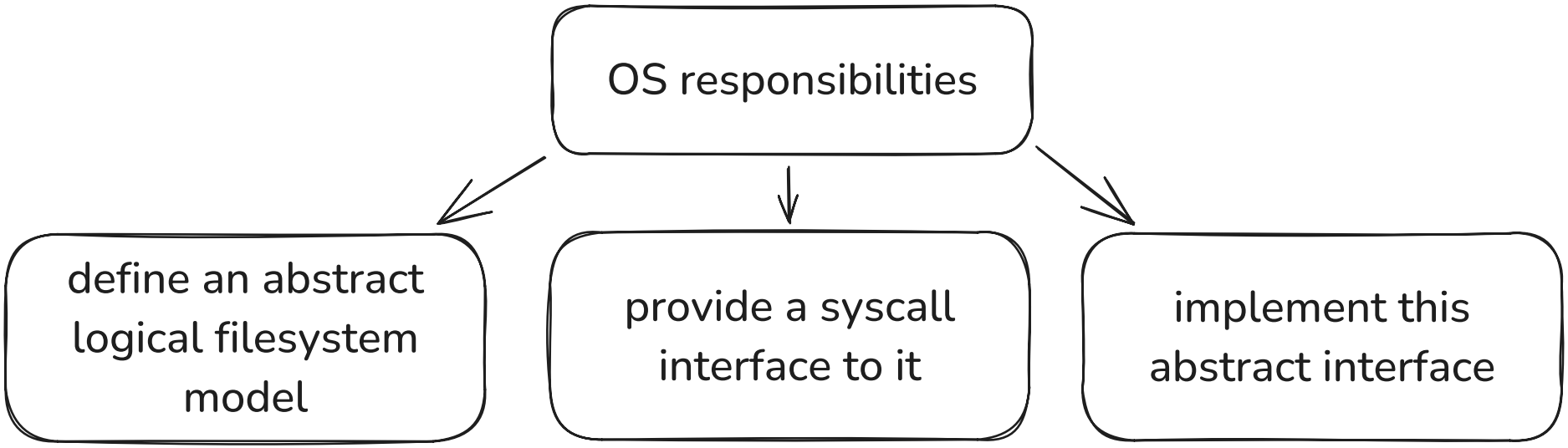
Lecture 2 - Filesystem API

Operating Systems 1

Warsaw University of Technology - Faculty of Mathematics and Information Science

User programs access the filesystem to store data on a persistent storage.



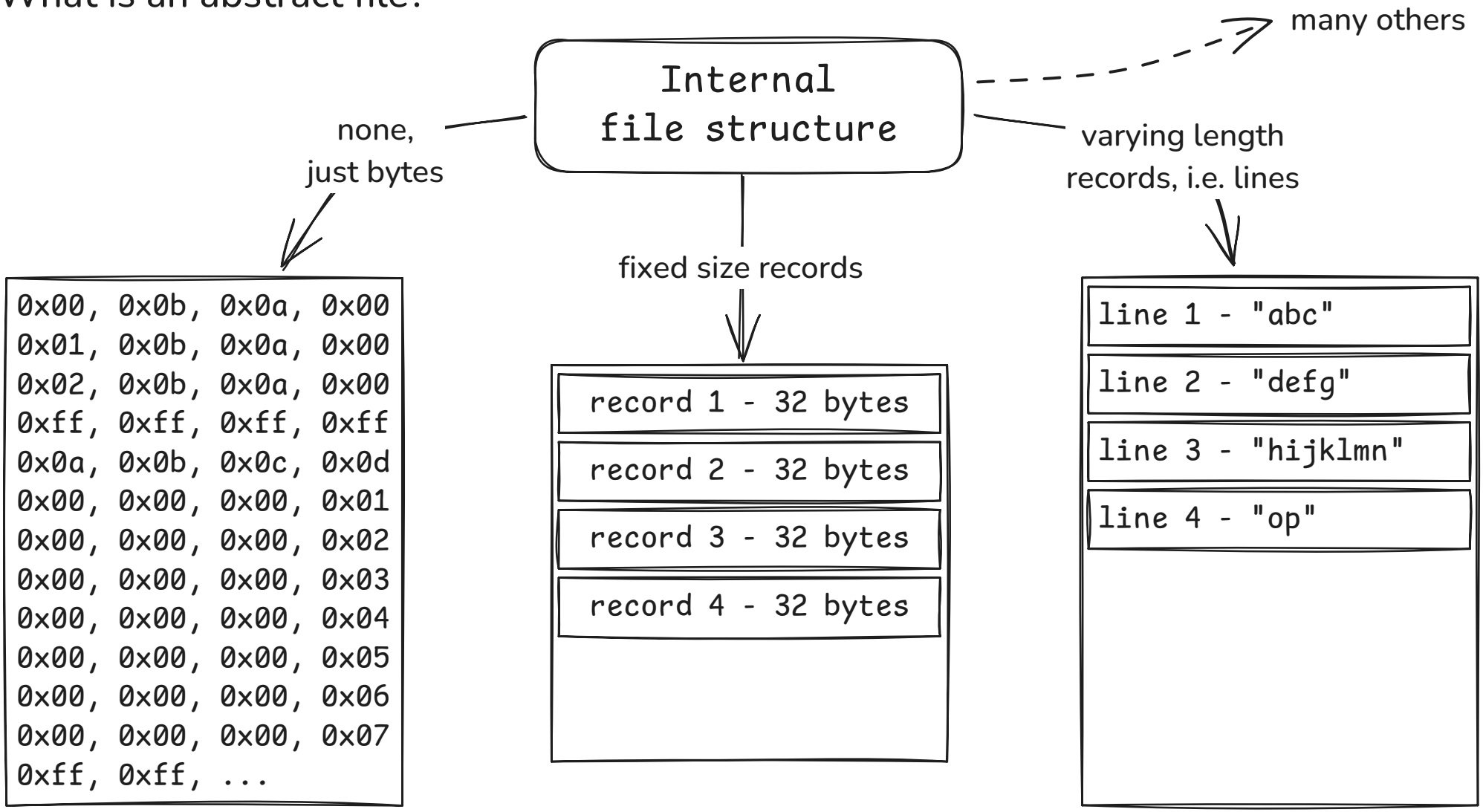
- what is a file?
- how it looks like inside?
- what are file attributes?
- are files structured? how?

- create file
- open/close file?
- delete file
- read/write from file
- create directories?

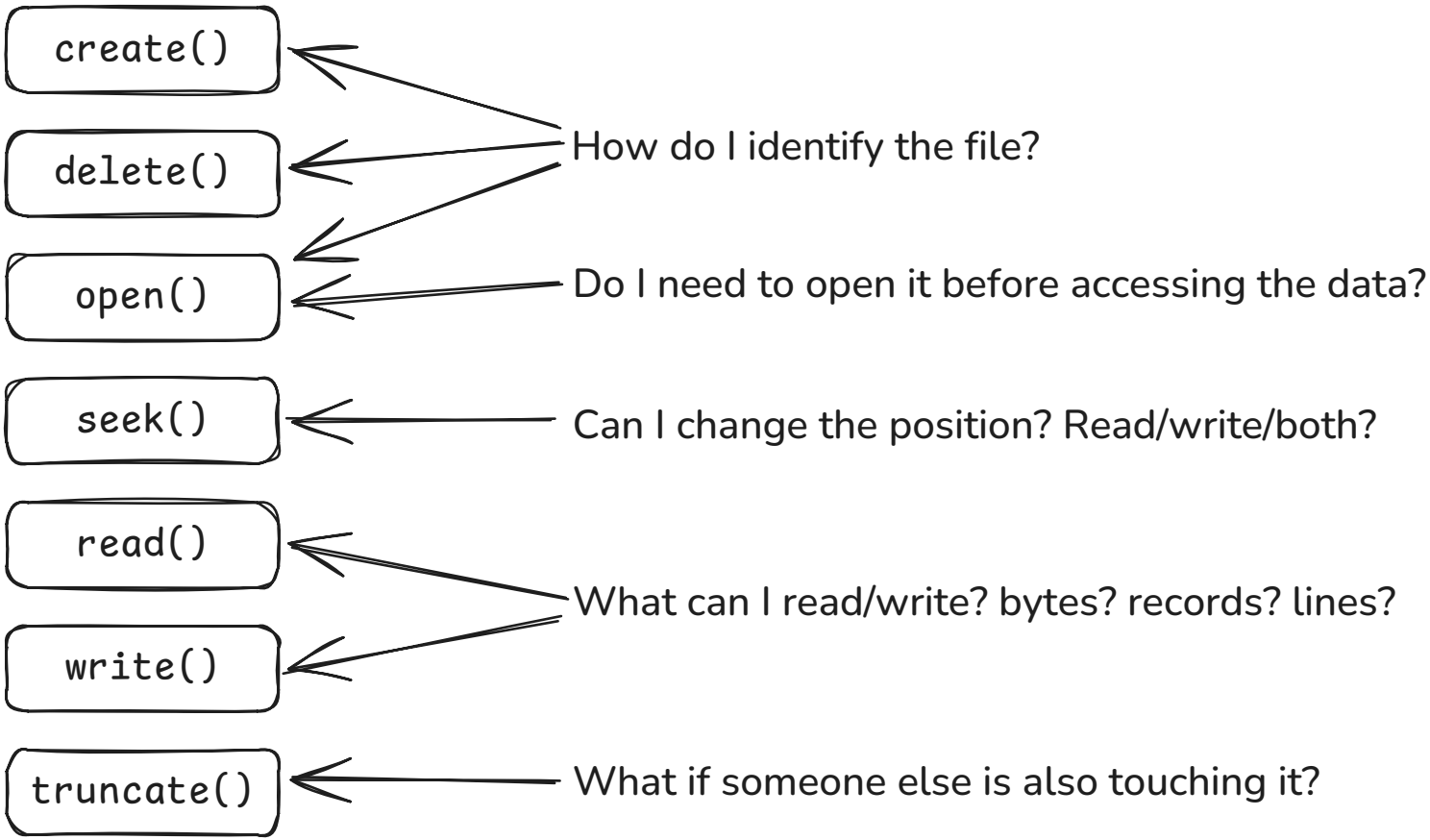
...

- map the logical model to a physical representation
- efficiently
- safely
- using variety of hardware

What is an abstract file?



What can I do to it?



How do I access the actual data?

Sequential access

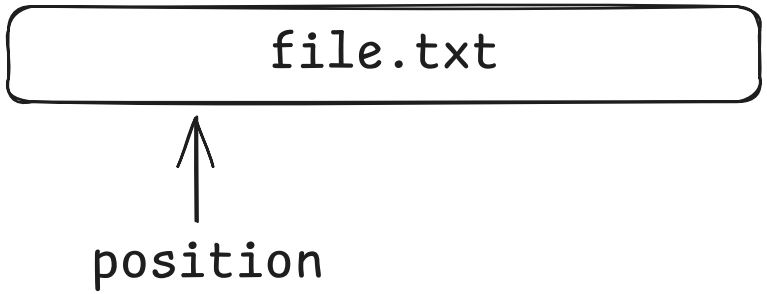
```
open(file)
read(file, len)
write(file, len)
seek(file, offset)
```

OS maintains position within file

Random (direct) access

```
read(file, offset, len)
write(file, offset, len)
```

Here it's an application responsibility

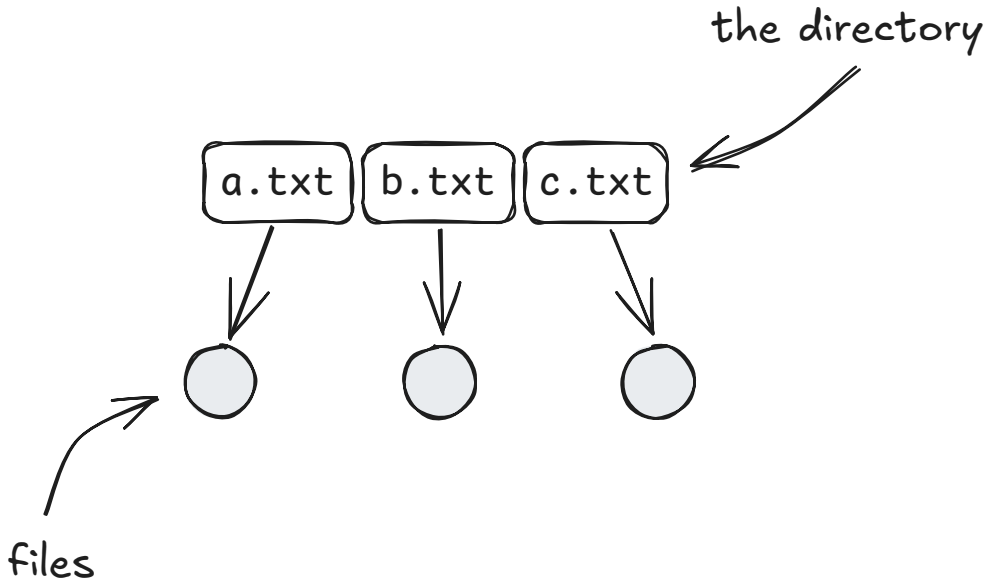


What attributes does file have beyond data?

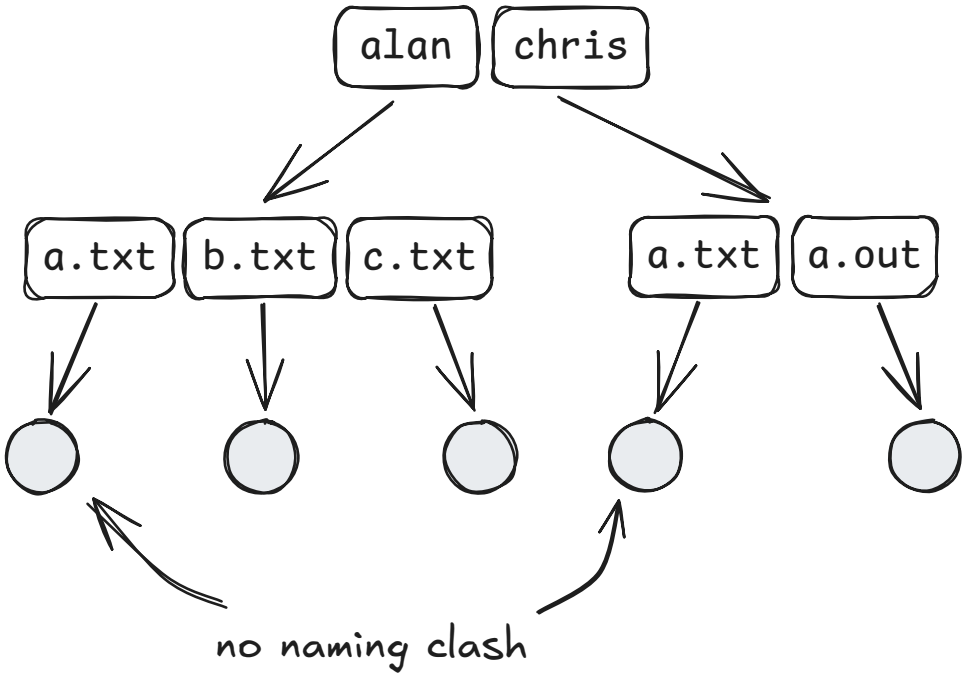
`name`	might also be a kept within the directory
`identifier`	does not change with renames
`type`	only if OS supports distinct file types
`location`	physical storage information
`size`	logical and physical
`protection`	who is the owner? who can access it?
`timestamps`	when and how was it used?

How are files organized?

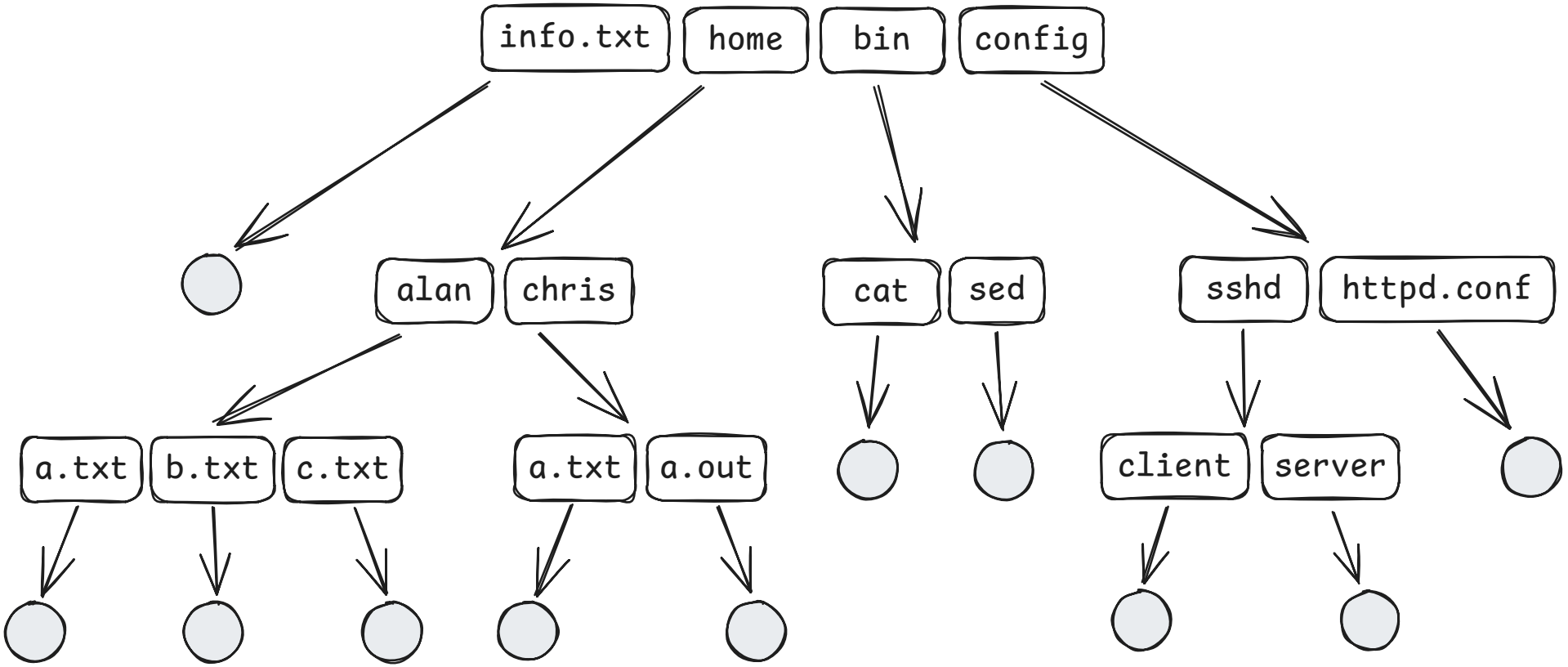
Flat - single directory in FS



2-level - Per-user directory

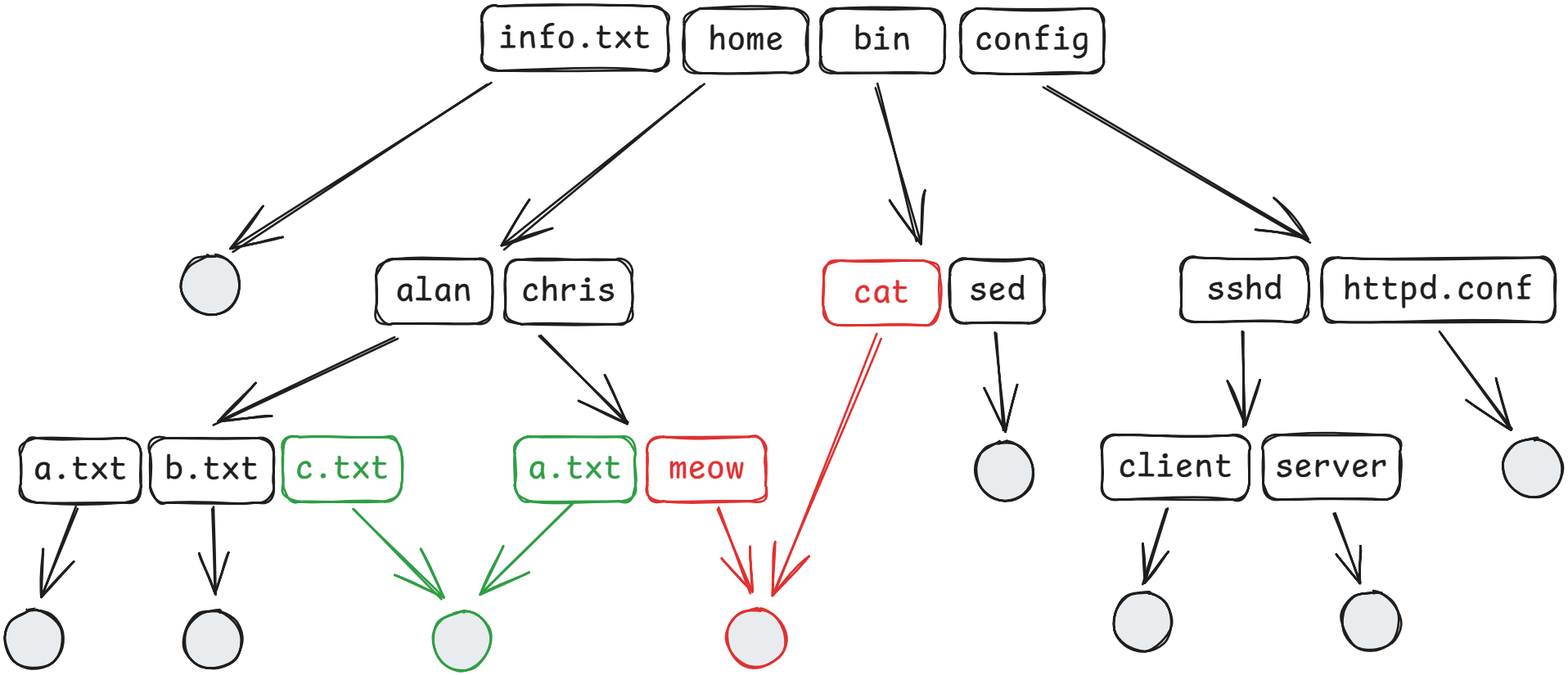


Multi level - tree structure



traversal is fast, we can speak of relative and absolute paths now

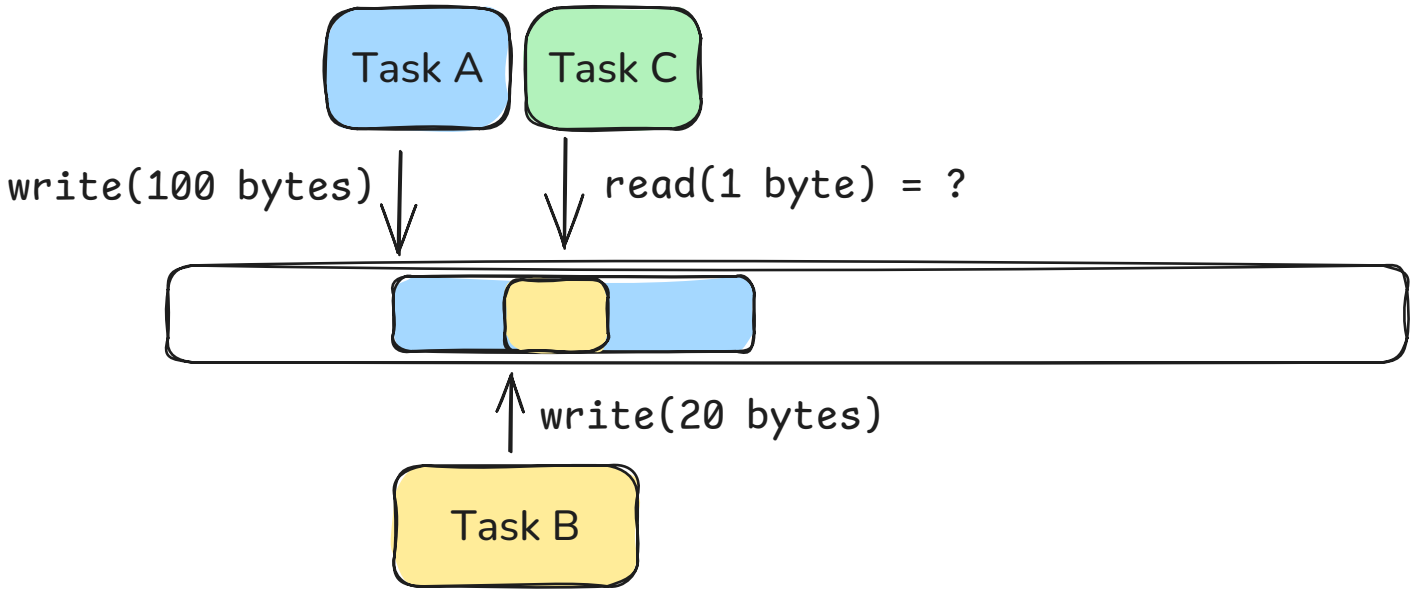
Acyclic graph structure



files can have multiple links (and paths), directories cannot!

Sharing semantics

In a multiprogramming scenario OS has to define how concurrent operations behave



The system might simply disallow concurrent access - simple but inefficient

Otherwise it must at least protect the FS against corruption

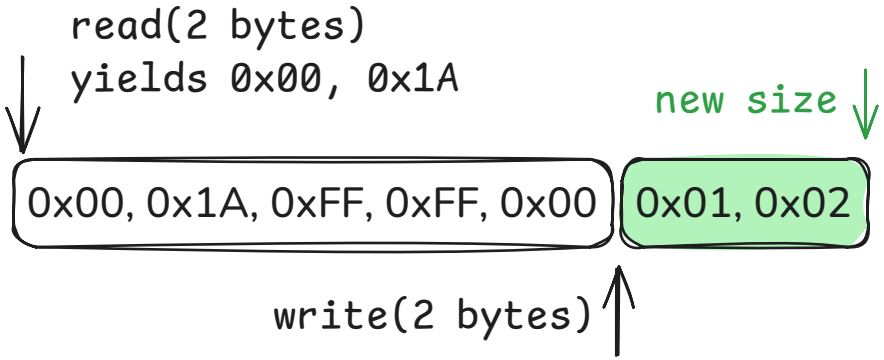
Writes might become visible immediately or after closing the file session

The POSIX filesystem

A concrete example of an abstract interface

File - an object that can be written to, or read from, or both.

files are just bytes
and grow when necessary

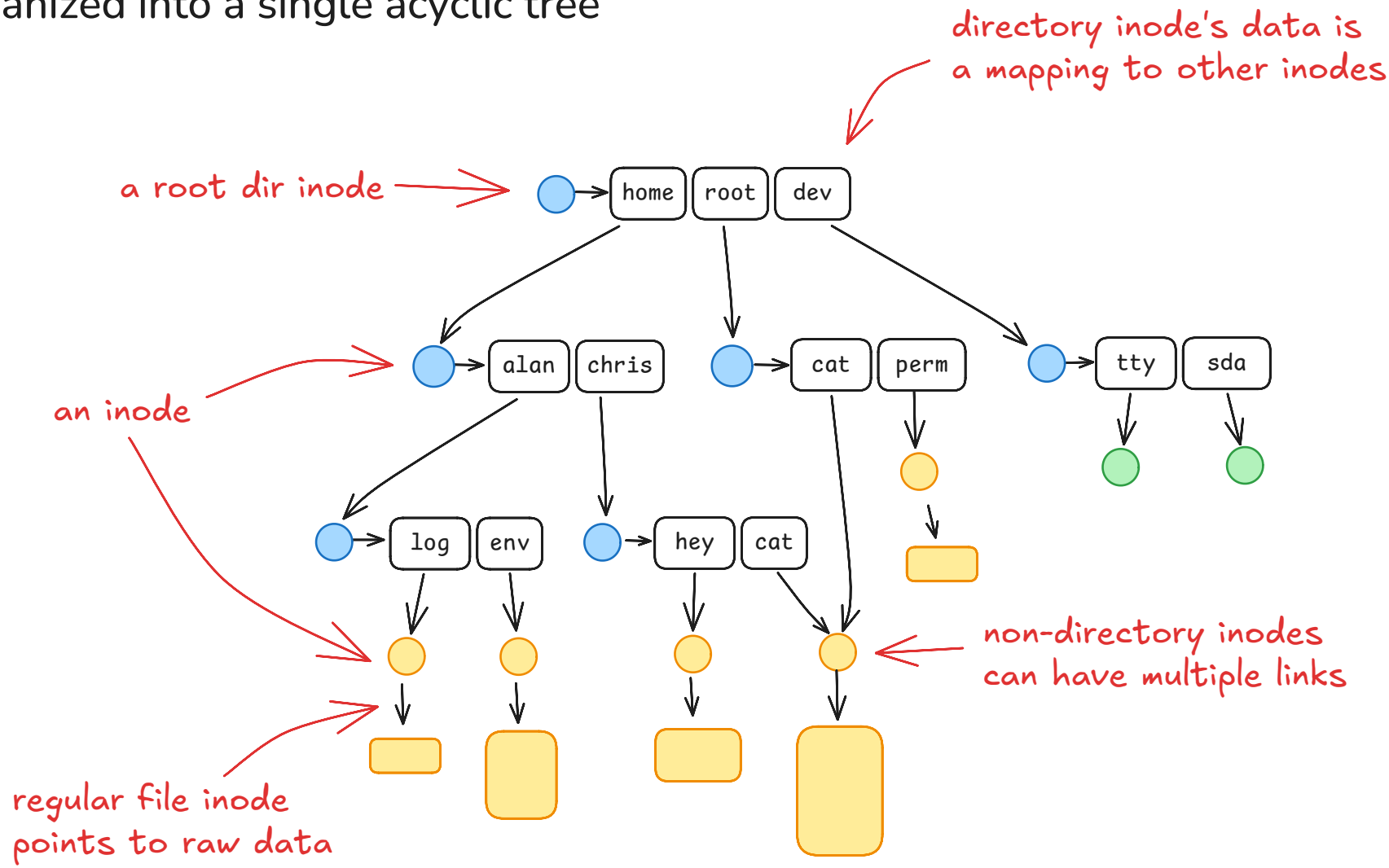


files require opening before any I/O

```
fd = open(file)
read(fd, buffer, size)
write(fd, buffer, size)
close(fd)
```

The system maintains file position per open file session
There's both sequential and random access API

It's organized into a single acyclic tree



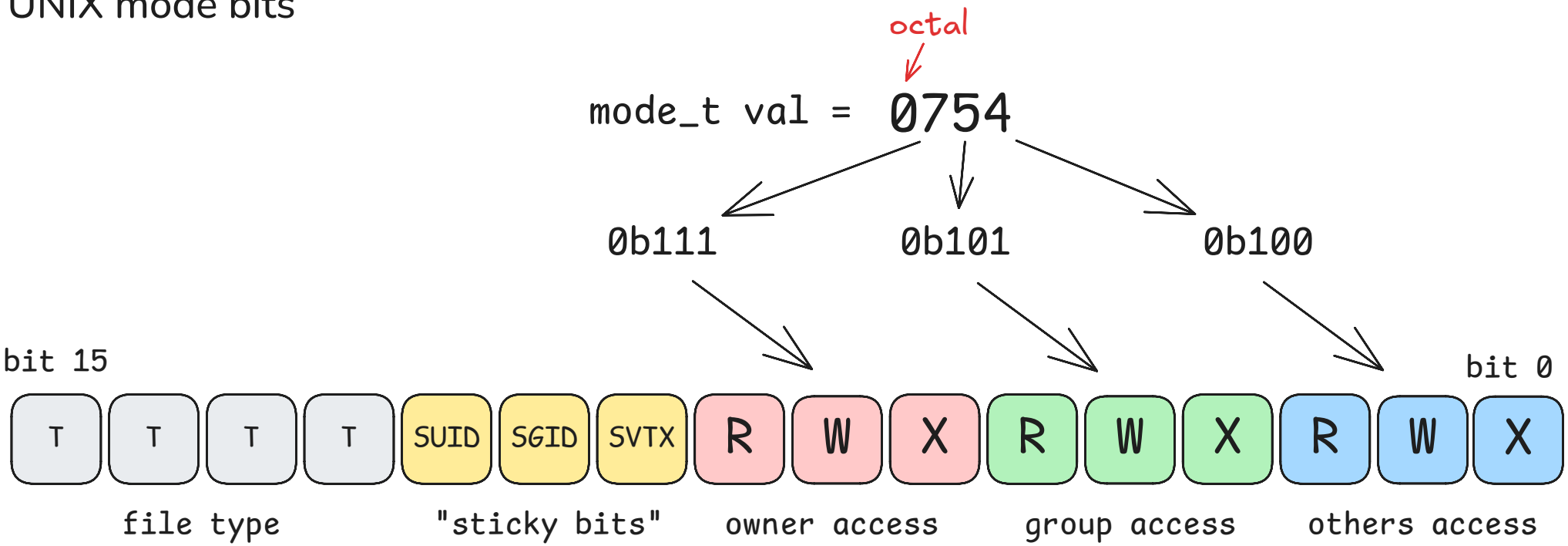
POSIX file attributes

```
> man 7 inode
```

Each file has an inode containing metadata about the file

- Device ID - ID of the device (FS) which this Inode belongs to
- Inode number - FS unique ID of the file
- File type and mode - a mode_t value encoding permissions and file type
- Link count - a reference counter
- User ID - owner UID
- Group ID - owning group UID
- Represented device - only if file is a special device file
- File size - logical, in bytes
- Preferred block size - for most efficient I/O
- Number of blocks - number of allocated device blocks
- Various Timestamps - access/modification/(birth)/status change

UNIX mode bits



You can freely set SUID, SGID, SVTX with another octal digit.

You cannot set the file type but you can check it by inspecting the highest bits.

POSIX file types

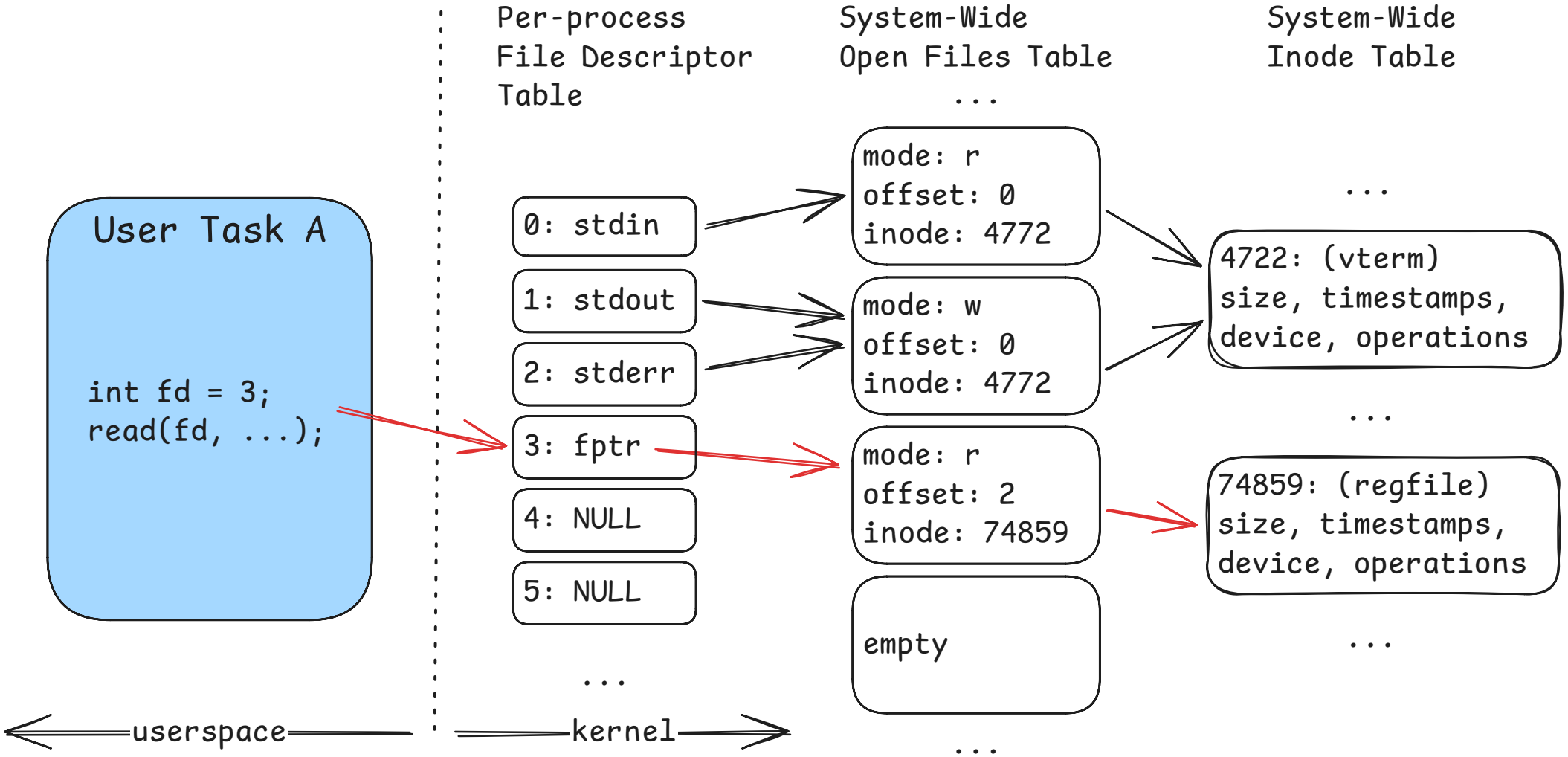
```
The following mask values are defined for the file type:

S_IFMT 0170000 bit mask for the file type bit field

S_IFSOCK 0140000 socket
S_IFLNK 0120000 symbolic link
S_IFREG 0100000 regular file
S_IFBLK 0060000 block device
S_IFDIR 0040000 directory
S_IFCHR 0020000 character device
S_IFIFO 0010000 FIFO
```

this is what you usually expect

All of the above may be read from, written to, or both but these operations are not necessarily mapped to hard drive operations

File session as seen by the kernel

```
> ls -l /proc/<pid>/fd(info)
```



The open() syscall

```
> man 3p open
```

```
int fd = open(const char *pathname, int oflag, ...);
```

The open flags determine how read/write's shall later behave

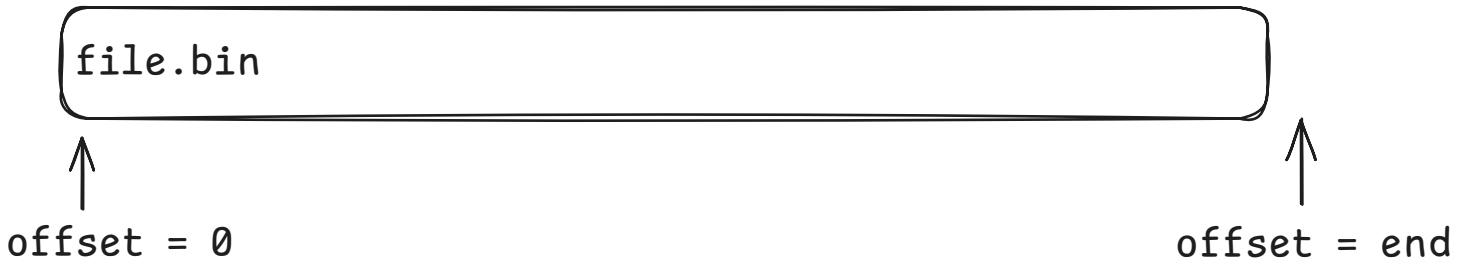
```
O_RDONLY - read only access, write() won't work
O_WRONLY - write only access, read() won't work
O_RDWR - read-write access, both will work
```

O_RDONLY | O_WRONLY means nothing !

The returned file descriptor is a unique per-process identifier of an open file passed to following file-maniupulation syscalls.

Typically the shell automatically opens file descriptors 0, 1 and 2 (stdin, stdout and stderr) for each process.

File position

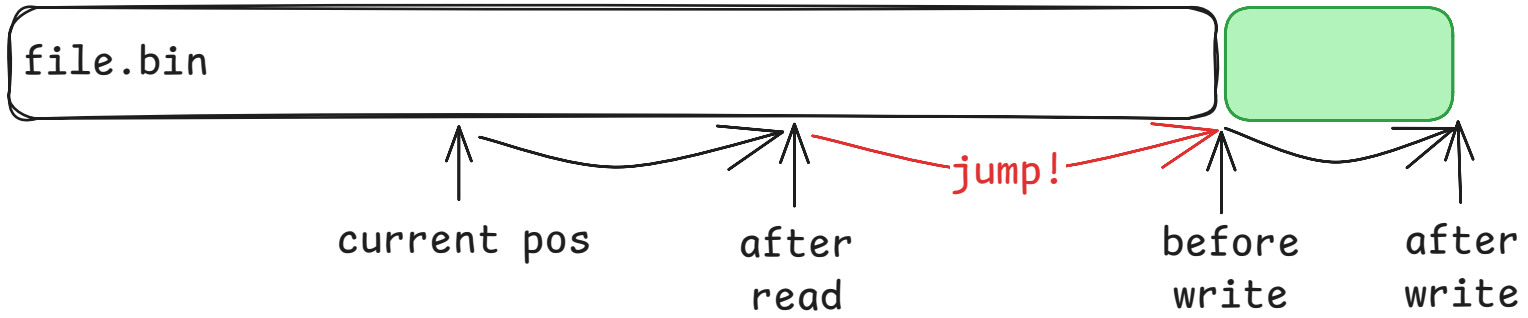


Newly opened file descriptors are positioned at the offset 0 (unless O_APPEND is used).

Files size automatically grows with each write that would go past the end of file.

With O_APPEND position is automatically changed to the end prior to each write().

```
fd = open("file.bin", O_RDWR | O_APPEND);
read(fd, ...);
write(fd, ...);
```



Open mode modifiers

```
O_APPEND - write() always appends at the end
O_TRUNC - truncate file size to 0 bytes too
O_NONBLOCK - read/write won't block, even if operation can't complete immediately
O_CREAT - create file if not exists, requires mode parameter (...)
O_EXCL - use with O_CREAT, fail if file already exists
```

Open file.txt previously creating it if it doesn't exist:

we do not know if the file was created or not

```
open("file.txt", O_RDWR | O_CREAT, 0744);
```

oh look, the 3rd argument, required with O_CREAT

Create and open pid.lock file but fail if it is already there:

```
open("pid.lock", O_RDWR | O_CREAT | O_EXCL, 0666);
```

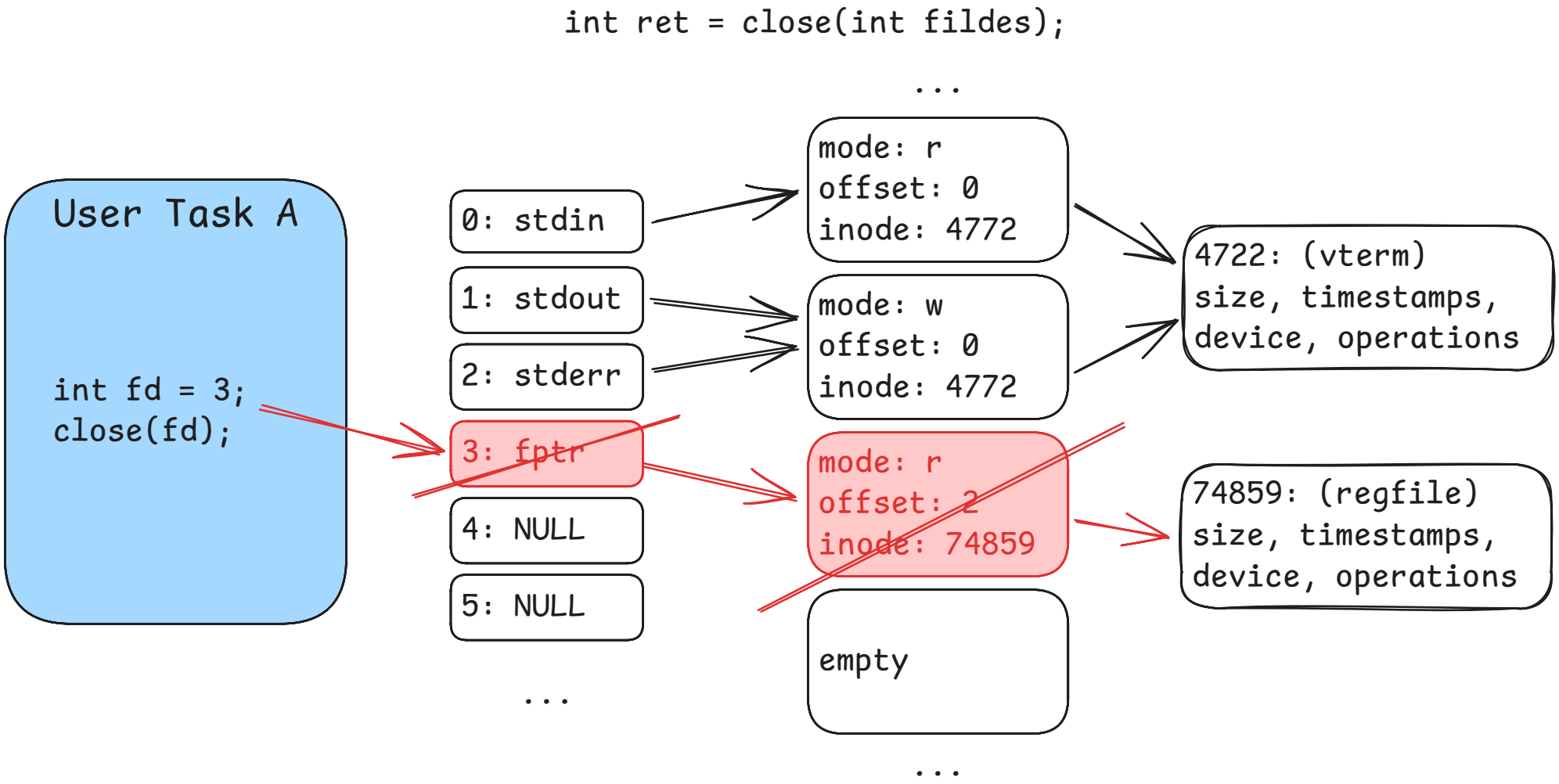
this is atomic, only one process will succeed in creating the pid.lock

Bits set in umask are subtracted from mode bits supplied to open(). `> man 3p umask`

Closing the file session

```
> man 3p close
```

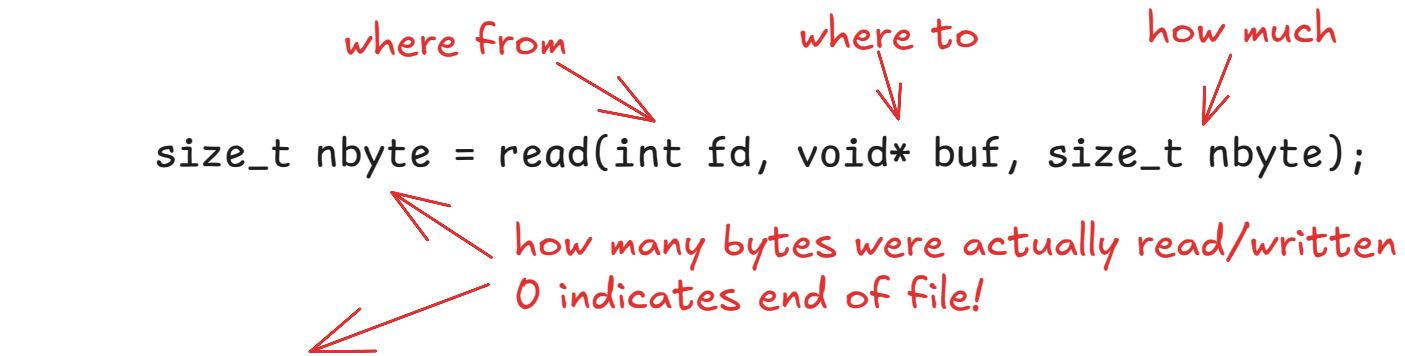
```
int ret = close(int fildes);
```



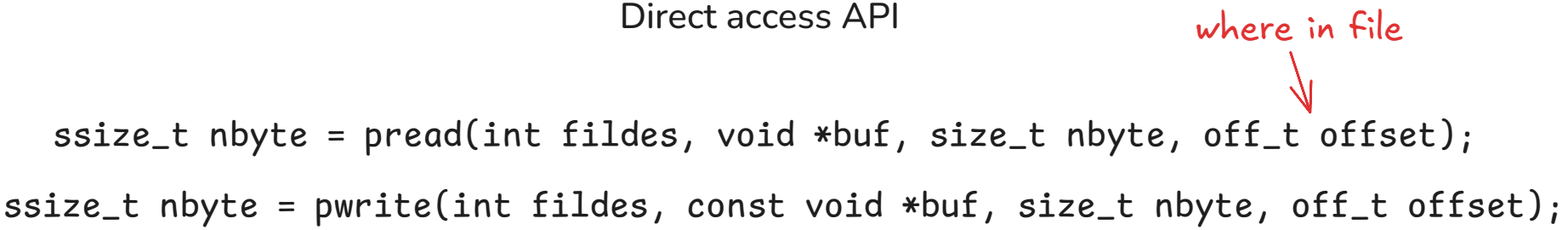
Synchronous I/O syscalls

```
> man 3p read
> man 3p write
```

Sequential access API



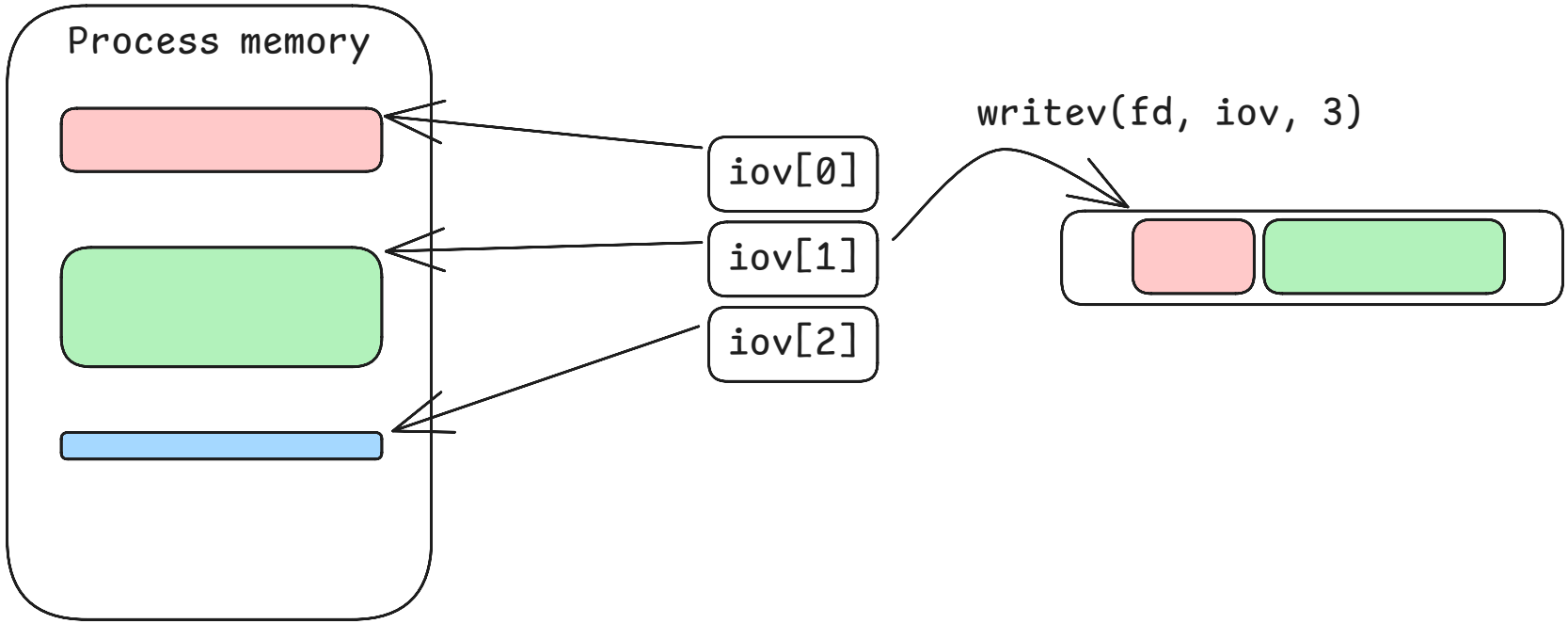
Direct access API



Scatter-gather I/O syscalls

```
> man 3p readv
> man 3p writev
```

```
ssize_t nbytes = readv(int fildes, const struct iovec *iov, int iovcnt);
ssize_t nbytes = writev(int fildes, const struct iovec *iov, int iovcnt);
```



Obtaining file attributes

```
> man 3p fstatat
```

```
int fstatat(int fd, const char *restrict path,
struct stat *restrict buf, int flag);
int lstat(const char *restrict path, struct stat *restrict buf);
int stat(const char *restrict path, struct stat *restrict buf)
```

lstat() does not follow symlinks

path or open fd

where to put attributes

These functions just return information from the i-node structure sitting in the kernel space.

Moving through the file

Changing the system-maintained file position

```
> man 3p lseek
```

```
off_t pos = lseek(int fildes, off_t offset, int whence);
```

Seek mode:

```
SEEK_SET - offset is relative to the beginning
SEEK_CUR - offset is relative to current position
SEEK_END - offset is relative to file end
```

This call might also be used to find out current position.
Seek allows jumping past the logical end of file.

File descriptor cloning

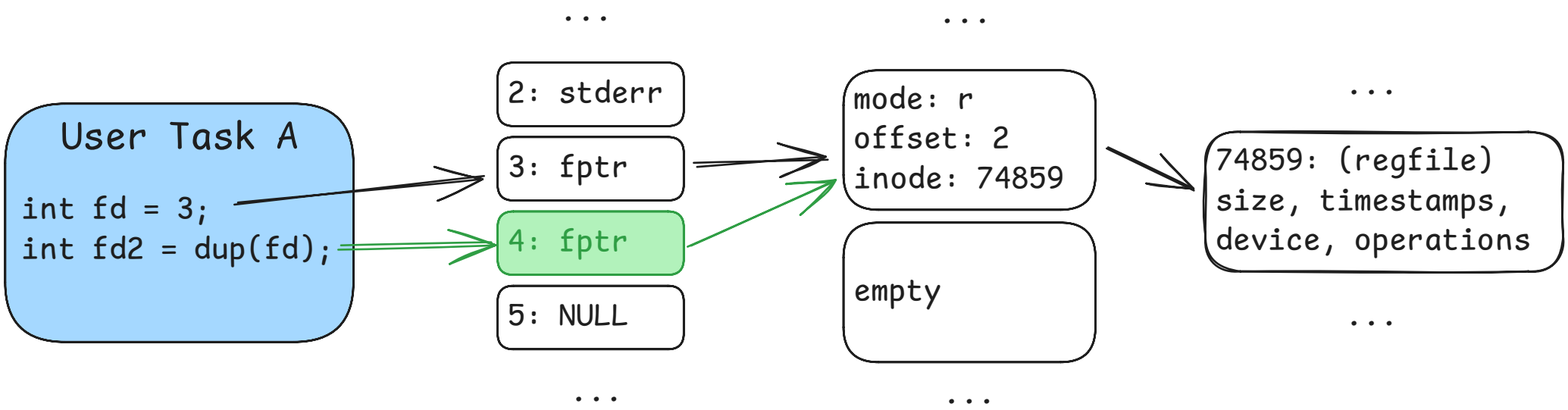
```
> man 3p dup
```

```
int fd = dup(int fildes);
int fd = dup2(int fildes, int fildes2);
```

new fd refering to same thing as filedes

what to clone

desired clone value



Directory operations

```
> man 3p readdir
```

Directories usually cannot be accessed via regular open/read since they have a record structure.

an opaque type representing a directory stream

```
DIR *opendir(const char *dirname);
int closedir (DIR *dirstream);

struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *restrict dirp,
struct dirent *restrict entry,
struct dirent **restrict result);

void rewinddir (DIR *dirstream);
int telldir (DIR *dirstream);
void seekdir (DIR *dirstream, long int pos);
```

returns a pointer to a library allocated storage

a thread-safe version

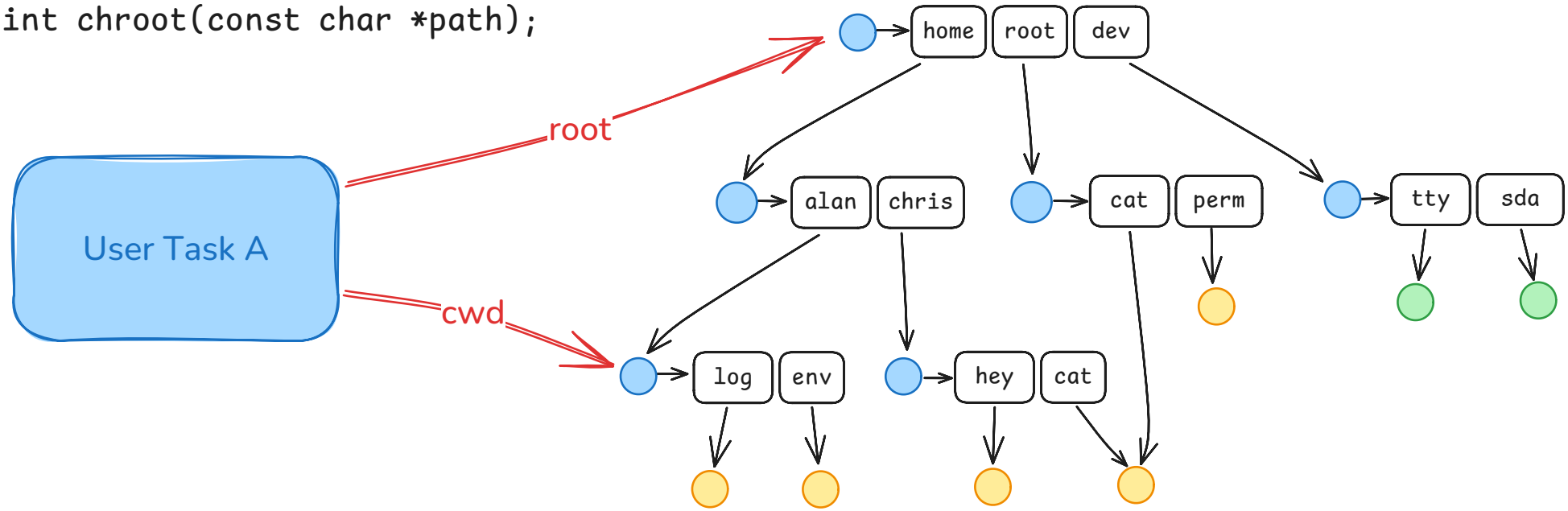
equivalent of lseek()

Each directory entry contains a name and an i-node number.

Process root and current working directory

```
> man 3p chdir
> man 3p getcwd
> man 2 chroot
```

```
int chdir(const char *path);
char *getcwd(char *buf, size_t size);
int chroot(const char *path);
```



Relative paths are resolved starting from cwd attribute.

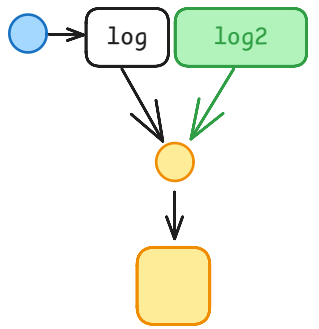
File link operations

```
> man 3p link
> man 3p unlink
```

```
int link(const char *path1, const char *path2);
```

Creates a new directory entry path2 pointing to the same i-node as path1.
Increments a reference count stored in the i-node.

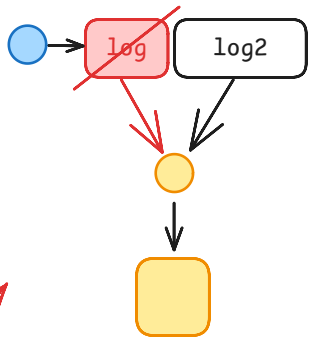
link("log", "log2")




```
int unlink(const char *path);
```

Removes a new directory entry path pointing to some i-node.
Decrements a reference count stored in the i-node.
If it drops to 0 then the i-node and associated data is removed.

unlink("log")



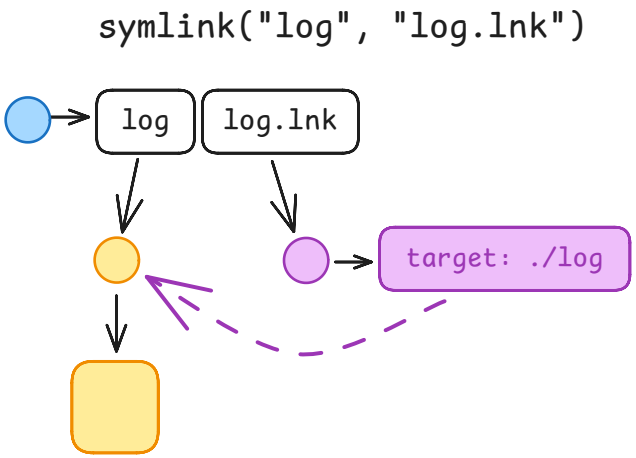
note: this is a non-atomic rename operation
use rename() to get atomicity

Symlink operations

```
> man 3p symlink
```

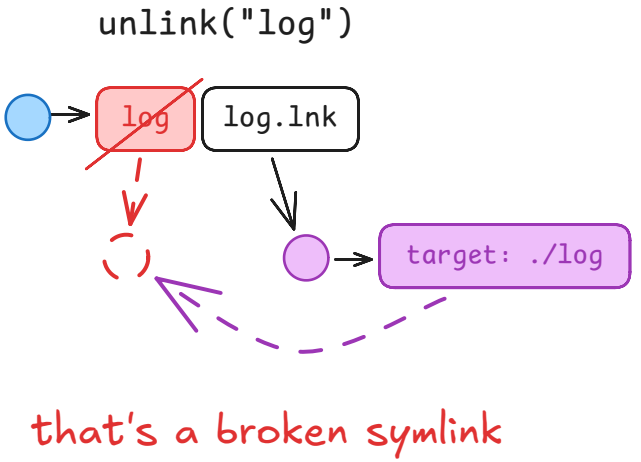
```
int symlink(const char *path1, const char *path2);
```

Creates a new directory entry pointing to a new symlink i-node.
The symlink internally points to path1, which may or may not exist.
Does not increment a reference count stored in the target i-node.




```
int readlink(const char *filename, char *buf, size_t size)
```

Obtains target path stored in the symlink.



Standard C streams

A more portable, weaker API wrapping POSIX filesystem API

```
> man 3p fopen
> man 3p fclose
```

not a syscall!

```
FILE *fopen(const char *restrict pathname, const char *restrict mode);
```

returns an opaque structure pointer

mode: r|w|a|r+|w+|a+

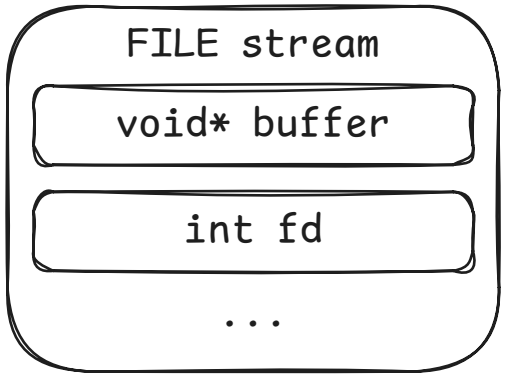
A stream wraps low-level file descriptor and translates I/O operations into read/write syscalls.

Stream operations are buffered to reduce syscall overhead.

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

Closing a stream writes any pending data, deallocates the buffer, and closes the associated file descriptor.

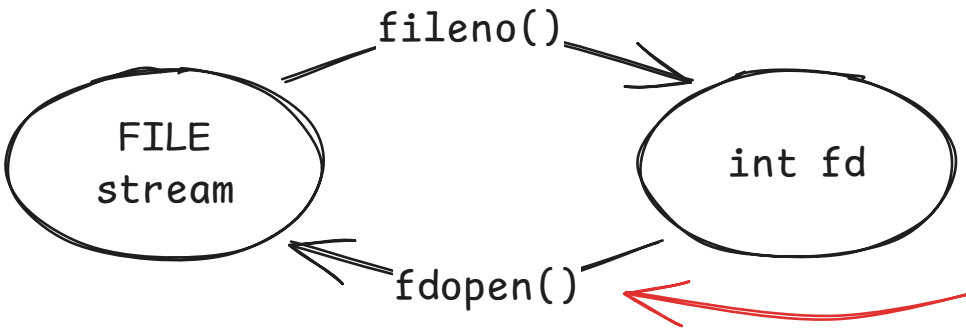
C runtime automatically creates 3 streams: stdin, stdout and stderr.



Mixing API

```
> man 3p fileno
> man 3p fdopen
```

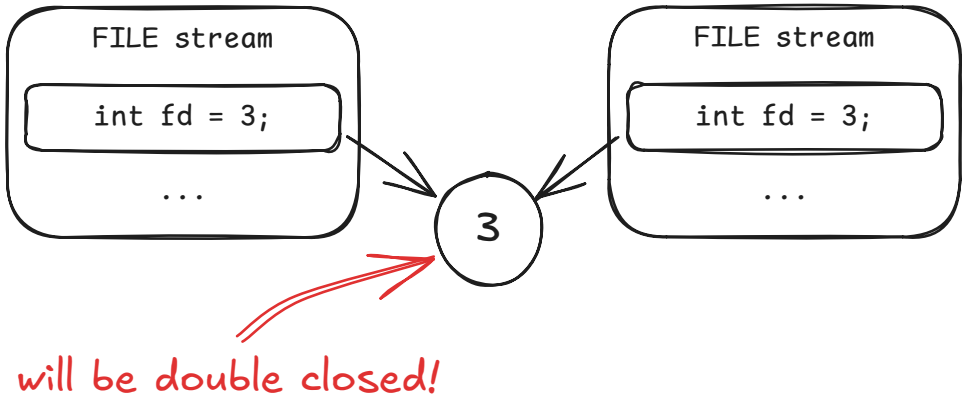
You can extract underlying file descriptor



This unlock full power of formatting library even if you have just a raw fd

Or wrap raw fd into a buffered stream

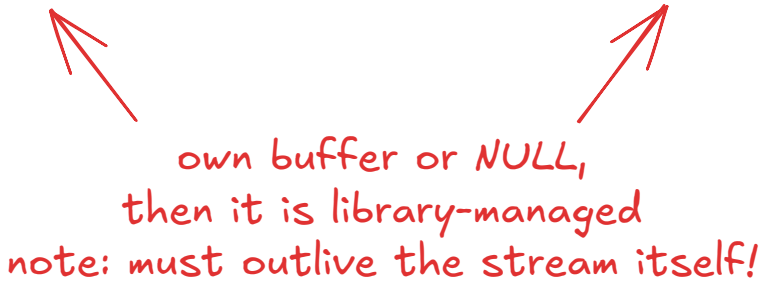
Be careful not to share the same descriptor in two different streams



Buffering modes

```
> man 3p setvbuf
```

```
int setvbuf(FILE *restrict stream, char *restrict buf, int type, size_t size)
```



`_IONBF`	Non-buffered - all operations are immediately translated to syscalls.
`_IOLBF`	Line-buffered - flushed after reaching a newline character.
`_IOBF`	Fully-buffered - flushed after filling the buffer completely.

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

```
> man 3p fflush
```

Manually forces write of pending data and discards any not cosumed read data.

Default buffering mode depends on underlying file type!

Stream I/O functions

```
> man 3p <all>
```

Character by character:

```
int fgetc(FILE *stream); int getc(FILE *stream);
int getchar(void);
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
```

String by string:

```
char* fgets(char *buf, int buflen, FILE *stream);
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

With conversions:

```
int fprintf(FILE *stream, const char *template, ...);
int fscanf(FILE *stream, const char *template, ...);
```

Raw:

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream);
```

Stream I/O functions

```
> man 3p <all>
```

Character by character:

```
int fgetc(FILE *stream); int getc(FILE *stream);
int getchar(void);
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
```

String by string:

```
char* fgets(char *buf, int buflen, FILE *stream);
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

With conversions:

```
int fprintf(FILE *stream, const char *template, ...);
int fscanf(FILE *stream, const char *template, ...);
```

Raw (OS and FS dependent!):

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream);
```

Stream position functions

```
> man 3p <all>
```

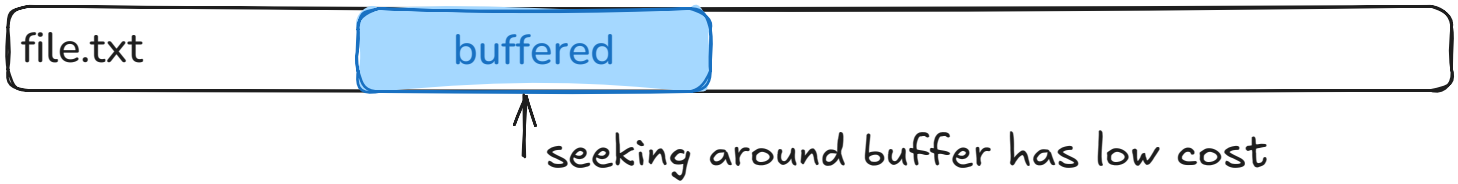
Getters:

```
long int ftell(FILE *stream);
off_t ftello(FILE *stream); // for large files
int fgetpos(FILE *stream, fpos_t *position);
```

Setters:

```
int fseek(FILE *stream, long int offset, int whence);
int fseeko(FILE *stream, off_t offset, int whence); // for large files
void rewind(FILE *stream);
int fsetpos(FILE *stream, const fpos_t *position);
```

Small adjustments to position within the stream do not need an lseek syscall thanks to buffering.



Stream state functions

```
> man 3p feof
> man 3p ferror
```

Each stream contains error and EOF indicator.

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
```

EOF and error indicator can be cleared only by calling clearerr()!
This means that stream after reaching the end won't capture concurrent file changes.

Stream synchronization

```
> man 3p flockfile
```

Each stream contains a lock (mutex) allowing for safe-by-default multi-threaded access.

```
void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

you can ensure proper serialization of series of reads/writes

If applications wish to opt-out from this default synchronization overhead it can do so with so called unlocked stdio.

```
> man 3 unlocked_sdio
```

```
int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
...
```