

MyStream

Deadline: September 19th, 2014

1 Instructions

For this assignment you must:

1. define your own wrappers around the system calls `open`, `close`, `read`, `write`, `lseek`.
2. implement a buffered I/O library.

In this assignment, you may use the standard C functions `malloc`, `realloc` and `free`; as well as functions from previous assignments (by including their source in your submission). All other external functions are forbidden.

Each function to implement must be defined in its own `.c` file, named after the function. You must provide a suitable `Makefile` which compiles all the functions and places them in `libminic.a`.

2 System calls

1. You must implement the following functions:

```
int my_open(const char *path, int oflag, int mode);
int my_close(int fd);
long my_read(int fd, void *buf, unsigned long nbyte);
long my_write(int fd, const void *buf, unsigned long nbyte);
```

Each function must contain inline assembly to perform a system call to the operating system. You must support at least one of the following combinations: Linux/i386, Linux/x86-64, FreeBSD/i386 or FreeBSD/amd64.

You can easily find information about the syscall interface of Linux and FreeBSD online. Some example links are provided in [References](#) below.

2. You must place their prototype in a header named `mysys.h`, together with suitable definitions for the preprocessor macros `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`.

NB: Checks for adherence to the C coding standard for the 4 source files containing the definition of these functions will be relaxed, in particular regarding compiler warnings. Ensure that you properly document violations using comments in your source code.

3 Libstream

You must define your own type `st_t` in a header named `mystream.h`, based on a preprocessor macro `ST_BUFFER_SIZE`:

```
#ifndef MYSTREAM_H
#define MYSTREAM_H
#ifndef ST_BUFFER_SIZE
#define ST_BUFFER_SIZE 8192
#endif
typedef struct st
{
    // Your definitions here...
} st_t;
// Functions declarations may follow here.
#endif
```

You must then implement the following functions:

```
st_t *st_open(const char *, const char *);
void st_close(st_t *);
unsigned st_read(st_t *, void *, unsigned);
unsigned st_write(st_t *, const void *, unsigned);
void st_flush(st_t *);
int st_gets(st_t *, char *, unsigned);
int st_puts(st_t *, char *);
int st_putchar(st_t *, int);
int st_getchar(st_t *);
char *st_getline(st_t *);
```

3.1 Function semantics

The purpose of this library is to implement buffered I/O:

st_open(path, mode) Open the file identified by `path` and return an `st_t`. `mode` may be "r" (open for reading), "w" (open for writing), "r+" (open for reading and writing). Return a freshly allocated `st_t` object, or 0 if an error occurs.

st_close(st) Flush the buffer(s) and close the stream. The `st` object must also be deallocated.

st_write(st, buf, sz) Write `sz` bytes from the buffer `buf` to the stream identified by `st`. Return the number of bytes that were written/buffered successfully, or 0 if no bytes could be written/buffered.

st_read(st, buf, sz) Read up to a maximum of `sz` bytes from the stream identified by `st` onto `buf`. Return the number of bytes that were read successfully, or 0 if no bytes could be read.

st_flush(st) Flush the buffer(s) associated with `st`.

st_putchar(st, c) Write the character `c` to the stream `st`. Return 1 if the character could be written/buffered, 0 otherwise.

st_getchar(st) Read a character from `st`. Return the character value (between 0 and 255 inclusive), or -1 if no character could be read.

`st_puts(st, str)` Write the nul-terminated string identified by `str` onto the stream identified by `st`. Return the number of characters that were written/buffered successfully, or 0 if no characters could be written/buffered.

`st_gets(st, str, sz)` Read at most one less than the number of characters specified by `sz` from the stream `st` and store them into `str`. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a nul byte is appended to end the string. Return the number of characters read (including the newline character, if any, but excluding the appended nul byte), or 0 if no characters could be read.

`st_getline(st)` Read a character string from `st` into a freshly allocated heap object. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is read from the stream but not written to the output string. If any characters are read, a nul byte is appended to end the string. Return the freshly allocated string, or 0 if no characters could be read.

4 Optional extra features

You may implement the following for a higher grade:

- `lseek` system call:

```
// Argument for whence:
# define SEEK_SET 0
# define SEEK_CUR 1
# define SEEK_END 2

long my_lseek(int fd, long offset, int whence);
```

- a valid preprocessor macro definition for `O_APPEND` in `mysys.h`;
- the following additional libstream function:

```
long st_lseek(st_t *, long offset, int whence);
```

which calls `my_lseek` for the underlying file descriptor and handles buffering appropriately.

- modes "a" and "a+" for `st_open`, which open the stream using modes "append-only" and "read/append".
- a global preprocessor macro called `ERRNO` that can be used to retrieve the most recent error, like `errno` in C.

5 Grading

- 0.25 point per system call correctly implemented in the mandatory list (1 points in total).
- 0.5 points per `st_` function correctly implemented in the mandatory list (5 points in total).

- +1 if the library supports changing the direction on streams opened read/write (from reading to writing or vice-versa) without invoking `st_flush` in between.
- +0.5 if `my_lseek` and `st_lseek` are properly implemented.
- +0.5 if `O_APPEND` is properly defined and `st_open` supports modes "a" and "a+".
- +0.5 if the syscall wrappers support more than one operating system / platform combination.
- +0.5 if the library works properly with different values of `ST_BUFFER_SIZE`.
- +1 if `ERRNO` is implemented properly without using a global variable.

6 References

Details about the syscall interfaces for Linux and FreeBSD:

- <http://man7.org/linux/man-pages/man2/syscall.2.html>
- <http://www.freebsd.org/doc/en/books/developers-handbook/x86-system-calls.html>
- <http://www.int80h.org/bsdasm/#system-calls>

System call numbers:

```

#if defined(__FreeBSD__)
# define O_RDONLY      0x0000
# define O_WRONLY      0x0001
# define O_RDWR        0x0002
# define O_APPEND      0x0008
# define O_CREAT        0x0200
# define O_TRUNC        0x0400
# define SYS_read       3
# define SYS_write      4
# define SYS_open       5
# define SYS_close      6
# define SYS_lseek      478
#elif defined(__linux__)
# define O_RDONLY      00
# define O_WRONLY      01
# define O_RDWR        02
# define O_CREAT        0100
# define O_TRUNC        01000
# define O_APPEND      02000
# if defined(__i386__)
# define SYS_read       3
# define SYS_write      4
# define SYS_open       5
# define SYS_close      6
# define SYS_lseek      19
# elif defined(__x86_64)
# define SYS_read       0
# define SYS_write      1
# define SYS_open       2
# define SYS_close      3
# define SYS_lseek      8
# endif
#endif

```

7 Copyright and licensing

Copyright © 2014, Raphael 'kena' Poss. Permission is granted to distribute, reuse and modify this document and other documents for the Systems Programming course by the same author according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>.