Input/Output

Prof. Michele Loreti

Laboratorio di Sistemi Operativi

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

File descriptors...

Before a file can be read from or written to, it must be opened.

File descriptors...

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

File descriptors...

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File descriptors...

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File table is indexed via nonnegative integers known as file descriptors (often abbreviated fds).

File descriptors...

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File table is indexed via nonnegative integers known as file descriptors (often abbreviated fds).

Each entry in the list contains information about the file (permissions, location,...).

File descriptors...

Before a file can be read from or written to, it must be opened.

The kernel maintains a per-process list of open files, called the file table.

Definition:A process is a running program!

File table is indexed via nonnegative integers known as file descriptors (often abbreviated fds).

Each entry in the list contains information about the file (permissions, location,...).

File descriptors are obtained when a file is opened, and used to perform file operations..

File descriptors...

File descriptors are represented by the *C* `int` type.

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;
- by default this max value is `1024`, however can be increased up to `1048576`;

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;
- by default this max value is `1024`, however can be increased up to `1048576`;
- -1 is used to indicate an error.

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from `0` and go up to one less than this maximum value;
- by default this max value is `1024`, however can be increased up to `1048576`;
- -1 is used to indicate an error.

File descriptors...

File descriptors are represented by the *C* `int` type.

Each Linux process has a maximum number of files that it may open:

- start from 0 and go up to one less than this maximum value;
- by default this max value is 1024, however can be increased up to 1048576;
- -1 is used to indicate an error.

Each process has at least three file descriptors:

- standard input: 0 (STDIN_FILENO);
- standard output: 1 (STDOUT_FILENO);
- standard error: 2 (STDERR_FILENO);.

Opening files...

A file is opened and a file descriptor is obtained with the open() system call:

Opening files...

A file is opened and a file descriptor is obtained with the open() system call:

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

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

Opening files...

A file is opened and a file descriptor is obtained with the open() system call:

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

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

Example:

```
int fd;
fd = open( "/home/piton/potions" , O_RDONLY );
if (fd<0) {
  // Error!
}
```

Opening flags...

flags argument may be bitwise-ORed with zero or more of the following values, modifying the behavior of the open request.

Opening flags...

flags argument may be bitwise-ORed with zero or more of the following values, modifying the behavior of the open request.

File status flags fall into three categories:

- **File Access Modes**, specify what type of access is allowed to the file (reading, writing, or both).

Opening flags...

flags argument may be bitwise-ORed with zero or more of the following values, modifying the behavior of the open request.

File status flags fall into three categories:

- **File Access Modes**, specify what type of access is allowed to the file (reading, writing, or both).
- **Open-time Flags**, control details of what open will do.

Opening flags...

flags argument may be bitwise-ORed with zero or more of the following values, modifying the behavior of the open request.

File status flags fall into three categories:

- **File Access Modes**, specify what type of access is allowed to the file (reading, writing, or both).
- **Open-time Flags**, control details of what open will do.
- **I/O Operating Modes**, affect how operations such as read and write are done.

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

- O_RDONLY — Open the file for read access.

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

- O_RDONLY — Open the file for read access.
- O_WRONLY — Open the file for write access.

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

- O_RDONLY — Open the file for read access.
- O_WRONLY — Open the file for write access.
- O_RDWR — Open the file for both reading and writing.

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

- O_RDONLY — Open the file for read access.
- O_WRONLY — Open the file for write access.
- O_RDWR — Open the file for both reading and writing.

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

- O_RDONLY — Open the file for read access.
- O_WRONLY — Open the file for write access.
- O_RDWR — Open the file for both reading and writing.

Open-time Flags: The open-time flags specify options affecting how open will behave.

Opening flags...

File Access Modes: The file access modes allow a file descriptor to be used for reading, writing, or both:

- O_RDONLY — Open the file for read access.
- O_WRONLY — Open the file for write access.
- O_RDWR — Open the file for both reading and writing.

Open-time Flags: The open-time flags specify options affecting how open will behave. There are two sorts of options specified by open-time flags:

- File name translation flags affect how open looks up the file name to locate the file, and whether the file can be created.
- Open-time action flags specify extra operations that open will perform on the file once it is open.

Opening flags...

Open-time Flags

Name translation flags:

- O_CREAT: The file will be created if it doesn't already exist.

Opening flags...

Open-time Flags

Name translation flags:

- O_CREAT: The file will be created if it doesn't already exist.
- O_EXCL: Is used in combination with O_CREAT and let open fails if the specified file already exists.

Opening flags...

Open-time Flags

Name translation flags:

- O_CREAT: The file will be created if it doesn't already exist.
- O_EXCL: Is used in combination with O_CREAT and let open fails if the specified file already exists.
- O_NONBLOCK: Prevents open from blocking for a "long time" to open the file (used for devices such as serial ports).

Opening flags...

Open-time Flags

Name translation flags:

- O_CREAT: The file will be created if it doesn't already exist.
- O_EXCL: Is used in combination with O_CREAT and let open fails if the specified file already exists.
- O_NONBLOCK: Prevents open from blocking for a "long time" to open the file (used for devices such as serial ports).
- O_NOCTTY: If the named file is a terminal device, don't make it the controlling terminal for the process.

Opening flags...

Open-time Flags

Name translation flags:

- O_CREAT: The file will be created if it doesn't already exist.
- O_EXCL: Is used in combination with O_CREAT and let open fails if the specified file already exists.
- O_NONBLOCK: Prevents open from blocking for a "long time" to open the file (used for devices such as serial ports).
- O_NOCTTY: If the named file is a terminal device, don't make it the controlling terminal for the process.

Opening flags...

Open-time Flags

Name translation flags:

- O_CREAT: The file will be created if it doesn't already exist.
- O_EXCL: Is used in combination with O_CREAT and let open fails if the specified file already exists.
- O_NONBLOCK: Prevents open from blocking for a "long time" to open the file (used for devices such as serial ports).
- O_NOCTTY: If the named file is a terminal device, don't make it the controlling terminal for the process.

Open-time action flags:

- O_TRUNC: Truncate the file to zero length (This option is only useful for regular files).

Opening modes...

Parameter mode provides the permissions of the newly created file.

Opening modes...

Parameter mode provides the permissions of the newly created file.

It is composed by an octal value with three digits represeting:

- User permission;
- Group permission;
- Others permiossion.

Opening modes...

Parameter mode provides the permissions of the newly created file.

It is composed by an octal value with three digits represeting:

- User permission;
- Group permission;
- Others permiossion.

Each digit consists of three bits *rwx* indicating *read*, *write* and *exec* permissions.

Example...

```c
int fd;
fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, 0664);
if (fd == -1) {
  /* error */
}
```

Creating a file...

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

Creating a file...

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

Creating a file...

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

This is not a typo!

Creating a file...

The combination of O_WRONLY | O_CREAT | O_TRUNC is so common that a system call exists to provide just that behaviour:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

This is not a typo!

```
int fd;
fd = creat (filename, 0644);
if (fd == 1) {
  /* error */
}
```

Reading from files:

The most basic mechanism used for reading is the `read()` system call:

Reading from files:

The most basic mechanism used for reading is the read() system call:

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

Reading from files:

The most basic mechanism used for reading is the read() system call:

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

ssize_t is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to size_t, but must be a signed type.

read(): Return Values

System call read returns the number of bytes that are read from the file:

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is -1, there is an error, a code is store in variable errno:

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is -1, there is an error, a code is store in variable errno:
  - EINTR (operation has been suspended, and it can be reissued);

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is -1, there is an error, a code is store in variable errno:
  - EINTR (operation has been suspended, and it can be reissued);
  - EAGAIN (no data is available, operation should be reissued later);

read(): Return Values

System call read returns the number of bytes that are read from the file:

```
ret = read( fd , buf , len )
```

- ret is equal to len;
- ret is less than len;
- ret is 0, end-of-file has been reached;
- ret is -1, there is an error, a code is store in variable errno:
  - EINTR (operation has been suspended, and it can be reissued);
  - EAGAIN (no data is available, operation should be reissued later);
  - ...

Example: reading all bytes

```
ssize_t ret;
while (len != 0 && (ret = read (fd, buf, len)) != 0) {
  if (ret==-1) {
    if (errno == EINTR)
      continue;
        perror ("read");
    break;
  }

  len -= ret;
  buf += ret;
}
```

Writing on files

The most basic and common system call used for writing is write():

Writing on files

The most basic and common system call used for writing is write():

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

Writing on files

The most basic and common system call used for writing is write():

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

A call to write() writes up to count bytes starting at buf to the current position of the file referenced by the file descriptor fd.

Writing on files

The most basic and common system call used for writing is write():

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

A call to write() writes up to count bytes starting at buf to the current position of the file referenced by the file descriptor fd.

Example:

```
const char *buf = "My ship is solid!";
ssize_t nr;
/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1) {
  /* error */
}
```

Append mode...

When fd is opened in append mode (via O_APPEND), writes do not occur at the file descriptor's current file position. Instead, they occur at the current end of the file.

Behaviour of write

When a call to write() returns, the kernel has copied the data from the supplied buffer into a kernel buffer, but there is no guarantee that the data has been written out to its intended destination.

Behaviour of write

When a call to write() returns, the kernel has copied the data from the supplied buffer into a kernel buffer, but there is no guarantee that the data has been written out to its intended destination.

When a user-space application issues a write() system call, the Linux kernel performs a few checks and then simply copies the data into a buffer.

Behaviour of write

When a call to write() returns, the kernel has copied the data from the supplied buffer into a kernel buffer, but there is no guarantee that the data has been written out to its intended destination.

When a user-space application issues a write() system call, the Linux kernel performs a few checks and then simply copies the data into a buffer.

Later, in the background, the kernel gathers up all of the dirty buffers, which are buffers that contain data newer than what is on disk, sorts them optimally, and writes them out to disk (a process known as writeback).

Behaviour of write

When a call to write() returns, the kernel has copied the data from the supplied buffer into a kernel buffer, but there is no guarantee that the data has been written out to its intended destination.

When a user-space application issues a write() system call, the Linux kernel performs a few checks and then simply copies the data into a buffer.

Later, in the background, the kernel gathers up all of the dirty buffers, which are buffers that contain data newer than what is on disk, sorts them optimally, and writes them out to disk (a process known as writeback).

This allows write calls to occur relatively fast, returning almost immediately. It also allows the kernel to defer writes to more idle periods and batch many writes together.

Synchronised I/O

The simplest method of ensuring that data has reached the disk is via the fsync() system call:

```
#include <unistd.h>

int fsync (int fd);
```

Synchronised I/O

The simplest method of ensuring that data has reached the disk is via the fsync() system call:

```
#include <unistd.h>

int fsync (int fd);
```

A call to fsync() ensures that all dirty data associated with the file mapped by the file descriptor fd are written back to disk.

Synchronised I/O

The simplest method of ensuring that data has reached the disk is via the fsync() system call:

```
#include <unistd.h>

int fsync (int fd);
```

A call to fsync() ensures that all dirty data associated with the file mapped by the file descriptor fd are written back to disk. The file descriptor fd must be open for writing.

Synchronised I/O

The simplest method of ensuring that data has reached the disk is via the fsync() system call:

```
#include <unistd.h>

int fsync (int fd);
```

A call to fsync() ensures that all dirty data associated with the file mapped by the file descriptor fd are written back to disk. The file descriptor fd must be open for writing.

The call writes back both data and metadata in the inode. It will not return until the hard drive says that the data and metadata are on the disk.

Synchronised I/O

Linux also provides the system call fdatasync():

```
#include <unistd.h>

int fdatasync (int fd);
```

Synchronised I/O

Linux also provides the system call fdatasync():

```
#include <unistd.h>

int fdatasync (int fd);
```

This system call does the same thing as fsync(), except that it only flushes data and metadata required to properly access the file in the future.

Synchronised I/O

Linux also provides the system call fdatasync():

```
#include <unistd.h>

int fdatasync (int fd);
```

This system call does the same thing as fsync(), except that it only flushes data and metadata required to properly access the file in the future.

Example: a call to fdatasync() will flush a file's size (needed to read the file correctly). The call does not guarantee that nonessential metadata (modification timestamp) is synchronised to disk.

Synchronised I/O

Linux also provides the system call fdatasync():

```
#include <unistd.h>

int fdatasync (int fd);
```

This system call does the same thing as fsync(), except that it only flushes data and metadata required to properly access the file in the future.

Example: a call to fdatasync() will flush a file's size (needed to read the file correctly). The call does not guarantee that nonessential metadata (modification timestamp) is synchronised to disk.

Remark: fdatasync() is therefore potentially faster than fsync().

Synchronised I/O

Less optimal, but wider in scope, the sync() system call is provided for synchronising all buffers to disk:

```
#include <unistd.h>

void sync (void);
```

Synchronised I/O

Less optimal, but wider in scope, the sync() system call is provided for synchronising all buffers to disk:

```
#include <unistd.h>

void sync (void);
```

The function always succeeds and, upon return, all buffers—both data and metadata—are guaranteed to reside on disk.

Synchronised I/O

Less optimal, but wider in scope, the sync() system call is provided for synchronising all buffers to disk:

```
#include <unistd.h>

void sync (void);
```

The function always succeeds and, upon return, all buffers—both data and metadata—are guaranteed to reside on disk.

The standards only requires that sync() initiates the process of committing all buffers to disk.

Synchronised I/O

Less optimal, but wider in scope, the sync() system call is provided for synchronising all buffers to disk:

```
#include <unistd.h>

void sync (void);
```

The function always succeeds and, upon return, all buffers—both data and metadata—are guaranteed to reside on disk.

The standards only requires that sync() initiates the process of committing all buffers to disk. Linux, however, does wait until all buffers are committed.

Synchronised I/O

Less optimal, but wider in scope, the sync() system call is provided for synchronising all buffers to disk:

```
#include <unistd.h>

void sync (void);
```

The function always succeeds and, upon return, all buffers—both data and metadata—are guaranteed to reside on disk.

The standards only requires that sync() initiates the process of committing all buffers to disk. Linux, however, does wait until all buffers are committed.

The O_SYNC flag may be passed to open(), indicating that all I/O on the file should be synchronised.

Closing files

After a program has finished working with a file descriptor, it can unmap the file descriptor from the associated file via the close() system call:

```
#include <unistd.h>

int close (int fd);
```

Closing files

After a program has finished working with a file descriptor, it can unmap the file descriptor from the associated file via the close() system call:

```
#include <unistd.h>

int close (int fd);
```

The given file descriptor is then no longer valid, and the kernel is free to reuse it as the return value to a subsequent open() or creat() call.

Seeking

Normally, I/O occurs linearly through a file, and the implicit updates to the file position caused by reads and writes are all the seeking that is needed.

Seeking

Normally, I/O occurs linearly through a file, and the implicit updates to the file position caused by reads and writes are all the seeking that is needed.

Some applications, however, want to jump around in a file, providing random rather than linear access.

Seeking

Normally, I/O occurs linearly through a file, and the implicit updates to the file position caused by reads and writes are all the seeking that is needed.

Some applications, however, want to jump around in a file, providing random rather than linear access.

The lseek() system call is provided to set the file position of a file descriptor to a given value.

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fd, off_t pos, int origin);
```

Seeking

The behavior of lseek() depends on the origin argument, which can be one of the following:

- SEEK_CUR: The current file position of fd is set to its current value plus pos, which can be negative, zero, or positive.

Seeking

The behavior of lseek() depends on the origin argument, which can be one of the following:

- SEEK_CUR: The current file position of fd is set to its current value plus pos, which can be negative, zero, or positive.
- SEEK_END: The current file position of fd is set to the current length of the file plus pos, which can be negative, zero, or positive.

Seeking

The behavior of lseek() depends on the origin argument, which can be one of the following:

- SEEK_CUR: The current file position of fd is set to its current value plus pos, which can be negative, zero, or positive.
- SEEK_END: The current file position of fd is set to the current length of the file plus pos, which can be negative, zero, or positive.
- SEEK_SET: The current file position of fd is set to pos.

To be continued...

Buffered I/O

Prof. Michele Loreti

Laboratorio di Sistemi Operativi

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

Buffered I/O

A block is an abstraction representing the smallest unit of storage on a filesystem.

Buffered I/O

A block is an abstraction representing the smallest unit of storage on a filesystem.

Inside the kernel, all filesystem operations occur in terms of blocks.

Buffered I/O

A block is an abstraction representing the smallest unit of storage on a filesystem.

Inside the kernel, all filesystem operations occur in terms of blocks.

Programs that have to issue many small I/O requests to regular files often perform user-buffered I/O.

Standard I/O

The standard C library provides the standard I/O library (often simply called stdio), which in turn provides a platform-independent, user-buffering solution.

Standard I/O

The standard C library provides the standard I/O library (often simply called stdio), which in turn provides a platform-independent, user-buffering solution.

StandardI/O routines do not operate directly on file descriptors. Instead, they use their own unique identifier, known as the file pointer.

Standard I/O

The standard C library provides the standard I/O library (often simply called stdio), which in turn provides a platform-independent, user-buffering solution.

StandardI/O routines do not operate directly on file descriptors. Instead, they use their own unique identifier, known as the file pointer.

Inside the *C* library, the file pointer maps to a file descriptor. The file pointer is represented by a pointer to the FILE typedef.

Opening files...

Files are opened for reading or writing via fopen():

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

Opening files...

Files are opened for reading or writing via fopen():

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

This function opens the file path with the behaviour given by mode and associates a new stream with it.

Opening files...

Modes:

- r: Open the file for reading (stream positioned at the start of the file).
- r+: Open the file for both reading and writing (stream positioned at the start of the file).
- w: Open the file for writing. If the file exists, it is truncated to zero length.
- w+: Open the file for both reading and writing. If the file exists, it is truncated to zero length. If the file does not exist, it is created (stream positioned at the start of the file).
- a: Open the file for writing in append mode. The file is created if it does not exist (stream positioned at the end of the file).
- a+: Open the file for both reading and writing in append mode. The file is created if it does not exist (stream positioned at the end of the file).

Opening a Stream via File Descriptor

The function fdopen() converts an already open file descriptor (fd) to a stream:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

Opening a Stream via File Descriptor

The function fdopen() converts an already open file descriptor (fd) to a stream:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

The possible modes are the same as for fopen() and must be compatible with the modes originally used to open the file descriptor.

Opening a Stream via File Descriptor

The function fdopen() converts an already open file descriptor (fd) to a stream:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

The possible modes are the same as for fopen() and must be compatible with the modes originally used to open the file descriptor.

Once a file descriptor is converted to a stream, I/O should no longer be directly performed on the file descriptor.

Closing streams

The fclose() function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Closing streams

The fclose() function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Any buffered and not-yet-written data is first flushed. On success, fclose() returns 0. On failure, it returns EOF and sets errno appropriately.

Closing streams

The fclose() function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Any buffered and not-yet-written data is first flushed. On success, fclose() returns 0. On failure, it returns EOF and sets errno appropriately.

The fcloseall() function closes all streams associated with the current process, including standard in, standard out, and standard error:

```
#define _GNU_SOURCE
#include <stdio.h>

int fcloseall (void);
```

Reading from a stream

Reading a Character at a Time:

```
int fgetc (FILE *stream);
```

Reading from a stream

Reading a Character at a Time:

```
int fgetc (FILE *stream);
```

Putting the character back:

```
int ungetc (int c, FILE *stream);
```

Reading from a stream

Reading a Character at a Time:

```
int fgetc (FILE *stream);
```

Putting the character back:

```
int ungetc (int c, FILE *stream);
```

Reading Binary Data:

```
size_t fread(void *buf, size_t size, size_t nr, FILE *stream)
    ;
```

Writing to a stream

Writing a Single Character:

```
int fputc (int c, FILE *stream);
```

Writing to a stream

Writing a Single Character:

```
int fputc (int c, FILE *stream);
```

Writing a String of Characters:

```
int fputs (const char *str, FILE *stream);
```

Writing to a stream

Writing a Single Character:

```
int fputc (int c, FILE *stream);
```

Writing a String of Characters:

```
int fputs (const char *str, FILE *stream);
```

Writing Binary Data:

```
size_t fwrite(void *buf,size_t size, size_t nr, FILE *stream)
    ;
```

Writing to a stream

Writing a Single Character:

```
int fputc (int c, FILE *stream);
```

Writing a String of Characters:

```
int fputs (const char *str, FILE *stream);
```

Writing Binary Data:

```
size_t fwrite(void *buf,size_t size, size_t nr, FILE *stream)
    ;
```

Empty writing buffer:

```
int fflush (FILE *stream);
```

Seeking a stream

The fseek() function, the most common of the standard I/O seeking:

```
int fseek (FILE *stream, long offset, int whence);
```

Seeking a stream

The fseek() function, the most common of the standard I/O seeking:

```
int fseek (FILE *stream, long offset, int whence);
```

Alternatively, standard I/O provides fsetpos():

```
int fsetpos (FILE *stream, fpos_t *pos);
```

Seeking a stream

The fseek() function, the most common of the standard I/O seeking:

```
int fseek (FILE *stream, long offset, int whence);
```

Alternatively, standard I/O provides fsetpos():

```
int fsetpos (FILE *stream, fpos_t *pos);
```

Standard I/O also provides rewind(), as a shortcut:

```
void rewind (FILE *stream);
```

The invocation rewind(stream) resets the position back to the start of the stream. It is equivalent to:

```
fseek (stream, 0, SEEK_SET);
```

Excercise: Caesar cipher

In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques.

Excercise: Caesar cipher

In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques.

It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.

Excercise: Caesar cipher

In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques.

It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.

Example: $CC(\text{“Hello World!”}, 5) = \text{“Mjqqt Btwqi!”}$.

Excercise: Caesar cipher

In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques.

It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.

Example: $CC(\text{“Hello World!”}, 5) = \text{“Mjqqt Btwqi!”}$.

Exercise: Write a program `cesar.c` that encrypt/decrypt a file by using the Caesar cipher.

To be continued...