Files and low level I/O in the UNIX world

* open/close/read/write/lseek/fcnt/ioctl
* Redirection
* File implementation in UNIX
* File stats
* Temporary files
* Relationship between file descriptors and open files
* Buffering and stdio

Files

"On a UNIX system, everything is a file; if something is not a file, it is a process." (minor exceptions   
 apply)

Directories: files containing the names of files in them

Programs, texts, images, videos: files;

either binary or ASCII Devices, e.g. monitor, cpu, gpu, printer: all represented as files

File Types

Graphical user interface, text

Description automatically generated

b 🡪 loop devices c 🡪 serial device

Graphical user interface, text, application, email

Description automatically generated

Files

All files are organized within a file system; a rooted tree of directories

Timeline

Description automatically generated

Unlike windows where each drive has a letter that’s the root of its own FS, in UNIX, drives, partitions, removable media and even network shares can be mounted, and thus the entire volume’s FS appears as a directory. The root is always denoted by / (“slash”)

* /bin: binaries, ls, cp, etc.
* /home : user directories
* /boot: files needed for booting the system
* /etc: system-wide configuration files
* /dev: file representations of devices
* /lib: libraries
* /proc: files representing runtime system information
* /usr: non-system critical binaries, libraries, resources
* /var: logs, temporary files, mail, print jobs, etc.

An arbitrary location or address within this tree structure is known as a path,   
e.g.: /home/ezerger/cse344/week3.pdf

Every directory contains the files: “.” and “..” that represent respectively the current and parent directories.

If a path starts with / then it’s an absolute or fully qualified path, otherwise, the program pretends the absolute path of the current working directory;

* e.g, you are located at: /home/ezerger/cse344
* ../cse685/midterm.pdf -> /home/ezerger/cse685/midterm.pdf

All system calls that deal with files (of any type) refer to them through file descriptors; i.e. a small non-negative integer.

All programs start with 3 open files that are opened on their behalf by the shell:

Table

Description automatically generated

There are four key system calls upon which programming libraries (fopen, fclose, fwrite, etc) rely for file I/O:

* fd = open(pathname, flags, mode)
  + Opens the file pathname and returns its file descriptor
* numread= read(fd, buffer, count)
  + Read at most count bytes from the open file fd and stores in buffer
* numwritten=write(fd, buffer, count)
  + Writes up to count bytes from buffer into the open file fd
* status=close(fd)
  + Is called after all I/O operations are completed, and releases resources
  + fd is not unique, we have 3 of them 🡪 stdin, stdout, stderr for each file descriptor

open

Text

Description automatically generated with low confidence

Flag examples:

* O\_RDONLY: read only; O\_WRONLY: write only
* O\_RDWR: read and write
* O\_CREAT: create a new file
* O\_APPEND: any data written to the file will be appended to its end
* O\_TRUNC: discard previous content, start from beginning
* O\_EXCL: used together with O\_CREAT, returns error if the file already exists
* O\_NONBLOCK: if the file cannot be opened, instead of blocking it returns an error
* ...and more..

Returns the fd or -1 in case of error.

* Way you call the flags or mode is somehow not suitable for that operation or
* Going into path is not possible.

**mode**: is an octal number specifying the permissions of the newly created file (in conjunction with umask and the access permissions of the parent directory).

POSIX defines symbolic names for the permission masks so that you can specify them independently of the underlying implementation (defined in sys/stat.h)

Text

Description automatically generated with low confidence

Text, letter

Description automatically generated

Permissons: user can read, user can write, group can read, group can write, everybody else can only read it.

Flags: We are gonna create the file using read/write and it will be executed.

close

Even though when a process terminates the OS closes all open fd’s associated with that process, it is good practice to close a file once you are done with it.

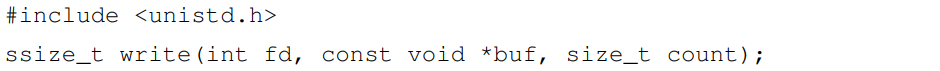


Returns zero on success and -1 on error.

This is also request to OS. Every resources will be released and others will be able to use them.

Open file descriptors use kernel resources (every process needs a table to keep track of its open files). The typical limit is (used to be) 1024 file descriptors per process. You can adjust this limit through the *getrlimit* and *setrlimit* system calls.

write



It writes up to count bytes from the buffer pointed by buf to the current offset of the file referred to by the file descriptor fd. It might write less than count due to a signal interruption, etc. The data to write need not be a character string; it works with arbitrary bytes.

Returns the number of bytes written or -1 on error.

Error examples:

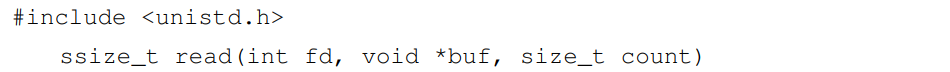
* EBADF: fd is not a valid file descriptor or is not open for writing.
* ENOSPC: the device containing the file referred to by fd has no room for the data.

Text, letter

Description automatically generated

Instead of 0666, you can use mode whose type is mode\_t and give it proper permissions as we did in open example.

read



fd is file descriptor we obtained from open.

Similar to write in principle. Returns the number of bytes read, 0 on EOF, -1 on error.

**Warning**: in the UNIX world file lines are separated by the newline character ‘\n’ (ASCII 10). In the windows world, lines are separated by two characters: a carriage return ‘\r’ (ASCII 13) and a newline character.

So if your file was saved in a windows environment, and you read it in a UNIX environment, do not be alarmed when you see the ^M expression (corresponding to the carriage return character) at the end of every line.

Text

Description automatically generated with medium confidence

Example:

Text, letter

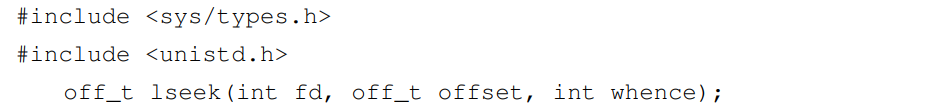
Description automatically generated

Text

Description automatically generated

lseek

A file descriptor remembers its position in a file. As you read or write the position advances depending on the number of bytes read or written. If you want to move arbitrarily within a file then:



offset: new position. The third argument determines how to interpret the second arg.

SEEK\_SET: number of bytes from the start of the file (only positive)

SEEK\_CUR: number of bytes from the current position (positive or negative)

SEEK\_END: number of bytes from the end of the file (positive or negative)

Returns the new position from the beginning. Cannot be used with sockets.

Text, letter

Description automatically generated

Text

Description automatically generated with medium confidence

ioctl

Non standard I/O operations: ioctl

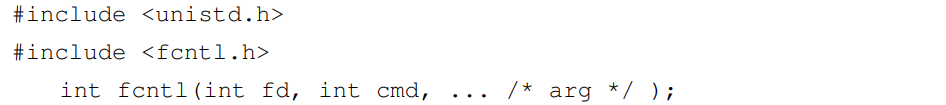
Graphical user interface, text, application, email

Description automatically generated

ioctl is device specific.

fcntl

Advanced file operations: fcntl



It can manipulate the flags associated with a fd (same ones used during opening).

It can duplicate file descriptors

It can lock/unlock files - very useful for inter process communication...

* lock, do what you want to do with the process, then release
* during lock and release, other process will be waiting for you
* BECAUSE YOU WANT TO WRITE ALL MESSAGE BEFORE ANOTHER PROCESS READS IT
* WHEN YOU LOCKED, OTHER PROCESS CAN’T READ IT UNTIL YOU UNLOCKED

To place a lock on a file, first create and zero out a struct flock variable. Set the l\_type field of the structure to F\_RDLCK for a read lock or F\_WRLCK for a write lock. Then call fcntl, passing a file descriptor to the file, the F\_SETLCKW operation code, and a pointer to the struct flock variable. If another process holds a lock that prevents a new lock from being acquired, fcntl blocks until that lock is released.

Table

Description automatically generated with medium confidence

Redirection

Redirecting results of file from one place to another. For each file, there are 3 file descriptors that are opened:

* for reading
* for writing
* for error

If you want to direct one of them output to other or you want to write contents of it to another file (what happens when some error happens), you can redirect them.

Under normal circumstances a process reads from standard input, outputs its result to standard output and in case of error, it is sent to standard error.

We can however redirect them!

In Bourne-style shells: $ ./myscript > results.log 2>&1

i.e. redirect stdout to results.log and redirect stderr to wherever stdout points to; so both stdout and stderr end at results.log

* 0 is stdin
* 1 is stdout
* 2 is stderr

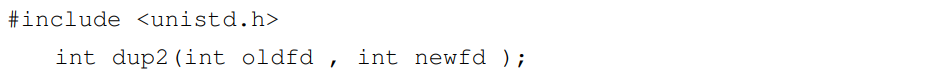
2 tanesi bufferlı (stdin ve stdout)



Both stderr and stdout of “myscript” become stdin of “less”

stdout and stderr of process A is fed into less process.

*How does redirection work behind the scenes?*



* It makes a duplicate of the file descriptor given in oldfd using the descriptor number supplied in newfd. If the file descriptor specified in newfd is already open, it closes it first; e.g.
  + dup2(1, 2)
* first closes stderr, then replaces it with a copy of stdout. Or you can use fcntl instead of dup2:
  + newfd = fcntl(oldfd, F\_DUPFD, startfd);
* Uses as newfd the lowest unused file descriptor greater than or equal to startfd

File implementation

The designers of UNIX have separated file data and meta-data.

All the meta-data concerning a given file reside in a fixed-length structure called inode (short for index node).

The inode contains information about the file size, the file location, the owner of the file, the time of creation, time of last access, time of last modification, permissions and so on.

In addition to descriptive information about the file, the inode contains pointers to the first few data blocks of the file. If the file is large, the indirect pointer is a pointer to a block of pointers that point to additional data blocks. If the file is still larger, the double indirect pointer is a pointer to a block of indirect pointers. If the file is really huge, the triple indirect pointer contains a pointer to a block of double indirect pointers.

Diagram

Description automatically generated

Directory implementation

Directories in UNIX are basically associate arrays of filenames and inode numbers; e.g.

* 1167010 .
* 1158721 ..
* 1167626 subdir
* 132651 barfile
* 132650 bazfile

The inode itself does not contain the filename. When a program references a file by pathname, the operating system traverses the file system tree to find the filename and inode number in the appropriate directory.

Directory contains filenames that are in the directory. When you copy a directory from one place to another, you don’t really copy the contents of the directory if the inode doesn’t change.

* Assume we have 2 directories A and B and A has a.txt in it.
* If I move, text containing a.txt is deleted from A and put it to database of B directory.
* If I copy, you don’t do anything on the disk. You just say whatever pointing a.txt, copy file in B should also point to there. You change the necessary entries on disk directory files. Both A and B link to same place. If somebody change content of a.txt in B, you have to put extra space on disk for that and remove hard link. This is valid only if A and B are on same disk.

Once it has the inode number, the OS can determine other information about the file by accessing the inode.

A directory implementation that contains only names and inode numbers has the following advantages.

1. Changing the filename requires changing only the directory entry. A file can be moved from one directory to another just by moving the directory entry, as long as the move keeps the file on the same partition.
2. Only one physical copy of the file needs to exist on disk, but the file may have several names or the same name in different directories. Again, all of these references must be on the same physical partition.
3. Directory entries are of variable length because the filename is of variable length. Directory entries are small, since most of the information about each file is kept in its inode. Manipulating small variable-length structures can be done efficiently. The larger inode structures are of fixed length.

Links

UNIX directories have two types of links—links and symbolic links

* A link is an association between a filename and an inode, sometimes called a hard link,
* A symbolic link, sometimes called a soft link, is a file that stores a string used to modify the pathname when it is encountered during pathname resolution

Each inode contains a count of the number of hard links to the inode.

When a file is created, a new directory entry is created an a new inode is assigned.

Additional hard links can be created with (no file generation, all is done is increasing the number of links on that specific file by one)

ln newname oldname

or with (newpath and oldpath will be directing to same file on the buffer or on the disk)

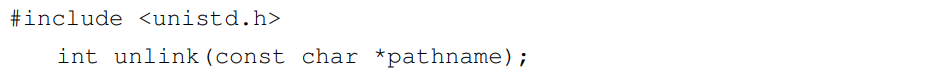
#include <unistd.h>

int link(const char \*oldpath, const char \*newpath);

A new hard link to an existing file creates a new directory entry but assigns no other additional disk space.

A new hard link increments the link count in the inode.

A hard link can be removed with the rm command or the unlink system call:



• These decrement the link count.

• The inode and associated disk space are freed when the count is decremented to 0.

Even when inode increment for that file comes to 0, it is not actually deleted. File is still on there but space is available for other users to use it. That’s why in most OS you can use undelete. It goes to file entry and increments associated link count to 1. Now file comes back. If somehow nobody has touched that place, you can recover your file completely.

A symbolic link is a special type of file that contains the name of another file.

• A reference to the name of a symbolic link causes the operating system to use the name stored in the file, rather than the name itself.

• Symbolic links are created with the command:

A picture containing text

Description automatically generated

• Symbolic links do not affect the link count in the inode.

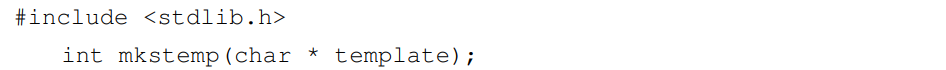
• Unlink hard links, symbolic links can span filesystems.

Softlink creates another file that points initial file that is created. Softlink has size on disk while hardlink doesn’t have size on disk. Hardlinks point to same file on disk. Softlink points to actual file that was initially linked. You can use unlink for both of them.

Temporary files

Some programs need to create temporary files that are used only while the program is running, and these files should be removed when the program terminates.

The mkstemp call generates a unique filename based on a template supplied by the caller and opens the file, returning a file descriptor that can be used with I/O system calls.



Returns the file descriptor on success, or -1 on error. If success, you can do write read, it is a normal file.

Typically, a temporary file is unlinked (deleted) soon after it is opened, using the unlink system call

Text, letter

Description automatically generated

Her şey direkt hardware’e yazılmaz. OS sana bir miktar buffer ayırır. “fd = mkstemp(template);” çağrısı sonunda sana 2 buffer – 3 file descriptor geri geldi. Hala o bufferlara yazıp okuyabilirsin. Ama buffer size limitli. Çok yazarsan buffer dolar ve bufferda bulunan kadarına ulaşabilirsin. Ama temporary işleri halen orada yapabilirsin.

Okuduğun zaman direkt diskten de okumuyorsun. O dosya için hazırlanmış olan stdouta gidiyorsun ve onun bufferından okuyorsun. Böyle olunca en son yapılan değişikliklere diske yazılmadan ulaşırsın.

Bir process dosya açtı ve read write işlemleri yapıyor diyelim. Eğer dosya gerçekten diskte olan bir dosyaysa, her şey direkt oraya yazılmaz. OS sana 2 buffer hazırlar: stdout ve stdin. Sen onlara yazarsın. Değişiklik hala onlarla yapılır. Belli size’a gelince OS artık bufferı diske yazayım, contentini boşaltıyım der ve diske yazar. Diske yazılırken biraz zaman geçer. Bufferdan okunurken o kadar zaman kullanılmıyor. Bu yüzden diske yazım işlemini baştan tahmin edemezsin. Çünkü OS buffer’a yazıp geri dönüyor olabilir veya buffer yeterince doldu gideyim diske yazayım diyebilir. OS gerçekten diske mi yazıyor, buffera mı yazıyor bilemeyiz ayrıca ne zaman yazacağını da bilemeyiz. Bu yüzden zamanı bilemeyiz.

File stats

You can access the inode information through the stat calls

You can change the status of files.

Text, letter

Description automatically generated

* stat works with filenames
* fstat works with file descriptors
* lstat is similar to stat, except that if the named file is a symbolic link, information about the link itself is returned, rather than the file to which the link points

If you have permissions to do anything with that file, you change all things below.

Text

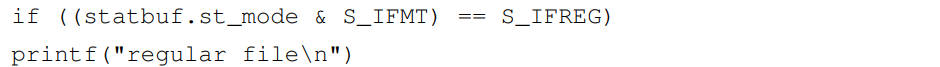
Description automatically generated

Graphical user interface, text, application, table

Description automatically generated

Permissions can also be changed.

The file type can be extracted by AND’ing (&) with the constant S\_IFMT.



But since this is a common operation there are macros for it.

A picture containing text

Description automatically generated

Table

Description automatically generated

You cant see hidden files with “ls -l”. You should use “ls -al” for that.

ACCESS MODES 🡪 E/G/U

Relationship between open files and processes

There is no one-to-one correspondence between file descriptors and open files. It is possible and useful to have multiple descriptors referring to the same open file. These file descriptors may be open in the same process or in different processes.

The kernel maintains 3 data structures

* the per-process file descriptor table (with fd flags);
* the system-wide table of open file descriptions (offset, inode, signal settings, etc)
* the file system i-node table (file type, owner, location, etc)

Diagram

Description automatically generated

In process A, descriptors 1 and 20 both refer to the same open file description (labeled 23). This situation may arise as a result of a call to dup2, or fcntl. Birinin descriptorını diğerine map ediyorsun.

Descriptor 2 of process A and descriptor 2 of process B refer to a single open file description (73). This scenario could occur e.g. if process A is the parent of process B, or vice versa), or if one process passed an open descriptor. 2 ayrı process içinde aynı dosyayı açıyorsun.

Finally, we see that descriptor 0 of process A and descriptor 3 of process B refer to different open file descriptions, but that these descriptions refer to the same i-node table entry (1976)-i.e. to the same file. This occurs because each process independently called open() for the same file. A similar situation could occur if a single process opened the same file twice. Aynı dosyayı birden fazla açıp diğerini diğer processe assign ediyorsun.

● Two distinct fd’s of the same file share the same offset

● File descriptor flags (i.e., the close-on-exec flag) are private to the process and file descriptor

Buffering

When you open a file 3 file descriptors (stdout, stdin, stderr) are associated to your process. Most of the time when you do write and read, kernel doesn’t have to go to hardware. It would be time consuming. We use buffering for that.

When working with disk files, the read() and write() system calls don’t directly initiate disk access. Instead, they simply copy data between a user-space buffer and a buffer in the kernel buffer cache. For example, the following call transfers 3 bytes of data from a buffer in user-space memory to a buffer in kernel space:

write(fd, "abc", 3);

At this point, write() returns. At some later point, the kernel writes (flushes) its buffer to the disk. (Hence, we say that the system call is not synchronized with the disk operation.) If, in the interim, another process attempts to read these bytes of the file, then the kernel automatically supplies the data from the buffer cache, rather than from (the outdated contents of) the file (a similar scenario is valid for read).

The kernel performs the same number of disk accesses, regardless of whether we perform 1000 writes of a single byte or a single write of a 1000 bytes. However, the latter is preferable, since it requires a single system call, while the former requires 1000. Although much faster than disk operations, system calls nevertheless take an appreciable amount of time, since the kernel must trap the call, check the validity of the system call arguments, and transfer data between user space and kernel space.

Let’s look at the effect of the buffer size on duplicating a large file.

Table

Description automatically generated

Buffering and File pointers

Buffering of data into large blocks to reduce system calls is exactly what is done by the C library I/O functions (e.g., fprintf(), fscanf(), fgets(), fputs(), fputc(), fgetc()) when operating on disk files. Thus, using the stdio library relieves us of the task of buffering data for output with write() or input via read().

Be careful as these higher level functions handle files in terms of pointers to FILE structures.

For all the operations you use on read-write/files, they are all buffered except for the errors. They have to be displayed instantly to invoke the programmer that sth bad happened.

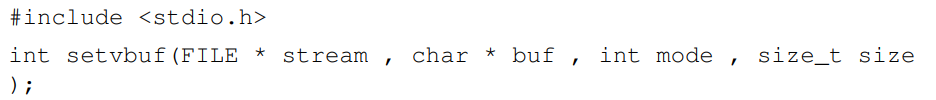
the FILE structure

Text

Description automatically generated

Buffering

The setvbuf() function controls the form of buffering employed by the stdio library.



Returns 0 on success, or nonzero on error. Valid for all subsequent stdio operations.

stream : the stream upon which the buffering will be applied

buf: the buffer

size: size of the buffer in bytes

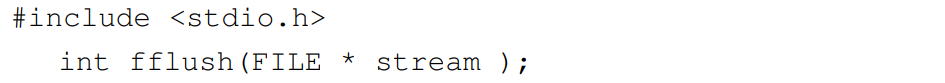
mode can be one of the following:

* \_IONBF: no buffering, immediate reads/writes, default of stderr
* \_IOLBF: Employ line-buffered I/O. This flag is the default for streams referring to terminal devices. For output streams, data is buffered until a newline character is output (unless the buffer fills first). For input streams, data is read a line at a time.
* \_IOFBF: Employ fully buffered I/O. Data is read or written (via calls to read() or write()) in units equal to the size of the buffer. This mode is the default for streams referring to disk files.

A picture containing text

Description automatically generated

Regardless of the current buffering mode, at any time, we can force the data in a stdio output stream to be written (i.e., flushed to a kernel buffer via write()) using the fflush() library function. This function flushes the output buffer for the specified stream. All write and read operations that are buffered will be written on disk or read will be done from disk.



Returns 0 on success, EOF on error

Diagram

Description automatically generated

Even if fflush operation is sent as a request to kernel, kernel will synchronize its buffered contents with actual disk.