File I/O

CS 3113 Fall 2018

Be sure to install all code prerequisites

```bash
sudo apt-get update
sudo apt-get install -y vim emacs htop tmux tree
sudo apt-get install -y gcc gdb make valgrind strace
sudo apt-get install -y glancer ranger
sudo apt-get install -y linux-tools-common linux-tools-generic
sudo apt-get install -y linux-tools-`uname -r`
sudo apt-get install -y libcap-dev
sudo apt-get install -y libacl1-dev
```

CG and AHF: Introduction to Operating Systems

Download TLPI book code

```bash
cd /projects
wget http://man7.org/tlpi/code/download/tlpi-180725-dist.tar.gz
tar xvzf tlpi-180725-dist.tar.gz
cd tlpi-dist/
make
```

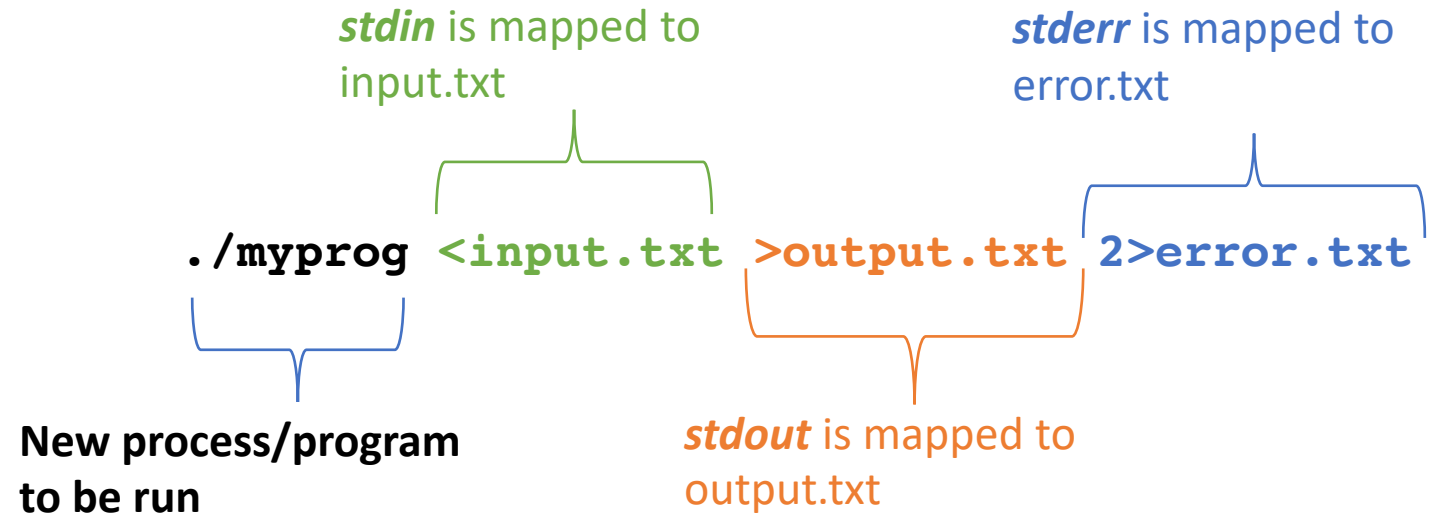# File Descriptors

| A <u>nonnegative integer</u> that may refer to:
  regular files, pipes, FIFOs, sockets, terminals or devices.

| Each process has **its own assigned set** of file descriptors.

| Used by the system to refer to files (not filenames)

| When requested, the lowest-numbered unused file descriptor is assigned

Standard File Descriptors

| When a shell program is run, these descriptors are copied from the terminal to the running program.

| I/O redirection may modify this assignment.

| IDEs may map output to stderr to a red color

| POSIX names are available in `<unistd.h>`

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

stdin is mapped to input.txt

stderr is mapped to error.txt

`./myprog <input.txt >output.txt 2>error.txt`

New process/program to be run

stdout is mapped to output.txt

Key I/O System Calls

$fd = open(pathname, flags, mode)$ | opens the file identified by *pathname*, returning a file descriptor.

$numread = read(fd, buffer, count)$ | reads at most *count* bytes from the open file referred to by *fd* and stores them in *buffer*.

$numwritten = write(fd, buffer, count)$ | writes up to *count* bytes from *buffer* to the open file referred to by *fd*.

$status = close(fd)$ | is called after all I/O has been completed, in order to release the file descriptor *fd* and its associated kernel resources.

Example: fileio/copy.c

Listing 4-1: Using I/O system calls

```
———————————————————————————————————————————————— fileio/copy.c
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#ifndef BUF_SIZE           /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024
#endif

int
main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Open input and output files */

    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH;      /* rw-rw-rw- */
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);

    /* Transfer data until we encounter end of input or an error */

    while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
        if (write(outputFd, buf, numRead) != numRead)
            fatal("couldn't write whole buffer");
    if (numRead == -1)
        errExit("read");

    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
———————————————————————————————————————————————— fileio/copy.c
```

Universality of I/O

same four system calls—*open()*, *read()*, *write()*, and *close()*—are used to perform I/O on **all** types of files.

```
$ ./copy test test.old          Copy a regular file
$ ./copy a.txt /dev/tty         Copy a regular file to this terminal
$ ./copy /dev/tty b.txt         Copy input from this terminal to a regular file
$ ./copy /dev/pts/16 /dev/tty   Copy input from another terminal
```

Open

opens the file identified by *pathname*, returning a file descriptor.

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
                        Returns file descriptor on success, or −1 on error
```

Listing 4-2: Examples of the use of *open()*

```
/* Open existing file for reading */

fd = open("startup", O_RDONLY);
if (fd == -1)
    errExit("open");

/* Open new or existing file for reading and writing, truncating to zero
   bytes; file permissions read+write for owner, nothing for all others */

fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");

/* Open new or existing file for writing; writes should always
   append to end of file */

fd = open("w.log", O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
                    S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if *pathname* is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)	
O_NOCTTY	Don't let *pathname* become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3
O_SYNC	Make file writes synchronous	v3

Read

reads at most *count* bytes from the open file referred to by *fd* and stores them in *buffer*.

```c
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
                    Returns number of bytes read, 0 on EOF, or −1 on error
```

```c
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");
printf("The input data was: %s\n", buffer);
```

```c
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

Write

writes up to *count* bytes from *buffer* to the open file referred to by *fd*.

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or −1 on error

Close

is called after all I/O has been completed, in order to release the file descriptor fd and its associated kernel resources.

```
#include <unistd.h>

int close(int fd);
```

Returns 0 on success, or −1 on error

```
if (close(fd) == -1)
    errExit("close");
```
Always check for errors.

Seeking

File offset

| Also called *read- write offset* or *pointer*

| the kernel records a *file offset* for **each open file**.

| The first byte of the file is at offset 0.

| The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to *read()* or *write()*

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
                    Returns new file offset if successful, or −1 on error
```


Figure 4-1: Interpreting the *whence* argument of *lseek()*

```c
lseek(fd, 0, SEEK_CUR); /* Returns current cursor loc of without change */
lseek(fd, 0, SEEK_SET); /* Start of file */
lseek(fd, 0, SEEK_END); /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END); /* Last byte of file */
lseek(fd, -10, SEEK_CUR); /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END); /* 10001 bytes past last byte of file */
```
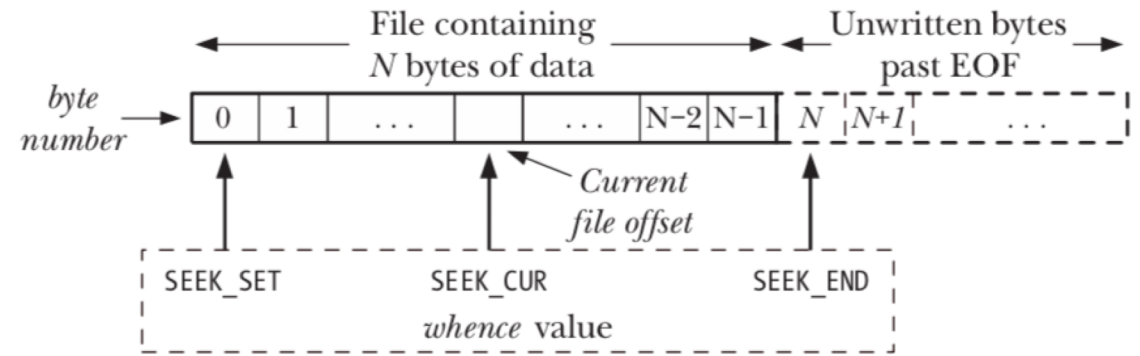
```
        if (argv[ap][0] == 'r')
            printf("%c", isprint((unsigned char) buf[j]) ?
                                        buf[j] : '?');
        else
            printf("%02x ", (unsigned int) buf[j]);
    }
    printf("\n");
}

free(buf);
break;

case 'w':    /* Write string at current offset */
    numWritten = write(fd, &argv[ap][1], strlen(&argv[ap][1]));
    if (numWritten == -1)
        errExit("write");
    ﾠf("%s: wrote %ld bytes\n", argv[ap], (long) numWri﹂

                        ﾠﾠoffset */
```

```
        ﾠation of read(), write(), an﹂
_____
ﾠude <sys/stat.h>
ﾠnclude <fcntl.h>
include <ctype.h>
include "tlpi_hdr.h"

ﾠt
ﾠin(int argc, char *argv[])
{
    size_t len;
    off_t offset;
    int fd, ap, j;
    char *buf;
    ssize_t numRead, numWritten;

    if (argc < 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s file {r<length>|R<length>|w<string>|s<offset>}...﹂
                    argv[0]);

    fd = open(argv[1], O_RDWR | O_CREAT,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH);              /* rw-rw-rw﹂
    if (fd == -1)
        errExit("open");

    for (ap = 2; ap < argc; ap++) {
        switch (argv[ap][0]) {
        case 'r':    /* Display bytes at current offset.﹂
        case 'R':    /* Display bytes at current offs﹂
            ﾠlen = getLong(&argv[ap][1], GN_ANY_﹂
```

Example: fileio/seek_io.c

lseek + read + write

```
$ touch tfile                              Create new, empty file
$ ./seek_io tfile s100000 wabc             Seek to offset 100,000, write "abc"
s100000: seek succeeded
wabc: wrote 3 bytes

 du tfile                       # The number of blocks used


$ ls -l tfile                              Check size of file
-rw-r--r--     1 mtk     users    100003 Feb 10 10:35 tfile
$ ./seek_io tfile s10000 R5                Seek to offset 10,000, read 5 bytes from hole
s10000: seek succeeded
R5: 00 00 00 00 00                         Bytes in the hole contain 0

 ./seek_io tfile s10000 wefg     # write efg starting at byte point 10000

 du tfile                        # The number of blocks used
```