

# CSE 333 Section 3 Solutions - POSIX I/O Functions

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

## POSIX and Files

POSIX has similar file I/O operations as the C stdio library, but unbuffered by default, including:

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

- *name* is a string representing the name of the file. Can be relative or absolute.
- *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.

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

```
ssize_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.

## POSIX and Errors

Unfortunately, errors are not handled as nicely for the user as they are in the C stdio library. So it is important to make sure your code handles errors gracefully. Note that:

- When an error occurs, the error number is stored in `errno` (defined in `<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.
- ◆ `EAGAIN` - *fd* refers to a file other than a socket and has been marked nonblocking, and the read/write blocks.
- ◆ `EISDIR` - *fd* refers to a directory.

`EAGAIN` and `EINTR` are recoverable errors, unlike the rest.

## 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. Accepts relative and absolute paths. Can end with '/', but is not necessary.*

★ 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 */
};
```

### Exercises:

1) 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**

2) 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.*

**// \*\*NOTE: This is one way to solve this exercise.**

**// There exist other correct solutions to this exercise.**

```
int fd = open("333.txt", O_WRONLY); // open 333.txt
int n = ....;
char *buf = ..... ; // Assume buf initialized with 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 && errno != EAGAIN) {
            // a real error happened, return an error result
            close(fd); // cleanup
            perror("Write failed");
            return -1;
        }
        continue; // EINTR or EAGAIN happened, so loop around and try
again
    }
    ptr += result; // update loop variable
}
close(fd); // cleanup
```

3) Why is it important to store the return value from the `write()` function? Why don't we check for a return value of 0 like we do for `read()`?

**write() may not actually write all the bytes specified in count.**

**The 0 case for reading was EOF, but writing adds length to your file and we know exactly how much we are trying to write.**

4) 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.**

**Exercise:**

- 5) 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`. Be sure to handle any errors!  
Example usage: `./dirdump <path>` where `<path>` can be absolute or relative.

```
int main(int argc, char** argv) {
    /* 1. Check to make sure we have valid command line arguments */
    if (argc != 2) {
        fprintf(stderr, "Usage: ./dirdump <path>\n");
        return EXIT_FAILURE;
    }

    /* 2. Open the directory, look at opendir() */
    DIR* dirp = opendir(argv[1]);
    if (dirp == NULL) {
        fprintf(stderr, "Could not open directory\n");
        return EXIT_FAILURE;
    }

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

    entry = readdir(dirp);
    while (entry != NULL) {
        printf("%s\n", entry->d_name);
        entry = readdir(dirp);
    }

    /* 4. Clean up */
    closedir(dirp);
    return EXIT_SUCCESS;
}
```

### **Exercise (bonus)**

- 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 */
    if (argc != 2) {
        fprintf(stderr, "Usage: ./filedump <filename>\n");
        return EXIT_FAILURE;
    }
    /* 2. Open the file, use O_RDONLY flag */
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "Could not open file for reading\n");
        return EXIT_FAILURE;
    }
    /* 3. Read from the file and write it to standard out.*/
    char buf[SIZE];
    ssize_t len;
    do {
        len = read(fd, buf, SIZE);
        if (len == -1) {
            if (errno != EINTR && errno != EAGAIN) {
                close(fd);
                perror(NULL);
                return EXIT_FAILURE;
            }
            continue;
        }
        size_t total = 0;
        ssize_t wlen;
        while (total < len) {
            wlen = write(1, buf + total, len - total);
            if (wlen == -1) {
                if (errno != EINTR && errno != EAGAIN) {
                    close(fd);
                    perror(NULL);
                    return EXIT_FAILURE;
                }
                continue;
            }
            total += wlen;
        }
    } while (len != 0);
    /*4. Clean up */
    close(fd);
    return EXIT_SUCCESS;
}
```