Linux File I/O

HGU

System call: a Cornerstone of System programming

- System call:
  - System programming starts and ends with *system calls*
  - **System calls** are function invocations made from user space – your text editors, favorite game, and so on to requests a service from the **kernel** (core part of OS)
- C library
  - On modern Linux systems, the C library is provided by **GNU** libc, ***glibs***
  - The GNU C Library project provides the core library of GNU System
- C Compiler
  - In Linux, the standard C Compiler is provided by the **GNU Complier Collection** (**gcc**)
- API (Application Programming Interface)
  - It is a software intermediary that allows two applications to talk to each other
  - It is based on source code

File Management Functions

- file descriptor
  - fd stands for **file descriptor**
  - File descriptor is an integer that uniquely identifies an opened file
  - A file descriptor is a non-negative integer, generally represented in the C programming language as the **type int**
- Most of the File Management Functions are declared in the <fcntl.h> <unistd.h>, <stdio.h>, <sys/stat.h>, <sys/types.h>, and <errno.h>
  - fcntl.h (file control operation): open, create, fcntl
  - unistd.h (lower-level file handling): read, write, close, lseek, unlink
  - stdio.h (with FILE type): fopen, fclose, fread, fwrite, fgets, fputs, frpintf, fscanf
  - sys/stat.h (file information through stat): stat, fstat, lstst, chmod, mkdir
  - sys/types.h (data types used in system calls including file operations)

Opening Files

- A file is opened and a **file descriptor** is obtained with open() system call

- Syntax

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

int open(const char * name, int flags);  // open an existing file
int open (const char * name, int flags, mode_t mode);
                // create a new file if it does not exist when flags include O_CREAT
```

Opening Files using open()

- int open (const char * name, int flags);

File name with path   specifies mode

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

File name with path   **O_CREAT**   specifies mode in
3 octal digits for permission

open() flags

- **O_RDONLY** – open for reading only
- **O_WRONLY** – open for writing only
- **O_RDWR** -- open for reading and writing
- O_CREAT – create the file if it does not exist
- O_EXCL – when used with O_CREAT, if the file already exists it is an error and the open() will fail
- O_APPEND – open the file in append mode
- O_SYNC: Open the file for synchronous I/O. Writes will not return until the data is physically written to disk (all data and metadata are synchronized).
- O_DSYNC: Similar to O_SYNC, but only the file's data (not its metadata) is synchronized.
- O_RSYNC: Synchronizes reads with writes, ensuring that reads see the latest data written to the file.

Security of Files

- Linux is the Multi-user operating system which can be accessed by many users simultaneously
- This raises security concerns as an unsolicited user can corrupt, change or remove critical data
- For effective security, Linux divides authorization into two Levels:
  - Ownership
  - Permission

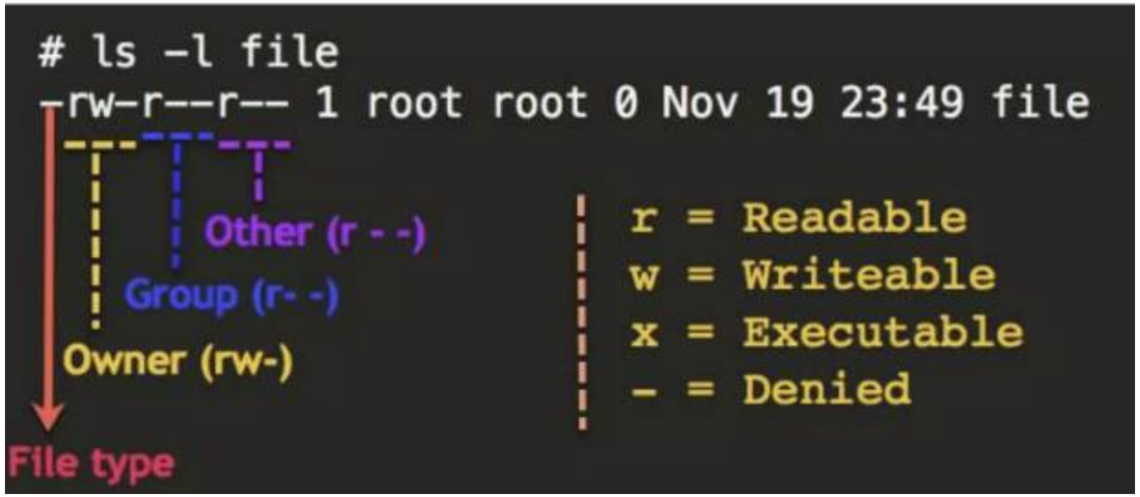
Ownership of Files

- Every file and directory on your Linux system is assigned 3 types of owner, given below:
- User: a user is the **owner of the file**. By default the person who created a file becomes its owner. (Hence, a user is also called an owner)
- Group
  - A user-group can contain multiple users. All users belonging to a group will have the same access permissions to the file
  - Instead of manually assigning permissions to each user, you could add all users to a group, and assign group permission to file
- Other
  - Any other user who has access to a file
  - Practically, it means everybody else.

Permissions of Files

- Every file and directory in your UNIX/Linux system has following 3 permissions defined for all the 3 owners discussed above
- The three permissions are
  - Read: permission giving the authority to **open and read** a file
  - Write: permission giving the authority to **modify** the contents of a file
  - Execute: permission giving the authority to **execute** a file

sample:



File Permission by ls –l command shows 9 characters

-rwxrwxrwx *or* **3 octal numbers for he mode arg of open()**

file type d, l, c, b

user group others

Permission of Files (Open System call mode)

- S_IRWXU : (00700) – All authority for owner
- S_IRUSR : (00400) – Read authority for owner
- S_IWUSR : (00200) – Write authority for owner
- S_IXUSR : (00100) – Execute authority for owner
- S_IRWXG : (00070) – All authority for group
- S_IRGRP : (00040) – Read authority for Group
- S_IWGRP : (00020) – Write authority for Group
- S_IXGRP : (00010) – Execute authority for Group
- S_IRWXO : (00007) – All authority for Others
- S_IROTH : (00004) – Read authority for Others
- S_IWOTH : (00002) – write authority for Others
- S_IXOTH : (00001) – Execute authority for Others

Return Values

- Both open() and create()
  - On Success, returning zero**(0)**
  - On Error, returning a non-zero number (**-1**), and **set errno** to an appropriate error value

open() Sample Code

```
int fd;
fd = open ("home/user1/car", O_RDONLY);
if (fd == -1) /* error */
```

the file /home/usr1/car .is opened for reading

```
int fd;
fd = open ("home/user1/pearl," O_WRONLY | O_TRUNC);
if (fd == -1) /* error */
```

if the file already exists, it will be truncated to a length of zero.
If the file does not exists, the call will fail.

```
int fd;

fd = open (file, O_WRONLY | O_CREAT  | O_TRUNC ,
    S_IWUSR  | S_IRUSR |  S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
    /* error */
```

rw-rw-r-- → 0661

0661 & ~0022 = 0644

If the file does not exist, assuming a umask of 022 it is created with the permission 0644.
If it exists, it is truncated to zero length

create()

- create() System call create and open a file
- create is a special case of open() system call
- Syntax:

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

File name with path　specifies mode

create() sample code

```
int fd;

fd = create( file, 0644);
if(fd == -1 )
    /* error handling */
```

==

```
int fd;

fd = open( file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if(fd == -1 )
    /* error handling */
```

Reading via read()

- The mechanism used for reading : the read() system call
- Syntax:

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

File descriptor

address of the first byte of buffer

length or count of buffer

- The ***ssize_t*** type is *signed version of size_t* (the negative values are used to connect errors)
- the maximum value of an ssize_t is **SSIZE_MAX** which is 2,147,483,647 bytes on a 32-bit machine

Reading via read()

- When read() returns successfully, its return value is **the number of bytes** actually read and placed in the buffer
- if len is zero, read returns zero and has no other results
- On Success, a non-negative integer is returned indicating the number of bytes actually read
- Otherwise(Failure), a **-1** is returned

Call to read() Results in many possibilities

- Case 1: returns a value equal to len. The results are as intended(**Success**).
- Case 2: returns a value less than len, but greater than 0 if any *interrupt* occurs during read or EOF is reached before len bytes were read.
- Case 3: **returns 0**, this indicates reaching *EOF and* there is nothing to read
- Case 4: the call blocks because no data is currently available. (It won't happen in nonblocking mode)
- Case 5: returns -1
  - errno == EINTR : a signal was received before any bytes were read. The call can be reissued.
  - errorno == EAGAIN : read would block because no data is currently available and the request should be reissued later. It happens *only in nonblocking mode*
  - Other errono error values : serious error (EINV, EBADF, EFAULT, EIO, ENOMEM. …)

read() Sample Code

```
unsigned long word;
ssize_t nr;

/* red a couple bytes into 'word' from 'fd' */
nr = read (fd, &word, sizeof(unsigned long));
if( nr == -1)
        /* error handling*/
```

```
/* reading all bytes */
ssize_t ret;

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

Nonblocking Reads

- By default, *read()* waits until at least one byte is available to return to the application; this default is called "**blocking**" mode
- Alternatively, individual file descriptors can be switched to "**non-blocking" mode,** which means that a *read()* on a slow file will return immediately, even if no bytes are available.
- Nonblocking I/O, if given file descriptor was opened <u>in nonblocking mode.</u>

  ← **open()** with **O_NONBLOCK** flag

Other Error Values of read()

- Possible errno vaues after a failure on read() include:
  - EBADF: The given file descriptor is invalid or is not open for reading
  - EFAULT: The pointer provided by buf is not inside the calling process's address space
  - EINVAL: The file descriptor is mapped to an object that does not allow reading
  - EIO: a low-level I/O error occurred
  - ENOMEM: not enough memory is available

Writing with write()

- The most basic and common system call for writing : write()
- it writes data from a buffer declared by the user to a given device, such as file
- This is the primary way to output data from a program by directly using a system call
- Syntax:

#include <unistd.h>

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

File descriptor

address of source

length of the data to be written

Return value of write()

- On success, the number of bytes written
- On Error, -1 is returned and errno is set appropriately
- Partial writes:
  - a successful write() may transfer fewer than count bytes
  - Such partial writes can occur for various reasons
    - If there was insufficient space to the disk to write all of the requested bytes
    - Because of interrupt

write() mode

- Append Mode
  - When fd is *opened in append mode* (**O_APPEND**), writes do not occur at the fd's current file position. Instead, they occur at the current end of the file
- Nonblocking Writes
  - When fd is *opened in nonblocking* mode (via **O_NONBLOCK**), and the write as issued would normally block, the write() system call returns -1 and set errno to EAGAIN
  - The request should be reissued later

Other Error Codes of Write

- EBADF: The given file descriptor is not valid or is not open for writing
- EFAULT: the pointer provided by buf points outside of the process's address space
- EINVAL: The given file descriptor is mapped to an object that is not suitable for writing
- ENOSPC: the file system backing the given file decript does not have sufficient space
- EINTR: write was blocked by interrupt (in this case retry write())

simple Write() sample

```
ssize_t ret, nr;

while (len != 0 && (ret = write(fd, buf, len))!= 0) {
   if(ret == -1) {
      if(errorno == EINTR)
         continue;
       perror(“write”);
       break;
    }
    len-= ret;
    buf += ret;
}
```

How to perform write()

- Size Limits on write()
  - If count is larger than SSIZE_MAX, the results of the call to write are undefined
  - A call to write() with count of zero results in the call returning immediately with a return value of 0
- Behavior of write
  - 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 buffer that contain data newer* than what is on disk, sorts them optimally, and writes them out to disk.

Synchronized I/O

user issues write() some data which contains three lines to file called sample.txt located in your z drive

request

Kernel or OS accept the request and collect data from write

time frame 1: data is collected to buffer
time frame 2: data is collected to buffer
time frame 3: data is collected to buffer
and then
It will sort the data in buffers and in optimal time the consolidated data will be moved to sample.txt located in your z drive

process

Synchronized I/O

- By default, the Linux kernel writes data to disk asynchronously
- Writes are buffered (cached) in memory, and written to the storage device at the optimal time
- The **Synchronous I/O** provides some functions to ensure that all operations finish before they return.
- **Sync** System call:
  - The sync system call forces an immediate write of all cached data to disk but it doesn’t wait to complete
  - This call initiates the process of committing all buffers to disk
- Sync Syntax: sync [option] [file]
- Examples:
  - sync –d : sync only file data not metadata
  - sync –f <file>: sync file systems which contains the files

Synchronized I/O: fsync()

- Syntax of fsync()

  #include <unistd.h>

  int fsync(int fd);

  fd: file descriptor of open file

- The call to above function ensures that *all dirty data* associated with the file mapped by the file descriptor fd are written back to disk
- fsync() writes back both data and metadata (information summary)

Synchronized I/O: fdatasync()

- Syntax

```
#include <unistd.h>
int fdatasync(int fd);
```

- A call to fdatasync() ensures that only the file's data blocks are synchronized to disk, along with essential metadata like file size.
- The call does **not** guarantee that nonessential metadata such as access time, ownership, or permission is synchronized to disk,
- is therefore potentially faster

Flags of open () for Synchronized I/O

- **O_SYNC** flag requires that any write operations block until all data and all metadata have been written to persistent storage
- **O_DSYNC** flag specifies that only normal data be synchronized after each write operation, not metadata

Direct I/O

- The Linux kernel implements a complex layer of caching, buffering, and I/O management between devices and applications
- A high-performance application may wish to bypass this layer of complexity and perform its own I/O management
- Providing the **O_DIRECT** flag to **open()** instructs the kernel to minimize the presence of I/O management
  - When O_DIRECT flag is provided, I/O will initiate directly from user-space buffer to the device, *bypassing the page cache*
  - Direct I/O will be synchronous; operations will not return until completion
- For an I/O operation to be performed as direct I/O, it must meet certain alignment criteria (depending on disk driver, disk controller, and memory management hardware). Otherwise the direct I/O request is performed as data synchronous I/O.

Closing Files

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

```
#include <unistd.h>
int close(int fd);
```

- A call to close() unmaps the open file descriptor fd and disassociates the file from process
- It is a common mistake to not check the return value if close()
- There are handful of possible errno values on failure. Other than EBADF the most important error value is EIO, indicating a low-level I/O error probably unrelated to the actual close

Seeking : lseek()

- lseek() is a system call that is use to *change the location of the read/write pointer* of a file descriptor
- Syntax:

#include <sys/types.h>

#include <unistd.h>

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

- origin parameter
  - **SEEK_CUR** : the current file position of fd is set to its current value plus pos, which can be negative, zero, or positive. **A pos of zero returns current file position**
  - **SEEK_END**: the current file position of fd is set to the current length of file + pos. **A pos of zero sets the offset to the end of the file**
  - **SEEK_SET**: the current file position of fd is set to pos. **A pos of zero sets the offset to the beginning of the file.**

Seeking: lseek() Error Values

- EBADF: the given *file descriptor* does not refer to an open file descriptor
- EINVAL: the value given for *origin* is not valid or the resulting file position would be negative
- EOVERFLOW: *The resulting file offset cannot be represented in an off_t*
- ESPIPE: the given file descriptor is associated with an *unseekable object*, such as a pipe, FIFO, or socket

Positional Reads/Writes

- Instead of lseek(), Linux provides two variants of the read() and write() system calls
- Both receive the file position from which to read or write
- Upon completion, they do *not update the file position*
- The read form is called *pread():*

```
#define _XOPEN_SOURCE  500
#include <unistd.h>
ssize_t pread(int fd, void *buf,  size_t count, off_t pos);
```

- This call read up to *count* bytes into buf from the file descriptor *fd* at file position *pos*.

Positional Reads/Writes

- The write form is called *pwrite():*

```
#define _XOPEN_SOURCE 500
#include <unistd.h>
ssize_t pwrite(int fd, void *buf, size_t count, off_t pos);
```

- This call writes up to count bytes from buf to the file descriptor fd at file position pos.

Truncating Files

- Linux provides two system calls, ftruncate() and truncate(), for truncating the length of a file

- Syntax of ftruncate:

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t len);
```

- Syntax of truncate:

```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *name, off_t len);
```

Truncating Files

- These system calls usually truncate a file to a *size smaller than its current length*
- the **ftruncate()** system call operates on the file descriptor given by ***fd***, which must be open for writing
- The **truncate()** system call operates on the ***filename*** given by path, which must be writable.
- Returns
  - On success, return **0**
  - On error, they return **-1** and set **errorno** as appropriate

Kernel Internals

- The Kernel subsystem consists of :
  - the virtual filesystems (VFS)
  - the page cache, and
  - page writeback
- *The virtual filesystem*, occasionally called a virtual file switch, is a mechanism of *abstraction* that allows the Linux kernel to call **filesystem** functions and manipulate filesystem data without knowing the specific type of filesystem being used
- *Linux filesystem* is generally a *built-in layer of a Linux OS* used to handle the data management of the storage.
  - helps to arrange the file on the disk storage.
  - manages the file name, file size, creation date, and more information about a file

Kernel Internals

- The Page Cache:
  - the page cache is an in-memory store of recently accessed data from an on-disk filesystem.
  - Disk access is painfully slow, particularly relative to today's processor speeds
  - Storing requested data in memory allows the kernel to fulfill subsequent requests for the same data from memory, <u>avoiding repeated disk access</u>

Kernel Internals

- Page Writeback
  - Eventually the dirty buffers need to be committed to disk, synchronizing the on-disk files with the data in memory. This is known as writeback. It occurs in two situations:
    - When free memory shrinks below a configurable threshold, dirty buffers are written back to disk so that the now-clean buffers may be removed, freeing memory
    - When a dirty buffer ages beyond a configurable threshold, the buffer is written back to disk. This prevents data from remaining dirty indefinitely

Buffered I/O

HGU

Introduction on buffered I/O

- I/O **buffering** is a mechanism that improves the throughput of I/O operation.
  - Throughput is the amount of work completed in a unit of time
- It is implemented directly in hardware and the corresponding drivers (hence the block devices found in UNIX-like systems), and also universal among programming language standard libraries

I/O Latencies and Buffering

- I/O operations often have high latencies.
  - latency: the time between the initiation of I/O process and its completion
  - Most of this latency is due to hardware itself; for example, information cannot be read from or written to a hard disk until spinning of the disk brings target sectors under the disk head
- The Latency is reduced by having one or more input and output buffers associated with each device
- A *buffer* is a memory area that stores data being transferred between two devices or between a device and an application

I/O Buffering

- The goal of the buffering provided by the standard I/O library
- Buffering is done for three reasons
  - To cope with a **speed mismatch** between producer and consumer of a data stream
  - To provide adaptation for data that have **different data-transfer sizes**
  - To support **copy semantics for** the application I/O
- Copy semantics
  - Data is first copied from user application memory into kernel memory
  - Data from kernel memory is then written to device
  - Prevents application from changing contents of a buffer before it is done being written

User-Buffered I/O

- User buffered I/O, shortened to *buffering or buffered I/O*, refers to the technique of temporarily *storing the results of an I/O operation in user-space* before transmitting it to the kernel (in case of write) or before providing it to user process (in the case of reads)
- By so buffering the data, you can minimize the number of system calls and can block-align I/O operations, which may *improve the performance* of user application.

Block Size and Buffered I/O

- In practice, Block size are 512, 1024, 2,048, 4,906, or 8,192 bytes.
- A large performance gain is realized simply by performing operations with respect to block size
- This is because the kernel and <u>hardware speaks in terms of blocks</u>
- Thus, using block size or a value that fits neatly inside of a block guarantees block-aligned I/O and prevents extraneous (external) work inside kernel.

Example(1): A Simple getchar()

```
int getchar(void) {
    static char c;
    if (read(0, &c, 1) == 1)
        return c;
    else return EOF;
}
```

Read **one character** from **`stdin`**

- File descriptor `0` is `stdin`
- `&c` points to the buffer
- `1` is the number of bytes to read

Read returns the number of bytes read

- In this case, `1` byte means success

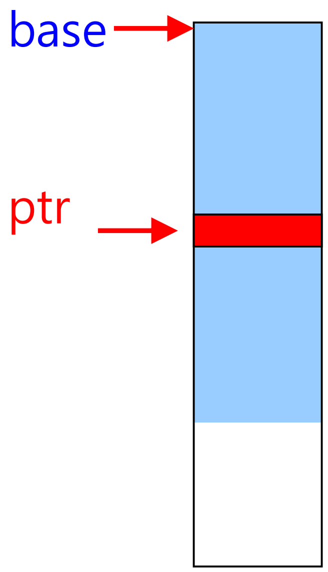
Example(2): Making getchar() More Efficient

- Poor performance reading one byte at a time
  - read() system call is accessing the device (e.g., a disk)
  - Reading one byte from disk is very time consuming
  - Better to read and write in *larger chunks*
- Buffered I/O
  - Read a large chunk from disk into a buffer
  - Similarly, for writing, write individual bytes to a buffer
    - And write to disk when full, or when stream is closed
    - Known as "flushing" the buffer

Example(3): Better getchar() with Buffered I/O

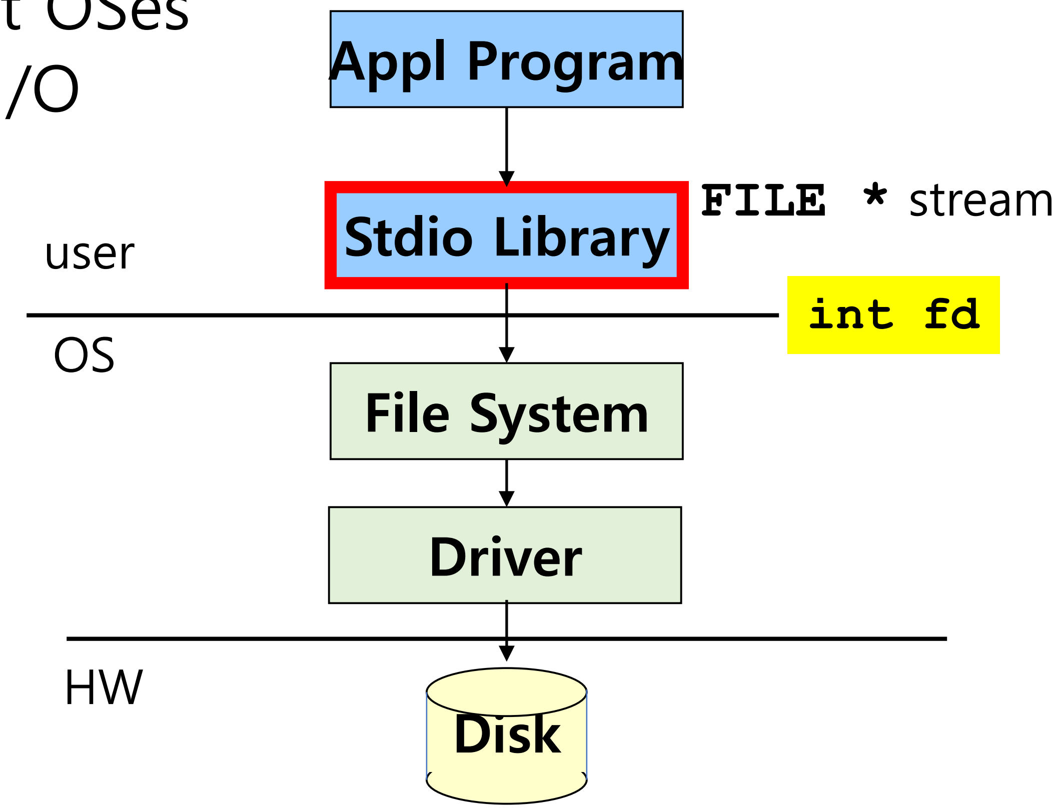
```c
int getchar(void) {
    static char base[1024];
    static char *ptr;
    static int cnt = 0;

    if (cnt--)
        return *ptr++;

    cnt = read(0, base, sizeof(base));
    if (cnt <= 0)
        return EOF;
    ptr = base;
    return getchar();
}
```

persistent variables



Standard I/O Library

- Portability
  - Generic I/O support for C programs
  - Specific implementations for various host OSes
  - Invokes the OS-specific system calls for I/O
- Abstractions for C programs
  - Streams
  - Line-by-line input
  - Formatted output
- Additional optimizations
  - Buffered I/O
  - Safe writing



The Core of Standard I/O Library in C

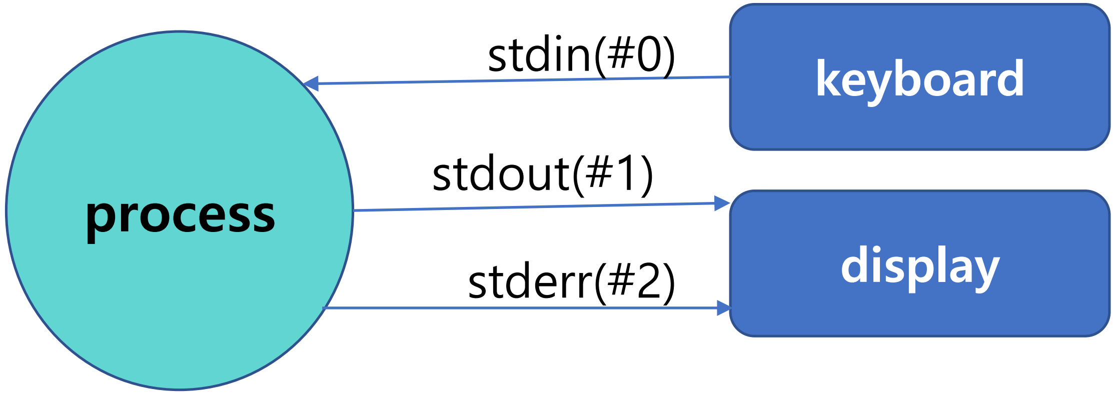
1. FILE Structure: fd, buffer, permission
2. Standard Streams: stdin(0), stdout(1), stderr(2)
3. FILE I/O functions
   - Generic: fopen(fp), fclose(fp), fread(buf, size, count, fp), fwrite(buf, size, count,fp),
   - Text file I/O: fgets(buf, size, fp), fputs(buf, fp)
4. Formatted I/O: printf, scanf, fprintf(fp, format, …), fscanf(fp,format, …)
5. character based I/O:fgetc(fp), fputc(c,fp), getchar(), putchar(c)
6. Buffering: fflush(fp)
7. Error Handling : ferror(fp), feof(fp), clearer(), perror()

Stream Abstraction

- Any source of input or destination for output is regarded as file
  - E.g., keyboard as input, and screen as output
  - E.g., files on disk or CD, network ports, printer port, …
- Accessed in C programs through **file pointers**
  - E.g., **`FILE *fp1;`**
  - E.g., **`fp1 = fopen("myfile.txt", "r");`**
- The standard C library provides I/O library (**stdio**), which in turn provides a platform-independent user-buffering solution
  - **Three standard streams** provided by stdio.h
  - Standard Streams: **stdin, stdout,** and **stderr**

Standard I/O Streams

- The three I/O connections are called **stdin (standard input), stdout (standard output), and stderr (standard error).**
- stdin is a stream from which a program reads its input data
- stdout is a stream to which a program writes its output data
- stderr is another output stream typically used by programs to output error messages or diagnostics.



Sequential Access to a Stream

- Each stream has an associated **file position**
  - Starting at beginning of file (if opened to read or write)
  - Or, starting at end of file (if opened to append)

- **Read/write** operations **advance the file position**
  - Allows sequencing through the file in sequential manner
- Support for random access to the stream
  - Functions to learn current position and **seek** to new one

Standard I/O: File Pointer

- Standard I/O Routines do not operate directly on *file descriptors*.
- Instead, they use **file pointer**
- file pointer is a pointer which is used to handle and keep track on the files being accessed
- a new data type called “FILE” is used to declare file pointer

  FILE *fp; // fp is a file pointer

- **FILE** is defined in stdio.h

Details of FILE in stdio.h (K&R 8.5)

```
#define OPEN_MAX 20  /* max files open at once */

typedef struct _iobuf {
   int  cnt;     /* num chars left in buffer */
   char *ptr;    /* ptr to next char in buffer */
   char *base;   /* beginning of buffer */
   int  flag;    /* open mode flags, etc. */
   char fd;      /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

Opening File with fopen()

- files are opened for reading or writing via fopen():

```
#include <stdio.h>
FILE *fopen(const char *file_path, const char *mode);
```

- This function opens the file path with the behavior given by mode and associates a new **stream** with it
- A stream is a sequence of data elements made available over time

Mode of fopen()

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

fopen vs. open

fopen()	open()
fopen series are **standard C library** functions	the open series are defined by POSIX and are **system calls** in UNIX system
when using fopen functions, you must define an object that refers to a file. It is called "**file handler**" and is a struct	the Open series uses an int integer call "**file descriptor**"
On failure, returns NULL and sets errorno On success, returns valid pointer to FILE	On failure, returns -1 and sets errorno On success, returns file descriptor (int)

Opening a Stream via File Descriptor

- function fdopen() converts an already open file descrtptor to a stream

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

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

```
FILE * stream;
int fd;
fd = open("/home/kid/map.txt", O_RDONLY);
if (fd == -1)
      /* Error Handling */
stream = fdopen(fd, "r");
if(stream == NULL)
     /* Error Handling */
```

Closing Streams with fclose()

- **fclose()** function closes a given stream:

```
#include <stdio.h>
int fclose(FILE *fp);
```

- Any buffered and not-yet-written data is first flushed.
- On success, fclose() returns 0.
- On Failure, it returns EOF and sets errno appropriately.
- **fcloseall()** function closes all steams associated with current process including stdin, stdout, and stderr:

```
#define _GNU_SOURCE
#include <stdio.h>
int fcloseall(void);
/* fcloseall always returns 0 */
```

Reading from Stream

- standard C library implements multiple functions for reading from an open stream
- most popular 3 functions:
  - reading one character at a time: **fgetc()**
  - reading one entire text line at a time: **fgets()**
  - reading binary data: **fread()**

Reading a Character: fgetc()

- The ideal I/O pattern is simply reading one charcter at a time. The function fgetc() is to read a single character from a stream

```
#include <stdio.h>
int fgetc(FILE *stream);
```

- This function reads the next character (**unsigned char**) from the specified stream and advances the position indicator for the stream

fgetc() Sample Code

```
#include <stdio.h>
int main() {
  FILE *fp;
  int c;
  int n =0;
  fp = fopen("file.txt", "r");
  if(fp==NULL) {
    perror("Error in opening file");
    return -1;
 }
```

```
   do {
     c = fgetc(fp);
     if( feof(fp))
         break;
    printf("%c", c);
    } while(1);
    fclose(fp);
    return(0);
}
```

Putting the character back : ungetc()

- Standard I/O provides a function for pushing a character back onto a stream, allowing you to “peek” at the stream and return the character if it turns out that you don’t want it:

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

ungetc() Sample Code

```
#include <stdio.h>
int main() {
  FILE *fp;
  int c;
  char buffer[256];

  fp = fopen("file.txt", "r");
  if(fp==NULL) {
    perror("Error in opening file");
    return (-1);
 }
```

```
  while (!feof(fp)) {
    c = fgetc(fp);
    if(c == '!') /* replace ! with + */
      ungetc('+', fp);
    else
      ungetc(c, fp);
    c= fgetc( fp);
    fputc(c, stdout);
  }
  return(0);
}
```

Reading an Entire Line: fgets()

- the function fgets() reads a string from a given stream:

  #include <stdio.h>

  char *fgets(char *str, int size, FILE *stream)
- This function reads up to one less than size bytes from stream and stores the results in str
- a null character('₩0') is stored in the buffer after the last byte read in

fgets() Sample Code

```
#include <stdio.h>
int main() {
  FILE *fp;
  char str[60];
  /* opening file for reading */
  fp = fopen("file.txt", "r");
  if (fp == NULL) {
      perror("Error opening file");
      return (-1);
  }
```

```
    if ( fgets(str, 60, fp) != NULL){
     /* writing contents to stdout*/
       puts(str);
    }
    fclose(fp);
    return (0);
}
```

Reading Binary Data: fread()

- Sometimes, developers want to read and write complex binary data
- For this, the standard I/O library provides **fread**():
  - reads data from a file and store it in a buffer

#include <stdio.h>

size_t fread(void *buffer, size_t size, size_t count, FILE *stream);

  - buffer: pointer to the buffer where data will be stored. A buffer is region of memory used to temporarily store data
  - size: the size of each elements to be read
  - count: the number of elements to be read
  - stream: pointer to the FILE object from where data is to be read
- Upon Success execution it returns integer value equivalent to count. In case of an error or EOF, a value less than count is returned

Writing to a Stream

- standard C library implements multiple functions for writing to an open stream
- most popular 3 functions:
  - writing one character at a time: fputc()
  - writing one entire line at a time: fputs()
  - writing binary data: fwrite()

Writing a Character: fputc()

- The counterpart of function fgetc() is fputc()

  #include <stdio.h>

  int fputc(int c, FILE *stream);

- Parameters
  - c : This is the character to be written. This is passed as its int promotion
  - stream : it is the pointer to a FILE object that identifies the stream where the character is to be written
- Return Value
  - If there are no errors, the same character that has been written is returned. If Error occurs, EOF is returned and the error indicator is set.

fputc() Sample Code

```
#include <stdio.h>
int main() {
  FILE *fp;
  int ch;

  fp = fopen("file.txt", "w+");
  if(fp==NULL) {
    perror("Error in opening file");
    return -1;
 }
```

```
 for(ch = 33; ch <= 100; ch++)
       fputc( ch, fp);
   fclose(fp);
   return(0);
}
```

Writing a String of Characters: fputs()

- the function fputs() writes an entire string to a given stream:

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

  int fputs(char *str, FILE *stream)
- Parameters
  - str: an array containing the null-terminated sequence of characters to be written
  - stream: the pointer to a FILE object that identifies the stream where the string is to be written
- Return Value: the function returns a non-negative value, or else on error it returns EOF

fputs() Sample Code

```
#include <stdio.h>
int main()
{
    const char *buffer =“Hello world!”;
    fputs (buffer, stdout);
    return (0);
}
```

Writing Binary Data: fwrite()

- To directly store binary data such as C variables, standard I/O provides fwrite():

```
#include <stdio.h>
size_t fwrite(void *buffer, size_t size, size_t nr, FILE *stream);
```

  - buffer: pointer to the array of elements to be written
  - size: the size in bytes of each elements to be written
  - count: the number of elements, each one with the size of **size** bytes
  - stream: pointer to the FILE object that specifies an output stream
- Return value: It returns the total number of elements successfully returned as a size_t object, which is an integral data type.

fwrite() Sample Code

```
#include <stdio.h>
int main()
{
    FILE *fp;
    int iCount;
    char arr[6] = "Hello";

   fp = fopen("sample.txt", "wb");
   iCount = fwrite(arr, 1, 5, fp);
   fclose(fp);
   return (0);
}
```

Complete Sample Code for Buffered I/O

```
#include <stdio.h>
int main (void) {
  FILE *in, *out;
  struct pirate {
     char name[100]; /* real name */
     unsigned long booty; /* in pounds sterling */
     unsigned int beard_len; /* in inches */
   } p, blackbeard = { "Edward Teach", 950, 48 };
  out = fopen ("data", "w");
  if (!out) {
     perror ("fopen");
     return 1;
   }
   if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
    perror ("fwrite");
    return 1;
 }
```

```
 if (fclose (out)) {
     perror ("fclose");
     return 1;
 }
 strcpy(p.name,”None”); p.booty = 0L; p.beard_len = 0;
 in = fopen ("data", "r");
 if (!in) {
    perror ("fopen");
    return 1;
 }
 if (!fread (&p, sizeof (struct pirate), 1, in)){
    perror ("fread");
    return 1;
  }
  if (fclose (in)) {
    perror ("fclose");
    return 1;
  }
  printf ("name=₩"%s₩" booty=%lu beard_len=%u₩n",
            p.name, p.booty, p.beard_len);
  return 0;
}
```

Seeking a Stream: fseek()

- fseek() function, the most common standard I/O seeking interfaces, manipulates the file position of stream in accordance with offset where:

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

- Parameters
  - stream: the pointer to FILE obhect that identifies the stream
  - offset: the number of bytes to offset from whence
  - whence: the position from where offset is added

whence value	SEEK_SET	SEEK_CUR	SEEK_END
description	beginning of file	current position of file	end of file

- Return value: On success Zero(0), On failure non-zero value

fseek() Sample Code

- random search

```
#include <stdio.h>
int main() {
  FILE *fp = fopen("file.txt", "w+");
  fputs("This is tutorials-point.com", fp);
  fseek (fp, 7, SEEK_SET);
  fputs(" C Programming Language", fp);
  fclose(fp);
  return (0);
}
```

This isC Programmingint.com

Seeking a Stream

- Alternative to fseek() is **fsetpos()** and **rewind()**
- fgetpos():

```
#include <stdio.h>
int fsetpos(FILE *stream, fpos_t *pos);
```

  - Upon Success, it returns 0, and places the current stream position of stream in pos
- rewind():

```
#include <stdio.h>
void rewind (FILE *stream);
```

  - sets the position back to the start of the stream
  - equivaent to fseek(stream, 0, SEEK_SET)

Obtaining the Current Stream Position: ftell()

- Unlike lseek(), fseek() does not return the updated position.
- the **ftell()** function returns the current stream position in the stream:

  #include <stdio.h>

  long ftell (FILE *stream);
- On Error, ftell() returns -1 and errno is set appropriately

```
#include <stdio.h>
int main() {
  int len;
  FILE *fp = fopen("file.txt", "w+");
  if(fp == NULL) perror("Error open");
  fseek (fp, 0, SEEK_END);
  len = ftell(fp);
  fclose(fp);
  printf("Size of file : %d bytes", len);
  return (0);
}
```

Flushing a Stream: fflush()

- The standard I/O library provides an interface for writing out the user buffer to the kernel, ensuring that all data written to a stream is flushed via write(). The fflush() functions provides this functionality:

  #include <stdio.h>

  int fflush(FILE *stream);

- On invocation, any unwritten data in the stream pointed to by stream is flushed to the kernel.
- If stream is NULL**, all open input stream in the process are flushed**
- **On success, fflush returns 0, On failure it returns EOF and errno is set**

Errors and EOF

- Some of standard I/O interfaces, such as fread(), communicate failure back to the caller poorly., as they provides no mechanism for differentiating between error and end-of-file(EOF)
- With these calls, and on other occasions, it can be useful to check the status of a given stream to determine *whether it has encountered an error or reached end-of-file.*
- *Standard I/O provides two interfaces to this end.*
    - *ferror()*
    - *feof()*

Errors and EOF

- The function ferror() tests whether the error indicator is set on stream:

  #include <stdio.h>

  int ferror(FILE *stream);

- The error indicator is set by standard I/O interfaces in response to an error condition
- The function **returns *a nonzero value*** *if the indicator is set*, and **0** otherwise

Errors and EOF

- The function feof() tests whether the EOF indicator is set on stream:

  #include <stdio.h>

  int feof(FILE *stream)

- The EOF indicators is set by standard I/O interfaces when the end of file is reached
- The ***function returns nonzero*** value if the indicator is set, and **0** otherwise (not EOF)

Errors and EOF

- The clearer() function clears the error and the EOF indicator for stream:

```
#include <stdio.h>
void clearerr(FILE *stream);
```

- It has no return value, and cannot fail
- You should make a call to clearerr() only after checking the error and EOF indicator

Obtaining the File Descriptor Associated

- To obtain the file descriptor backing a stream, use fileno():

  #include <stdio.h>

  int fileno(FILE *stream);

- Upon success, fileno() returns the file descriptor associated with stream. On failure it returns -1

```
#include <stdio.h>
void main() {
    FILE *fp = fopen("file.txt", r);
    printf("FIle number is %d₩n", fileno(fp);
    fclose(fp);
}
```

Controlling the Buffering

- Standard I/O implements three types of user buffering and provides developers with an interface for controlling the type and size of the buffer.
- The different types of user buffering serve different purposes:
  - Unbuffered : No buffering – characters are transmitted to the system as they are written
  - Line-buffered – characters are transmitted to the system as a block when a new-line character is encountered
  - Block-buffered: characters are transmitted to the system as a block when a buffer is filled.

Controlling buffering: setvbuf()

- setvbuf() function sets the buffering type of stream to mode:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

- Parameter
  - mode must be one of the following:
    - _IONBF : Unbuffered
    - _IOLBF : Line buffered
    - _IOFBF : Block buffered
  - buf : pointer to buffer (if NULL, buffer is automatically allocated by glibc)
  - size: buffer size in bytes

Controlling Buffering Errors

- Common Mistakes: supplied buffer must exist when stream is closed
  - Declaring buffer automatic variable in a scope that ends before stream is closed.
  - Be careful not to provide a buffer local to main and then fail to explicit close the stream.

```
#include <stdio.h>
void main() {
   char buf[BUFSIZ];   /* ERROR: local to main → global */
    /* set stdout to block-buffered with BUFSIZ buffer */
   setvbuf(stdout, buf, _IOFBF, BUFSIZ);
   printf("Arrr!₩n");
   return(0);
}
```

Thread Safety

- In computer programming, thread-safe describes a program portion or routine that can be called from multiple programming threads without unwanted interaction between the threads
- By using thread-safe routines, the risk that one thread will interfere and modify data elements of another thread is eliminated by circumventing potential data race situations with coordinated access to shared data.
- The standard I/O functions are inherently thread-safe
- Any given thread must acquire the lock and become the owning thread before issuing I/O requests
- Two or more threads operating on the same stream cannot interleave standard I/O operations, and thus within the context of single function calls, standard I/O operations are atomic.

Thread Safety: flockfile()

- Standard I/O provides a family of functions for individually manipulating the lock associated with a stream
- Manual File Locking:
  - The function flockfile() waits until stream is no longer locked, bumps the lock count, and then acquires the lock, becoming the owning thread of the stream, and returns

    ```
    #include <stdio.h>
    void flockfile(FILE *stream);
    ```

Thread Safety: funlockfile()

- funclockfile() decrements the lock count associated with stream:

  ```
  #include <stdio.h>
  void funlockfile(FILE *stream);
  ```

- If lock count reaches zero, the current thread relinquishes ownership of the stream.
- Another thread is now able to acquire the lock

Thread Safety: ftrylockfile()

- The ftrylockfile() function is a nonblocking version of flockfile()

  #include <stdio.h>

  int ftrylockfile(FILE *stream);

- If stream is currently locked, ftrylockfile() does nothing and immediately returns a nonzero value
- If stream is not currently locked, it acquires the lock, bumps the lock count, becomes the owning thread of stream, and returns 0