

CSE 333 Section 3 - POSIX I/O Functions

Welcome back to section! We're glad that you're here :)

POSIX

Posix is a family of standards specified by the IEEE. These standards maintains compatibility across variants of Unix-like operating systems by defining APIs and standards for basic I/O (file, terminal, and network) and for threading.

1) What does POSIX stand for?

Portable Operating System Interface

2) Why might a POSIX standard be beneficial? From an application perspective? Versus using the C stdio library?

List of answers:

- **More explicit control since read and write functions are system calls and you can directly access system resources.**
- **POSIX calls are unbuffered so you can implement your own buffer strategy on top of read()/write().**
- **There is no standard higher level API for network and other I/O devices**

POSIX and Files

File I/O using POSIX is very similar to file I/O using the a C stdio library. Some of operations that can be performed on files using Posix systems calls are: opening a file, reading from a file, writing to a file, closing a file.

```
int open(char* filename, int flags, mode_t mode);
```

→ *filename is a string representing the name of the file.*

→ *flags is an integer code describing the access. Some common flags are listed below:*

- ◆ *O_RDONLY - Open the file in read-only mode.*
- ◆ *O_WRONLY - Open the file in write-only mode.*
- ◆ *O_RDWR - Open the file in read-write mode.*
- ◆ *O_APPEND - Append new information to the end of the file.*

★ *Returns an integer which is the file descriptor. Returns -1 if there is a failure.*

```
int close(int fd);
```

→ *fd is the file descriptor (as returned by open()).*

★ *Returns 0 on success, -1 on failure.*

```
size_t read(int fd, void *buf, size_t count);
```

```
size_t write(int fd, const void *buf, size_t count);
```

→ *fd is the file descriptor (as returned by open()).*

→ *buf is the address of a memory area into which the data is read or written.*

→ *count is the maximum amount of data to read from or write to the stream.*

★ *Returns the actual amount of data read from or written to the file.*

Exercises:

- 3) A common use of the POSIX I/O function is to **write** to a file; fill in the code skeleton below that writes all of the contents of a string `buf` to the file `333.txt`. *You must use a different method than the "bytes_left" method shown in lecture.*

```
int fd = open("333.txt", O_WRONLY); // open 333.txt
char *buf = ...; // assume buffer has size n
int result;

char *ptr = buf; // initialize variable for loop

... // code that populates buf happens here

while (ptr < buf + n) {
    result = write(fd, ptr, buf + n - ptr);

    if (result == -1) {
        if (errno != EINTR) {
            ... // a real error happened, return an error result
        }
        continue; // EINTR happened, so loop around and try again
    }
    ptr += result; // update loop variable
}
```

- 4) Why is it important to store the return value from the `write()` function? Why do we not check for a return value of 0 like we do for `read()`?

**write() may not actually write all the bytes specified in count.
Writing adds length to your file, so you don't need to check for end of file.**

- 5) Why is it important to remember to call the `close()` function once you have finished working on a file?

In order to free resources i.e. other processes can acquire locks on those files.

POSIX and Errors

Unfortunately, errors that occur when using POSIX system calls are not handled for the user as they are with C standard library functions. So it is important thing is to make sure your code handles errors gracefully. When an error occurs, the error number is stored in `errno`, which is defined under `<errno.h>`. You can use `perror()` to print out a message based on `errno`. Remember that `errno` is shared by all library functions and overwritten frequently, so you must read it *right* after an error to be sure of getting the right code.

POSIX functions have a variety of error codes to represent different errors. Some common error conditions:

- ◆ `EBADF` - `fd` is not a valid file descriptor or is not open for reading.
- ◆ `EFAULT` - `buf` is outside your accessible address space.
- ◆ `EINTR` - The call was interrupted by a signal before any data was read.
- ◆ `EISDIR` - `fd` refers to a directory.

Exercise:

6) Given the name of a file as a command-line argument, write a C program that is analogous to `cat`, *i.e.* one that prints the contents of the file to `stdout`. Handle any errors!

```
int main(int argc, char** argv) {
    /* 1. Check to make sure we have a valid command line arguments */

    /* 2. Open the file, use O_RDONLY flag */

    /* 3. Read from the file and write it to standard out. Try doing
       this without using printf() and instead have write() pipe to
       Stdout. It might be helpful to initialize a buffer variable
       (of size 1024 bytes should be fine) to pass in to read() and
       write().

    /*4. Clean up */
}
```

See `filedump.c` on course website.

POSIX and directories

POSIX calls can also be used to access directories. This is because in linux directories are nothing more than special files. An example workflow might be: open a directory, iterate through directory contents, close the directory.

```
DIR *opendir(const char* name);
```

→ name *is the directory to open.*

★ Returns a pointer DIR* to the directory stream or NULL on error (with errno set).

```
int closedir(DIR *dirp);
```

→ dirp *is the directory stream to close.*

★ Returns 0 on success or -1 on error (with errno set).

```
struct dirent *readdir(DIR *dirp);
```

→ dirp *is the directory stream to process.*

★ Returns a pointer to a dirent structure representing the next directory entry in the directory stream or returns NULL on error or reaching the end of the directory stream.

On Linux, the dirent structure is defined as follows:

```
struct dirent {
    ino_t      d_ino;      /* inode number for the dir entry */
    off_t      d_off;      /* not necessarily an offset */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file (not what you think); not
                           supported by all file system types */
    char        d_name[NAME_MAX+1]; /* directory entry name*/
};
```

Exercise:

7) Given the name of a directory, write a C program that is analogous to `ls`, i.e. prints the names of the entries of the directory to `stdout`. Handle any errors!

```
int main(int argc, char** argv) {
    /* 1. Check to make sure we have a valid command line arguments */

    /* 2. Open the directory, look at opendir() */

    /* 3. Read through/parse the directory and print out file names
       Look at readdir() and struct dirent */

    /* 4. Clean up */
}
```

See `dirdump.c` on course website.