the buffer. Thus, sending `EOT` (`Ctrl-D`) twice does just that.

I heavily recommend playing around with the original `cat(1)` program a bit as this behavior is better seen than told.

*UNIX File I/O—in C*

Not only is the interactive console input a file, so is the console output. These files are called 'standard input' and 'standard output' respectively.

While a file on disk is identified by its name, in order to read or write from or to it, a file needs to be opened by the program. After that, the program can refer to each file by an identifying number, called file descriptor (typefd). Luckily for us, both standard in- and output are open on program startup with their fds being being 0 and 1 respectively.

Let's write a function that copies from a given `fd` to another.

```
int copy(int srcfd, int destfd)
{
        //TODO
        return (0);
}

int main(void)
{
        int err = copy(0, 1); // copy stdin to stdout
        return (err);
}
```

Since the input file may be *really* big we cannot copy it in one session but copy it 1024 B at a time. For that we need to have a temporary storage location of that size. We will use an array of 1024 elements of `char` since a `char` is exactly 1 B.

```
int copy(int srcfd, int destfd)
{
        char buffer[1024];

        // TODO
        return (0);
}
```

We call this variable 'buffer' (a different one from the mentioned above with `Ctrl-D`, they are really everywhere!) and can now access any byte in it from the first `buffer[0]` to the last `buffer[1023]`.

In order to actually *read* the input we need to execute a system call. Again, those who've taken Computer Architecture before will remember that these don't work the same way as regular function `call`s. However, the UNIX standard library header `unistd.h` provides us with friendly wrappers which make them feel like regular functions.

```
#include <unistd.h>

int copy(int srcfd, int destfd)
{
        char buffer[1024];

        ssize_t nread = read(srcfd, buffer, 1024);
        // TODO
        return (0);
}
```

This will read *up to* 1024 B from the input fd into the buffer. It will *also* return the number of bytes *actually read*, which is assigned to the variable `nread`. This result may be less than 1024 B, if there aren't that many bytes available. The special case of 0, as already mentioned earlier, indicates EOF and our `copy` function should end itself in that case:

```
ssize_t nread = /* ... */
if (nread == 0) {
        return (0);
}
```

Without going into too much detail, the type `ssize_t` is determined by the result type of `read` whicyh you can look up from the documentation at **`read(3p)`**, or on the console:

```
$ man 2 read    # or, if posix-man installed:
$ man 3p read
```

The next step is to write those `nread` many bytes to `destfd` which works similarly, with the documentation being at **`write(3p)`**

```
int copy(int srcfd, int destfd)
{
        char buffer[1024];

        ssize_t nread = read(srcfd, buffer, 1024);
        if (nread == 0) { /* EOF */
            return (0);
        }
        write(destfd, buffer, nread);

        return (0);
}
```

This will execute once only though. Try it!

In order to make history repeat itself, we wrap the everything from the `read` to the `write` into an endless loop:

```
while (1) {
        /* ... */
}
```

It's not really endless though, since, if the `read` function returns 0 we break out of the loop and terminate the function anyway! This seems to work now!

Unfortunately, we are not completely done as the `write` function has a small caveat: Even if we ask it to write `nread` many bytes—it may write less (although, at least 1 B). We might end up with reading 500 B and then writing just 200 B!

It does return how many bytes it *did actually write*. With this information, we can ask it to write out the remaining 300 B. We do that by keeping track of how many bytes we've written in total. This is needed as the `write` function doesn't remember were it left off and we need to tell it not to write out those 200 B again, but to skip those. In pseudocode this would look like:

```
nread = read(...)
if nread == 0:
    return (0)

nwritten = 0
while nwritten < nread:
    nwritten' = write(destfd, buffer+nwritten,
                      nread-nwritten)
    nwritten = nwritten + nwritten'
```

*Program Arguments*

We've now reached feature parity with the assembly version of our `cat(1)` program. However, the actual `cat(1)` program doesn't read just from standard input but also allows for program arguments supplying file names of files which are subsequently printed to the console output.

The following provides a skeleton for what we want to do in our `main` function:

```
int main(int argc, char *argv[])
{
        if (argc == 1) { /* no actual arguments */
                int err = copy(0, 1);
                return (err);
        }

        int status = 0;
        for (int i = 1; i < argc; i++) {
                //TODO:  "create" srcfd

                int err = copy(srcfd, 1);
                if (err) {
                        status = 1;
                        // break out of the loop
                        break;
                }
        }
        return (status);
}
```

What's left is assigning each file name a file descriptor as discussed earlier. This process is called 'opening' a file, which is done using the **open(3p)** function:

```
// open the file with name `argv[i]' with the
// read-only option:
int srcfd = open(argv[i], O_RDONLY);
```

This function assigns an ID (the `fd`) to refer to that file which we now can use for our `copy` function. After using it, we should however **close(3p)** the file descriptor again, in order to not waste IDs.

*Error Handling*

There are many open questions in our program now, however. Things like, what happens if we pass file names which don't refer

to valid files or files at all? Or to files we don't have the permission to read? In that case, the `open` function will return $-1$ as specified by the 'RETURN VALUE' section in its manual. Instead of passing it to our `copy` function, we should check for this special value and print an error message:

```
#include <stdio.h>

// ...

if (srcfd == -1) {
        // print an error, e.g.:
        // /etc/shadow: Permission denied
        perror(argv[i]);

        // Denote unsuccessful run, but don't break out
        // yet, just continue with the next file and
        // skip the current.
        status = 1;
        continue;
}
```

The **`perror(3p)`** function is quite handy here, as while the return value of $-1$ only denotes that the function failed, `open` has also set the variable `errno` to a special value denoting *how* it has failed. The `perror` function interprets this value and prints the respective error on the console, prefixed with whatever string we passed it (in our case, the file name), followed by a colon `:` and space.

If `open` fails, we simply want to carry on with the next file after printing the error, while 'remembering' that something went wrong, s.t., we can actually set the exit code (return value of `main`) appropriately. We do that by assigning 1 to a variable we've created and returning this variable from main, once we've finished copying all other files to the standard output.

Also note that the `read` and `write` functions can fail as well (although this is less likely). But it's better to check for these errors since otherwise debugging a program is really difficult.