4. Files and File Systems

- Principles
- Structure and Storage
- Kernel Structures
- System Calls
- Directories
- Extended File Descriptor Manipulation
- Device-Specific Operations
- I/O Redirection
- Communication Through a Pipe

4. Files and File Systems

- Principles
- Structure and Storage
- Kernel Structures
- System Calls
- Directories
- Extended File Descriptor Manipulation
- Device-Specific Operations
- I/O Redirection
- Communication Through a Pipe

Storage Structure: Inode

Index Node

- UNIX distinguishes *file data* and *information about a file* (or meta-data)
- File information is stored in a structure called *inode*
- Attached to a particular device

Attributes

File Type
Number of hard links (they all share the same inode)
File length in bytes
Device identifier (DID)
User identifier (UID, file owner)
User group identifier (GID, user group of the file)
Timestamps: last status change (e.g., creation), modification, and access time
Access rights and file mode

Possibly more (non-POSIX) attributes, depending on the file system

Inode: Access Rights

- Classes of file accesses

  user: owner
  group: users who belong to the file's group, excluding the owner
  others: all remaining users

- Classes of access rights

  read: *directories: controls listing*
  write: *directories: controls file status changes*
  execute: *directories: controls searching (entering)*

- Additional file modes

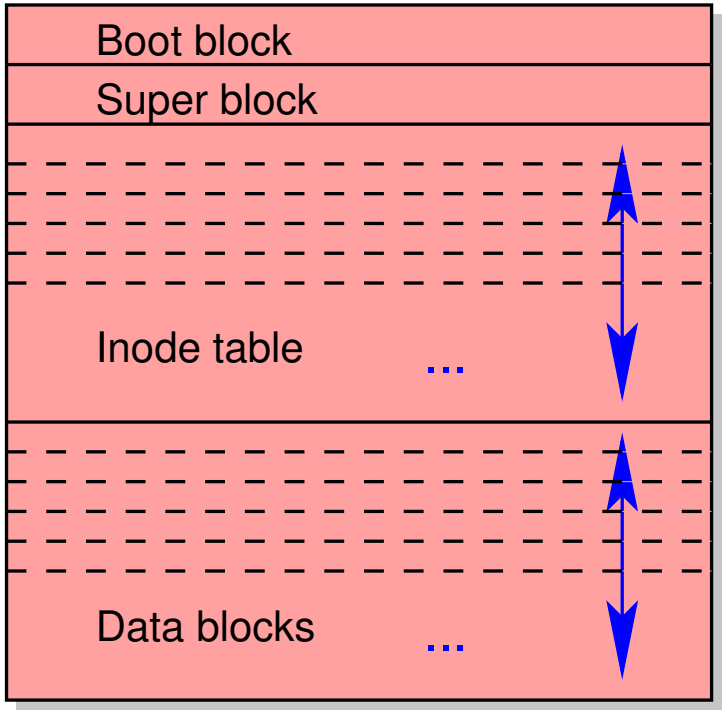
  suid: with **execute**, the process gets the file's UID
  directories: nothing
  sgid: with **execute**, the process gets the file's GID
  directories: created files inherit the creator process's GID
  sticky: loosely specified semantics related to memory management
  directories: files owned by others cannot be deleted or renamed

4. Files and File Systems

File System Storage

General Structure

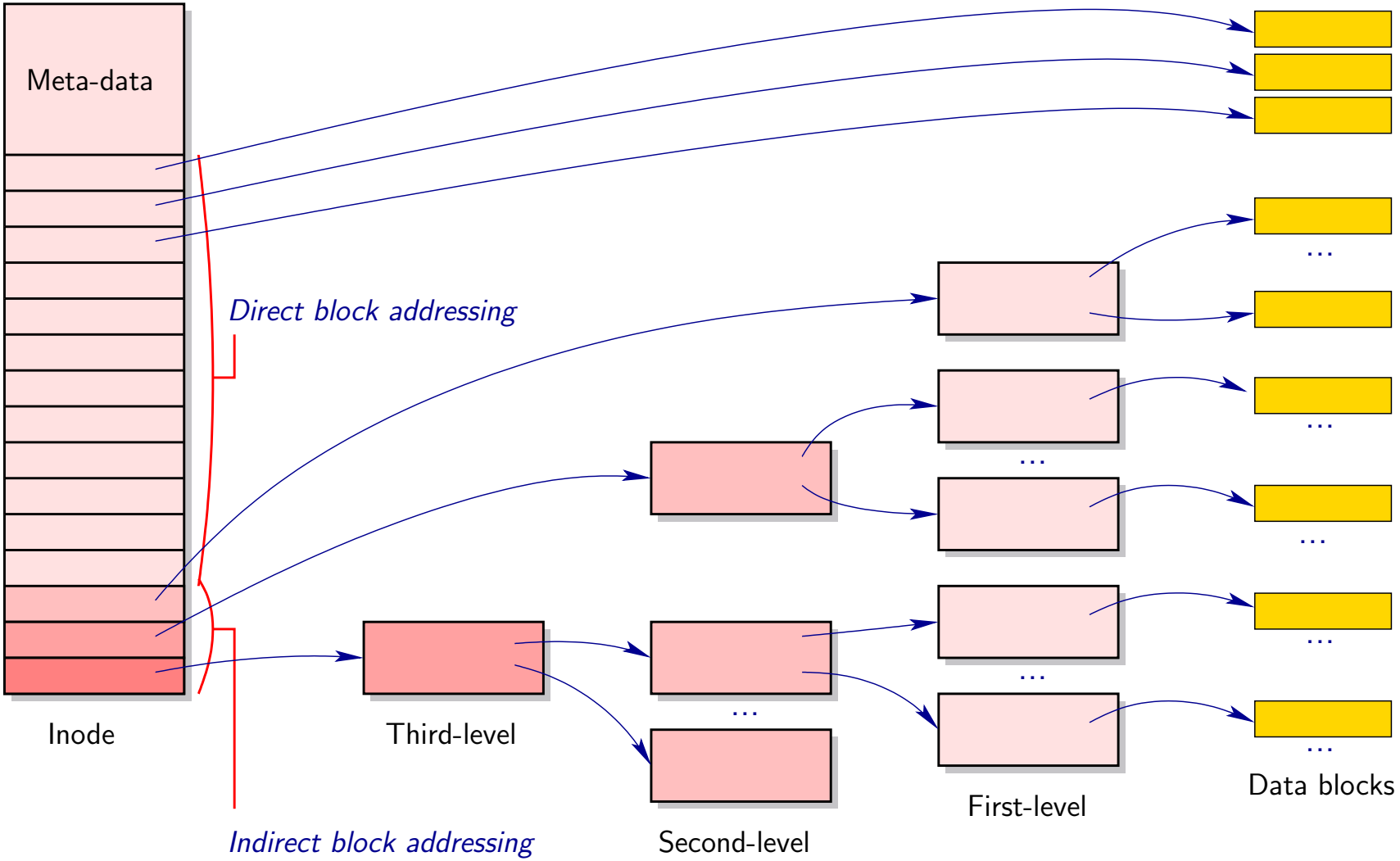
- Boot block
  - Bootstrap mode and "bootable" flag
  - Link to data blocks holding boot code
- Super block
  - File system status (mount point)
  - Number of allocated and free nodes
  - Link to lists of allocated and free nodes
- Inode table
- Data blocks
  - Note: directory = list of file names



Simplified file system layout

Inode: Data Block Addressing

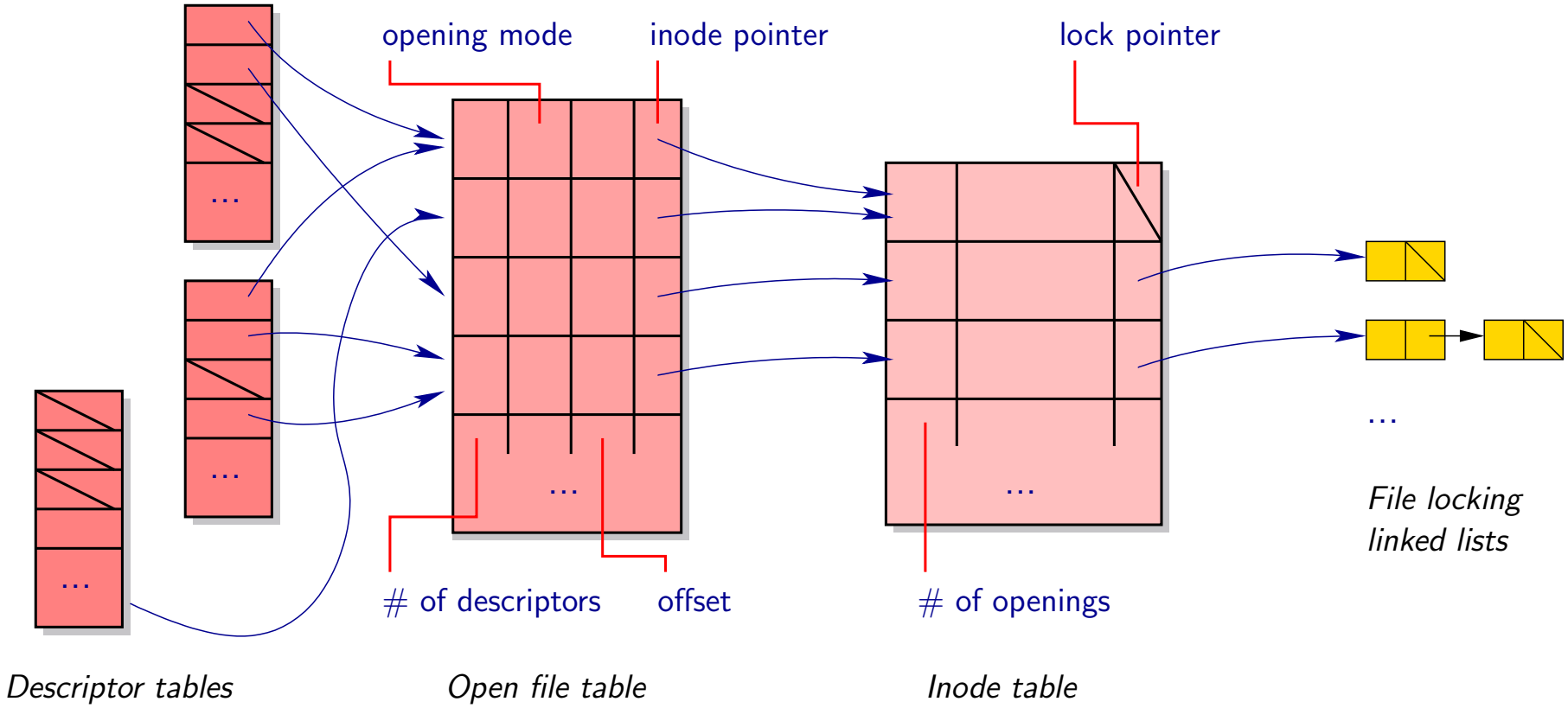
- Every Inode has a table of block addresses
- Addressing: direct, one-level indirect, two-levels indirect, ...



4. Files and File Systems

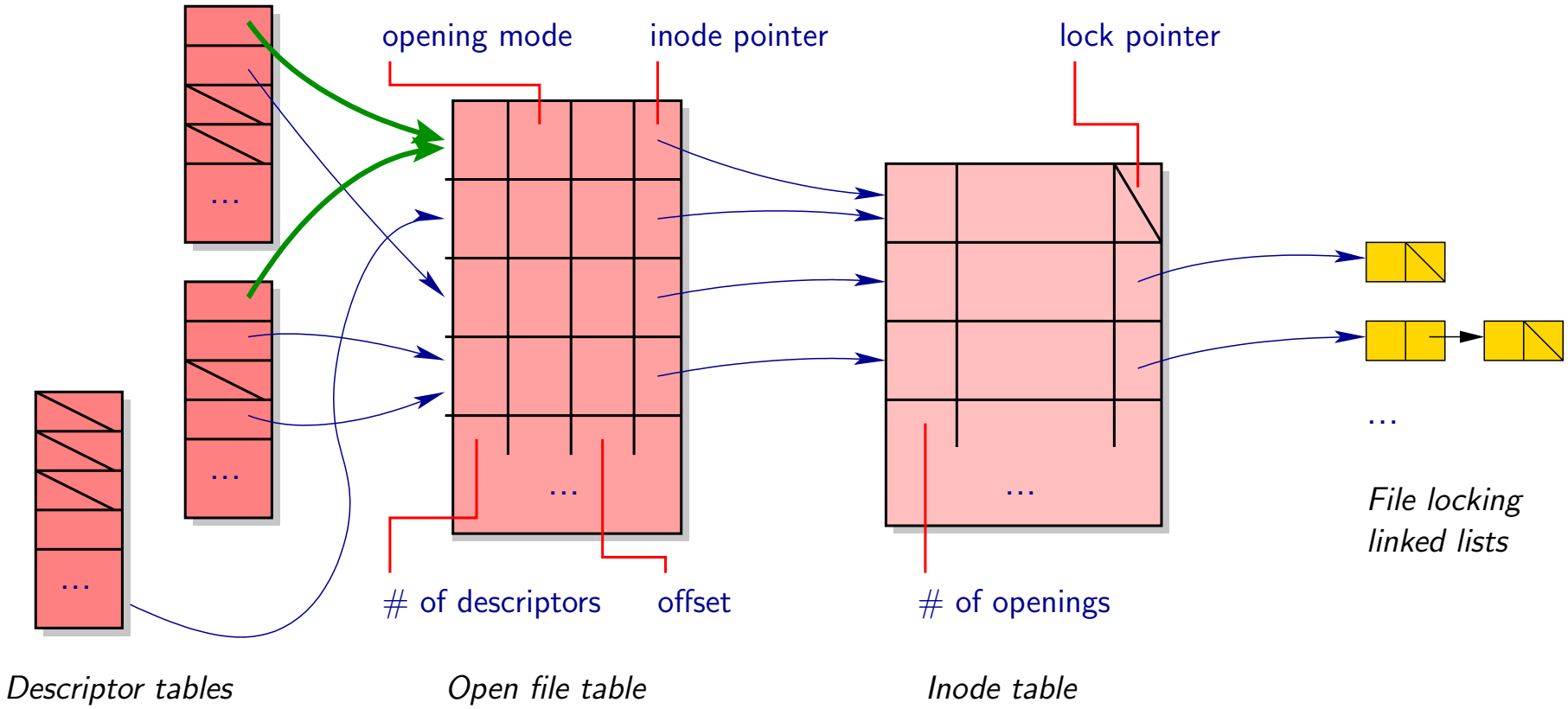
I/O Kernel Structures

- One table of *file descriptor*s per process: **0**:`stdin`, **1**:`stdout`, **2**:`stderr`
- Table of *open file*s (status, including opening mode and offset)
- Inode table (for all open files)
- File locks (*see chapter on advanced synchronization*)



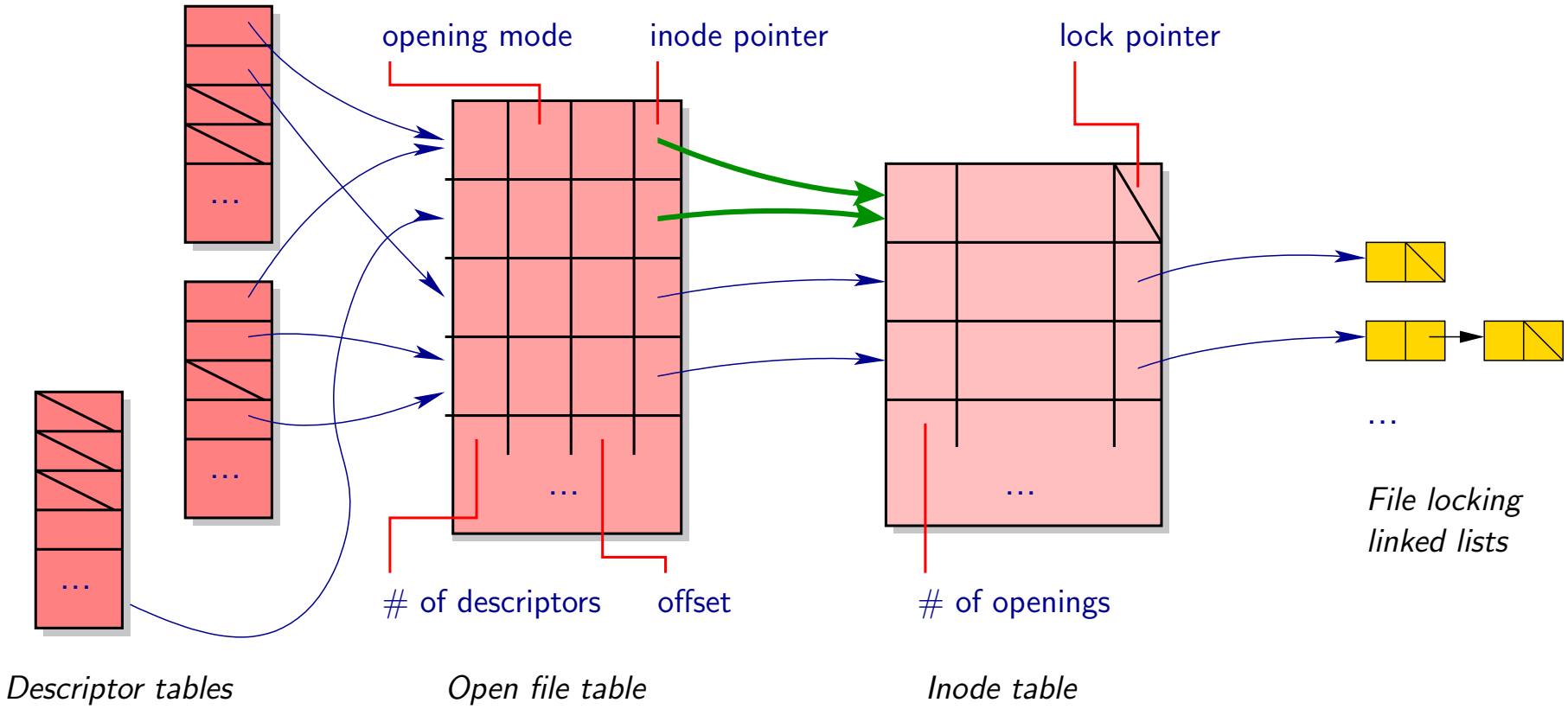
I/O Kernel Structures

- Example: file descriptor aliasing
  - E.g., obtained through the `dup()` or `fork()` system calls



I/O Kernel Structures

- Example: open file aliasing
  - E.g., obtained through multiple calls to `open()` on the same file
  - Possibly via hard or soft links



4. Files and File Systems

I/O System Calls

Inode Manipulation

- `stat()` `access()`, `link()`, `unlink()`, `chown()`, `chmod()`, `mknod()`, ...
- Note: many of these system calls have `l`-prefixed variants (e.g., `lstat()`) that *do not* follow soft links
- Note: many of these system calls have `f`-prefixed variants (e.g., `fstat()`) operating on *file descriptors*
  Warning: they are not to be confused with C library functions

I/O System Calls

Inode Manipulation

- `stat()` `access()`, `link()`, `unlink()`, `chown()`, `chmod()`, `mknod()`, ...
- Note: many of these system calls have `l`-prefixed variants (e.g., `lstat()`) that *do not* follow soft links
- Note: many of these system calls have `f`-prefixed variants (e.g., `fstat()`) operating on *file descriptors*
  Warning: they are not to be confused with C library functions

File descriptor manipulation

- `open()`, `creat()`, `close()`, `read()`, `write()`, `lseek()`, `fcntl()`...
- We will describe `dup()` when studying redirections
- Note: `open()` may also create a new file (hence a new inode)
- Use `fdopen()` and `fileno()` to get a file C library `FILE*` from a file descriptor and reciprocally, but *do not mix C library and system call I/O on the same file (because of C library internal buffers)*

I/O System Call: stat()

Return Inode Information About a File

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

int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Error Conditions

- The system call returns **0** on success, **−1** on error
- A few possible `errno` codes

  `EACCES`: search (enter) permission is denied for one of the directories in the prefix of `path`

  `ENOENT`: a component of `path` does not exist — file not found — or the path is an empty string

  `ELOOP`: too many symbolic links encountered when traversing the path

I/O System Call: stat()

Inode Information Structure

```
struct stat {
  dev_t     st_dev;     // ID of device containing file
  ino_t     st_ino;     // Inode number
  mode_t    st_mode;    // Protection
  nlink_t   st_nlink;   // Number of hard links
  uid_t     st_uid;     // User ID of owner
  gid_t     st_gid;     // Group ID of owner
  dev_t     st_rdev;    // Device ID (if special file)
  off_t     st_size;    // Total size, in bytes
  blksize_t st_blksize; // Blocksize for filesystem I/O
  blkcnt_t  st_blocks;  // Number of blocks allocated
  time_t    st_atime;   // Time of last access
  time_t    st_mtime;   // Time of last modification
  time_t    st_ctime;   // Time of last status change
};
```

I/O System Call: `stat()`

Deciphering `st_mode`

Macros to determine file type

- `S_ISREG(m)`: is it a regular file?
- `S_ISDIR(m)`: directory?
- `S_ISCHR(m)`: character device?
- `S_ISBLK(m)`: block device?
- `S_ISFIFO(m)`: FIFO (named pipe)?
- `S_ISLNK(m)`: symbolic link?
- `S_ISSOCK(m)`: socket?

File type constants

- `S_IFREG`: regular file
- `S_IFDIR`: directory
- `S_IFCHR`: character device
- `S_IFBLK`: block device
- `S_IFFIFO`: FIFO (named pipe)
- `S_IFLNK`: symbolic link
- `S_IFSOCK`: socket

I/O System Call: `stat()`

Deciphering `st_mode`

Macros to determine access permission and mode
Usage: *flags* and *masks* can be *or'ed* and *and'ed* together, and with `st_mode`

Constant	Octal value	Comment
S_ISUID	04000	SUID bit
S_ISGID	02000	SGID bit
S_ISVTX	01000	sticky bit
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

I/O System Call: access()

Check Whether the Process Is Able to Access a File

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Access Mode Requests

R_OK: check for read permission

W_OK: check for write permission

X_OK: check for execute permission

F_OK: check for the existence of the file

Error Conditions

- The system call returns **0** on success, **−1** on error
- A few original errno codes

EROFS: write access request on a read-only filesystem

ETXTBSY: write access request to an executable which is being executed

I/O System Call: `link()`

Make a New Name (Hard Link) for a File

```
#include <unistd.h>

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

See also: `symlink()`

Error Conditions

- Cannot hard-link directories (to maintain a tree structure)
- The system call returns **0** on success, **−1** on error
- A few original `errno` codes

  `EEXIST`: `newpath` already exists (`link()` preserves existing files)

  `EXDEV`: `oldpath` and `newpath` are not on the same file system

I/O System Call: `unlink()`

Delete a Name and Possibly the File it Refers To

```
#include <unistd.h>

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

Error Conditions

- The system call returns **0** on success, **−1** on error
- An original `errno` code

  `EISDIR`: attempting to delete a directory (see `rmdir()`)

I/O System Call: chown()

Change Ownership of a File

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

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Error Conditions

- The system call returns **0** on success, **−1** on error
- An original `errno` code

  `EBADF`: the descriptor is not valid

I/O System Call: chmod()

Change Access Permissions of a File

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

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Access Permissions

- Build `mode` argument by *or'ing* the access mode constants
  E.g., `mode = S_IRUSR | S_IRGRP | S_IROTH; // 0444`

Error Conditions

- The system call returns **0** on success, **−1** on error

I/O System Call: mknod()

Create any Kind of File (Inode)

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

File Type

- Set `mode` argument to *one* of the file type constants *or'ed* with any combination of access permissions
  E.g., `mode = S_IFREG | S_IRUSR | S_IXUSR; // Regular file`
- If `mode` is set to `S_IFCHR` or `S_IFBLK`, `dev` specifies the *major* and *minor* numbers of the newly created device special file
- File is created with permissions (`mode & ~current_umask`) where `current_umask` is the process's mask for file creation (see `umask()`)

I/O System Call: mknod()

Create any Kind of File (Inode)

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Error Conditions

- The system call returns **0** on success, **−1** on error
- A few original `errno` codes

  `EEXIST`: `newpath` already exists (`mknod()` preserves existing files)
  `ENOSPC`: device containing `pathname` has no space left for a new node

I/O System Call: open()/creat()

Open and Possibly Create a File

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

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

Return Value

- On success, the system call returns a (non-negative) *file descriptor*
  - Note: it is the process's *lowest-numbered file descriptor not currently open*
- Return -1 on error

I/O System Call: open()/creat()

Open and Possibly Create a File

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

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

Acces Permissions

- File is created with permissions (`mode & ~current_umask`) where `current_umask` is the process's mask for file creation (see `umask()`)

I/O System Call: open()/creat()

Open and Possibly Create a File

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

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

Flags

- Access mode set to *one* of O_RDONLY, O_WRONLY, O_RDWR
  Note: opening a file in read-write mode is very different from opening it twice in read then write modes (see e.g. the behavior of lseek())
- Possibly *or'ed* with O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NONBLOCK, ...

I/O System Call: `close()`

Close a File Descriptor

```
#include <unistd.h>

int close(int fd);
```

Remarks

- When closing the last descriptor to a file that has been removed using `unlink()`, the file is effectively deleted
- It is sometimes desirable to flush all pending writes (persistent storage, interactive terminals): see `fsync()`

Error Conditions

- The system call returns **0** on success, **−1** on error
- It is important to check error conditions on `close()`, to avoid losing data

I/O System Call: `read()`

Read From a File Descriptor

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Semantics

- Attempts to read *up to* `count` bytes from file descriptor `fd` into the buffer starting at `buf`
  - Return immediately if `count` is **0**
  - May read less than `count` bytes: it is not an error
    E.g., close to end-of-file, interrupted by signal, reading from socket...
- On success, *returns the number of bytes effectively read*
- Return **0** if at *end-of-file*
- Return -1 on error (hence the signed `ssize_t`)

I/O System Call: `read()`

Read From a File Descriptor

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Error Conditions

- Important `errno` codes

  `EINTR`: call interrupted by a signal *before anything was read*

  `EAGAIN`: non-blocking I/O is selected and no data was available

I/O System Call: `write()`

Write to File Descriptor

```
#include <unistd.h>

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

Semantics

- Attempts to write *up to* `count` bytes to the file file referenced by the file descriptor `fd` from the buffer starting at `buf`
  - Return immediately if `count` is **0**
  - May write less than `count` bytes: it is not an error
- On success, *returns the number of bytes effectively written*
- Return -1 on error (hence the signed `ssize_t`)

Error Conditions

- An original `errno` code

  `ENOSPC`: no space left on device containing the file

Example: Typical File Open/Read Skeleton

```
void my_read(char *pathname, int count, char *buf)
{
  int fd;
  if ((fd = open(pathname, O_RDONLY)) == -1) {
    perror("‘my_function’: ‘open()’ failed");
    exit(1);
  }
  // Read count bytes
  int progress, remaining = count;
  while ((progress = read(fd, buf, remaining)) != 0) {
    // Iterate while progess or recoverable error
    if (progress == -1) {
      if (errno == EINTR)
        continue; // Interrupted by signal, retry
      perror("‘my_function’: ‘read()’ failed");
      exit(1);
    }
    buf += progress; // Pointer artithmetic
    remaining -= progress;
  }
}
```

4. Files and File Systems

Directory Traversal

Directory Manipulation (C Library)

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

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

$ man 3 opendir, $ man 3 readdir, etc.

Directory Entry Structure

```
struct dirent {
  long d_ino;                   // Inode number
  off_t d_off;                  // Offset to this dirent
  unsigned short d_reclen;      // Length of this d_name
  char d_name[NAME_MAX+1];      // File name ('\0'-terminated)
}
```

Example: Mail Folder Traversal

```
int last_num(char *directory)
{
  struct dirent *d;
  DIR *dp;
  int max = -1;
  dp = opendir(directory);
  if (dp == NULL) {
    perror("‘last_num’: ‘opendir()’ failed");
    exit(1);
  }
  while ((d = readdir(dp)) != NULL) {
    int m;
    m = atoi(d->d_name); // Parse string into ‘int’
    max = MAX(max, m);
  }
  closedir(dp);
  return max; // -1 or n >= 0
}
```

Example: Mail Folder Traversal

```
void remove_expired(char *directory, int delay, int last_num)
{
  struct dirent *d;
  time_t now;
  struct stat stbuf;
  DIR *dp = opendir(directory);
  if (dp == NULL) {
    message(1, "‘remove_expired’: ’opendir()’ failed");
    return;
  }
  time(&now);
  while ((d = readdir(dp)) != NULL) {
    int m = atoi(d->d_name);
    if (m >= 0 && m != last_num) {
      if (stat(d->d_name, &stbuf) != -1 && stbuf.st_mtime < now - delay)
        unlink(d->d_name);
    }
  }
  closedir(dp);
}
```

4. Files and File Systems

I/O System Call: `fcntl()`

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Some Commands

`F_GETFL`: get the file status flags

`F_GETFL`: set the file status flags to the value of `arg`

- No access mode (e.g., `O_RDONLY`) and creation flags (e.g., `O_CREAT`), but accepts `O_APPEND`, `O_NONBLOCK`, `O_NOATIME`, etc.

And many more: descriptor behavior options, duplication and locks, I/O-related signals (terminals, sockets), etc.

- See chapter on processes and on advanced synchronization

I/O System Call: `fcntl()`

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Return Value

- On success, `fcntl()` returns a (non-negative) value which depends on the command
  - `F_GETFD`: the descriptor's flags
  - `F_GETFD`: **0**
- Return -1 on error

4. Files and File Systems

Device-Specific Operations

I/O "Catch-All" System Call: `ioctl()`

- Implement operations that do not directly fit into the stream I/O model (`read` and `write`)
- Typical examples
  - Block-oriented devices: CD/DVD *eject* operation
  - Character-oriented devices: *terminal* control
- Prototype

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, char *argp);
```

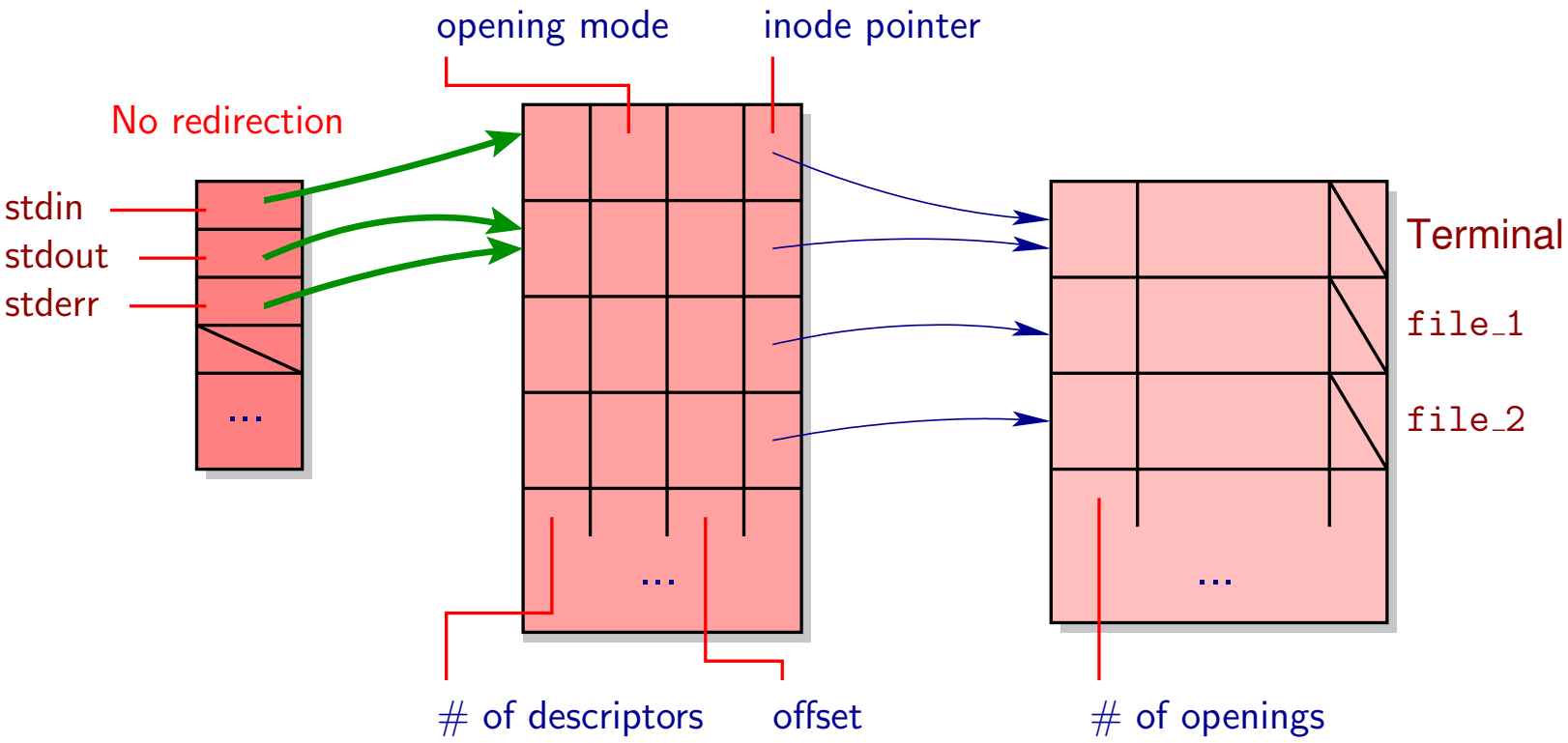
`fd`: open file descriptor
`request`: device-dependent request code
`argp`: buffer to load or store data
(its size and structure is request-dependent)

4. Files and File Systems

I/O Redirection

Example

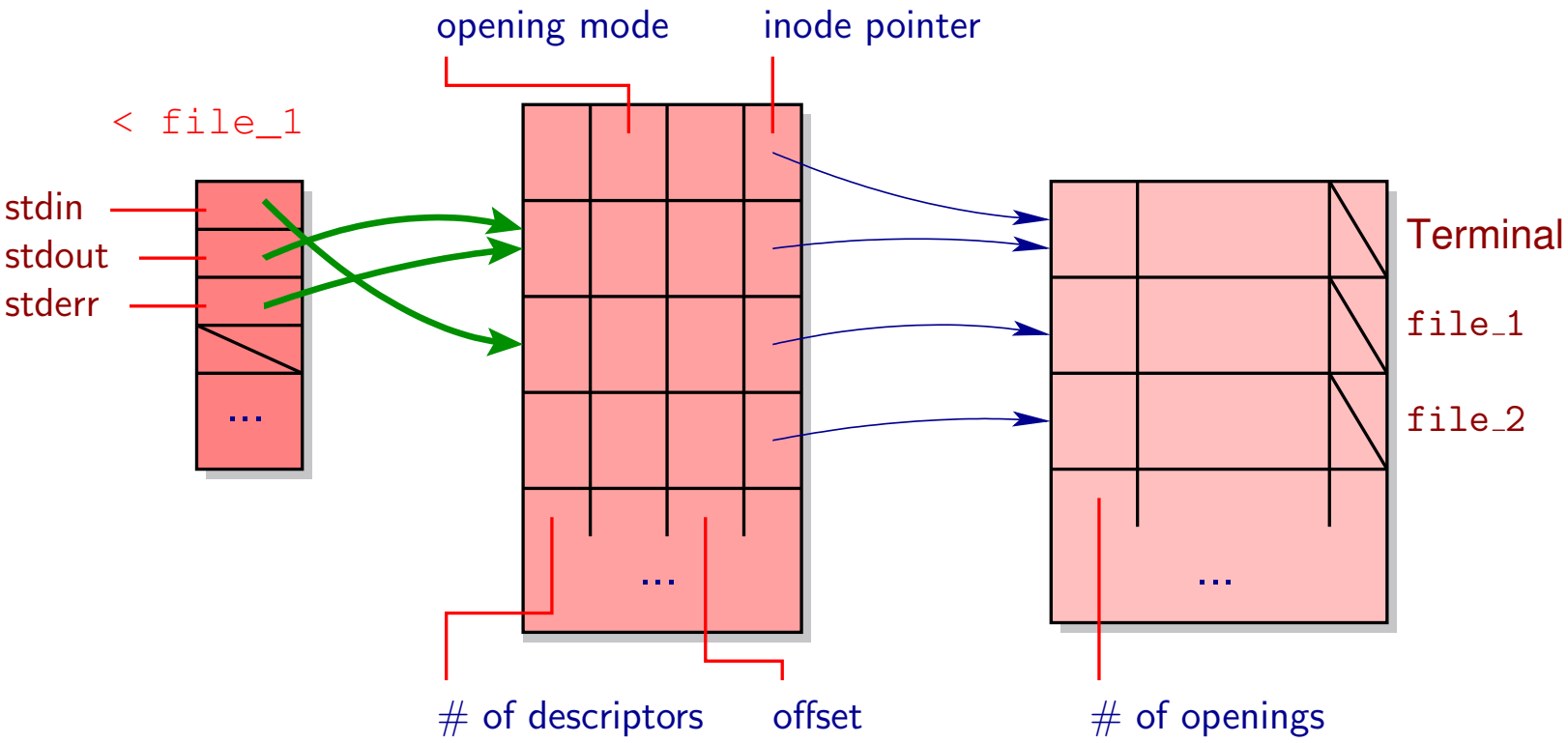
No redirection



I/O Redirection

Example

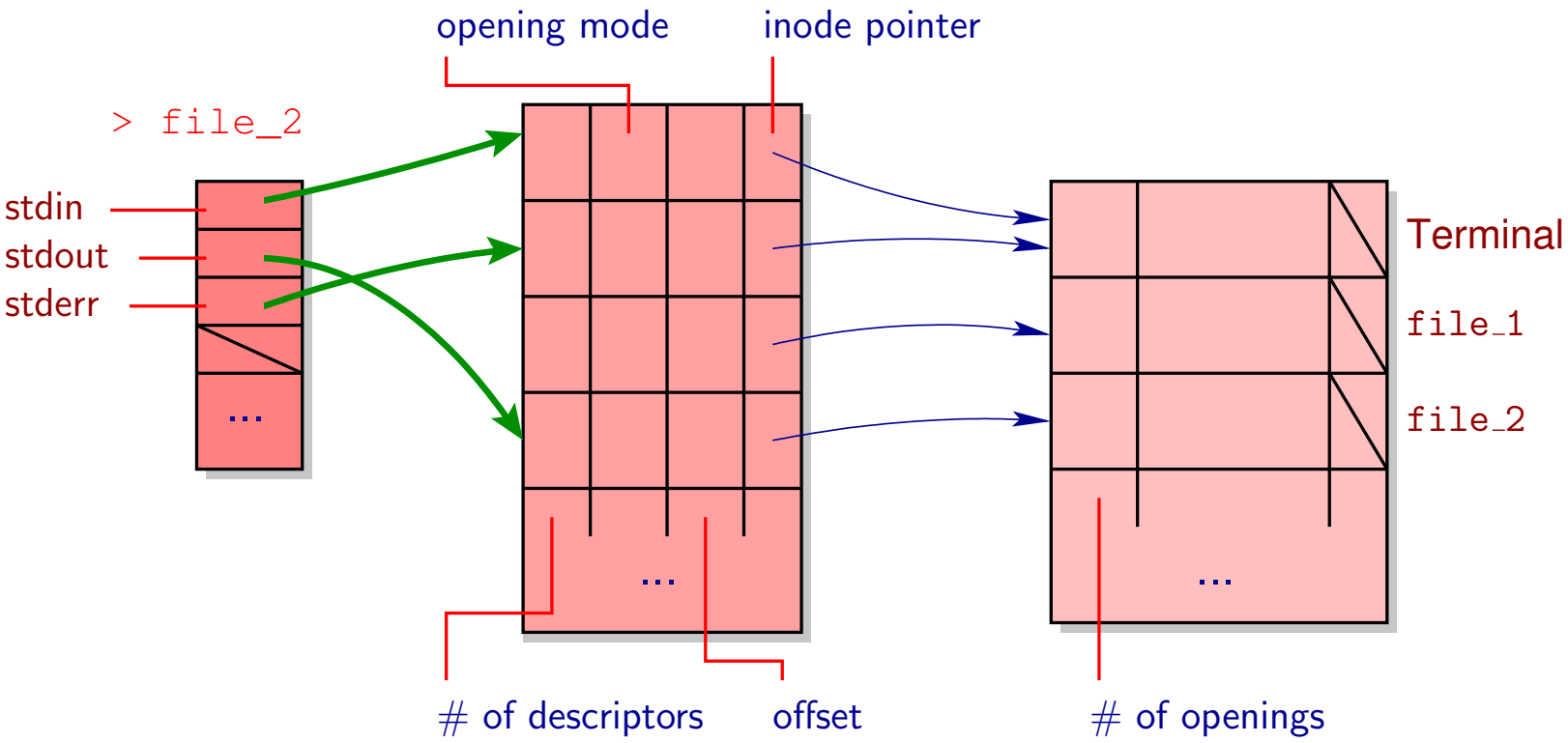
Standard input redirection



I/O Redirection

Example

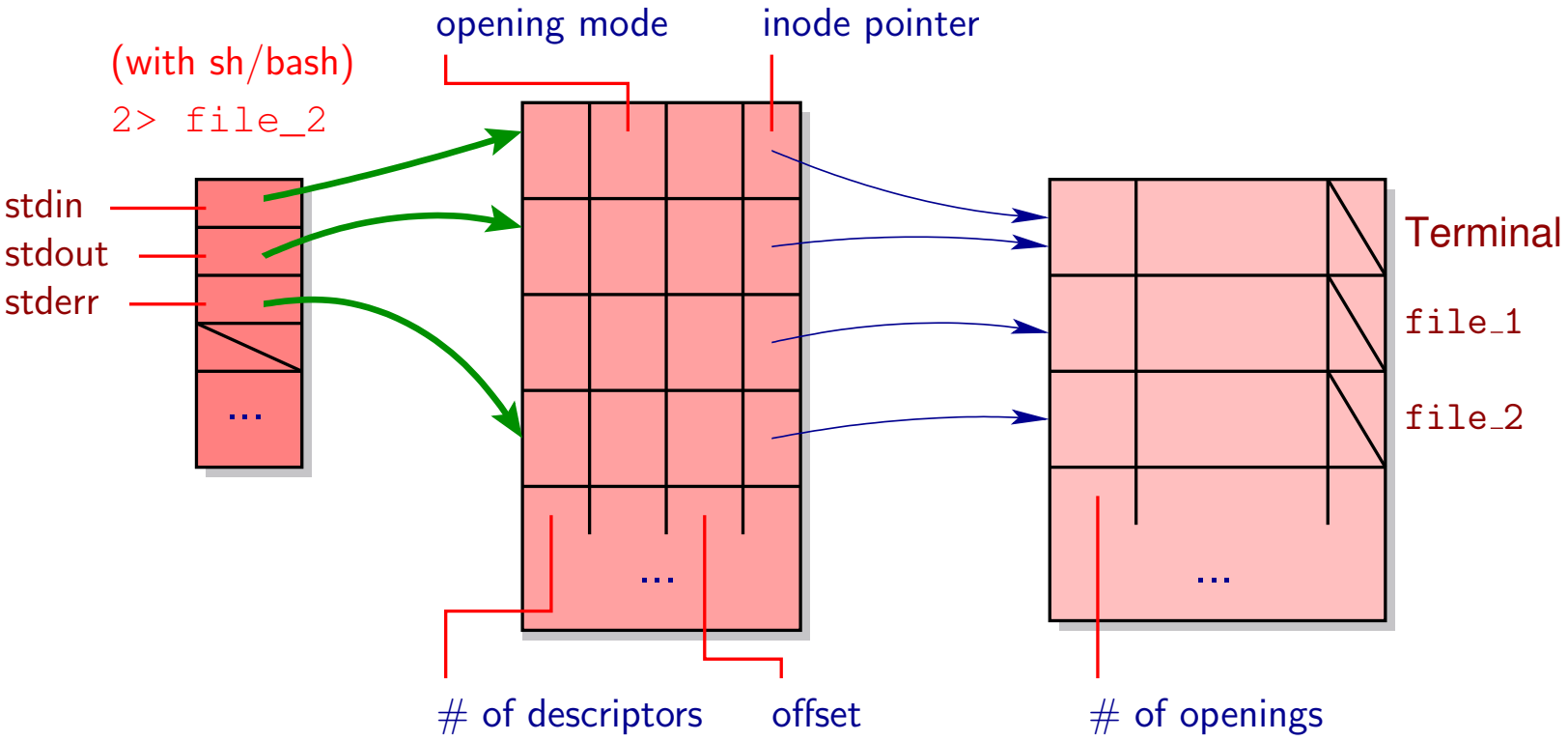
Standard output redirection



I/O Redirection

Example

Standard error redirection



I/O System Call: dup()/dup2()

Duplicate a File Descriptor

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Return Value

- On success, dup()/dup2() return a *file descriptor*, a copy of oldfd
  - For dup(), it is the process's *lowest-numbered descriptor not currently open*
  - dup2() uses newfd instead, closing it before if necessary
  - Clears the flags of the new descriptor (see fcntl())
  - Both descriptors share one single open file (i.e., one offset for lseek(), etc.)
- Return -1 on error

Error Conditions

- An original errno code

  EMFILE: too many file descriptors for the process

Redirection Example

```
$ command > file_1    // Redirect stdout to file_1

{
  close(1);
  open("file_1", O_WRONLY | O_CREAT, 0777);
}
```

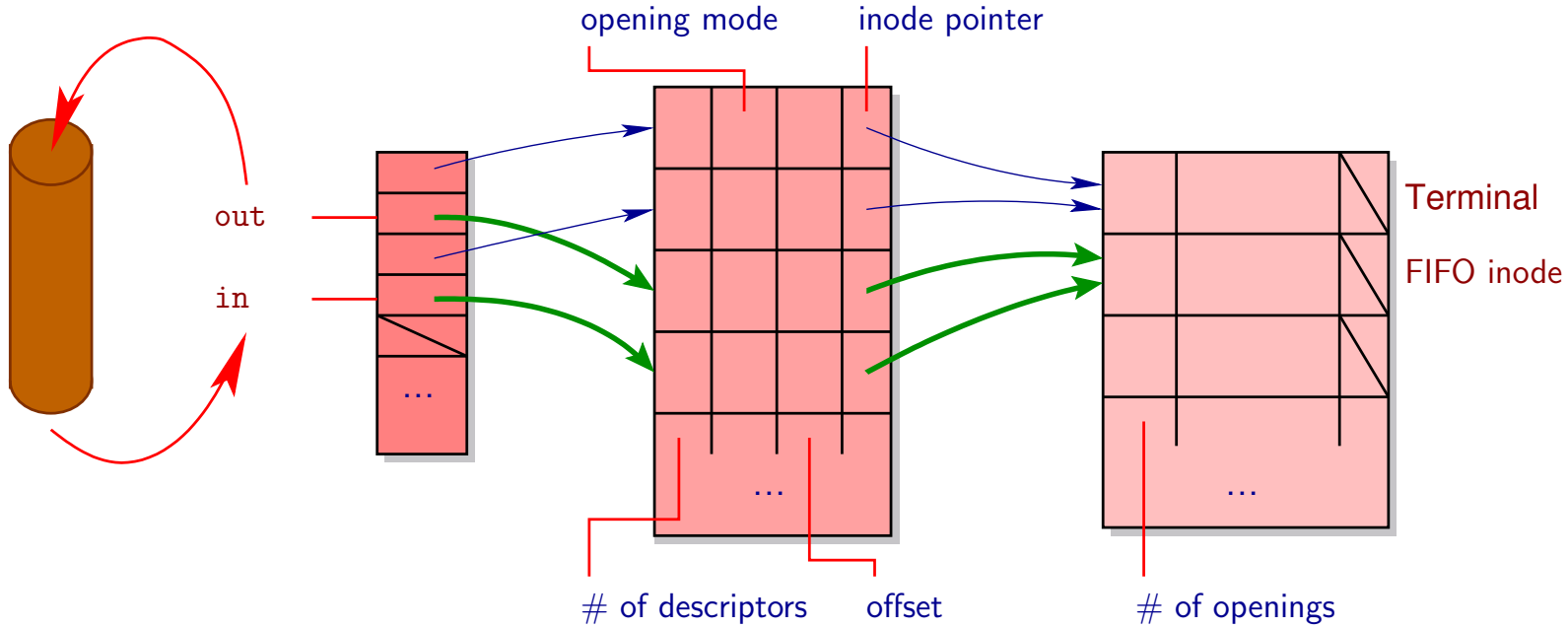
```
$ command 2>&1        // Redirect stderr to stdout

{
  close(2);
  dup(1);
}
```

4. Files and File Systems

FIFO (Pipe)

Principles

- Channel to stream data among processes
  - Data traverses the pipe *first-in* (write) *first-out* (read)
  - Blocking read and write by default (bounded capacity)
  - Illegal to write into a pipe without reader
  - A pipe without writer simulates *end-of-file*: `read()` returns **0**
- Pipes have *kernel* persistence



I/O System Call: `pipe()`

Create a Pipe

```
#include <unistd.h>

int pipe(int p[2]);
```

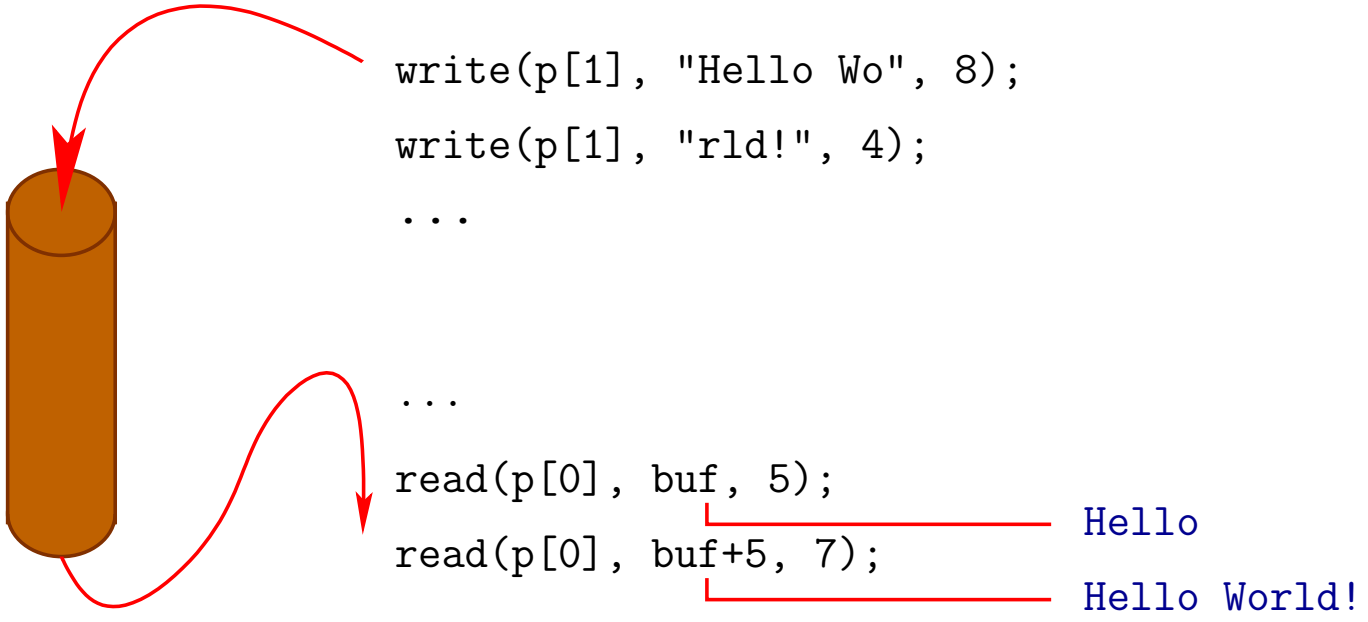
Description

- Creates a pipe and stores a pair of file descriptors into `p`
  - `p[0]` for reading (`O_RDONLY`)
  - `p[1]` for writing (`O_WRONLY`)
- Return **0** on success, $-\mathbf{1}$ if an error occurred

FIFO Communication

Unstructured Stream

- Like ordinary files, data sent to a pipe is unstructured: it does not retain "boundaries" between calls to `write()` (unlike IPC message queues)



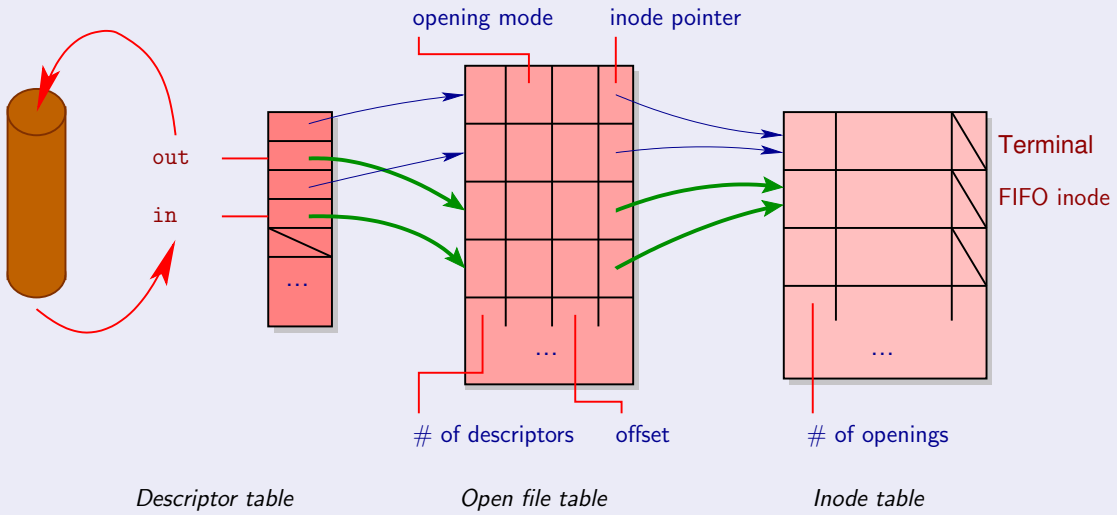
FIFO Communication

Writing to a Pipe

- Writing to a pipe without readers delivers of `SIGPIPE`
  - Causes termination by default
  - Otherwise causes `write()` to fail with error `EINTR`
- `PIPE_BUF` is a constant $\geqslant$ **512** (**4096** on Linux)
- Writing **n** bytes in blocking mode
  - **n** $\leqslant$ `PIPE_BUF`: atomic success (**n** bytes written), block if not enough space
  - **n** $>$ `PIPE_BUF`: non-atomic (may be interleaved with other), blocks until **n** bytes have been written
- Writing **n** bytes in non-blocking mode (`O_NONBLOCK`)
  - **n** $\leqslant$ `PIPE_BUF`: atomic success (**n** bytes written), or fails with `EAGAIN`
  - **n** $>$ `PIPE_BUF`: if the pipe is full, fails with `EAGAIN`; otherwise a partial write may occur

FIFOs and I/O Redirection

- Question: implement
  `$ ls | more`
- Solution
  - `pipe(p)`
  - `fork()`
  - Process to become `ls`
    - `close(1)`
    - `dup(p[1])`
    - `execve("ls", ...)`
  - Process to become `more`
    - `close(0)`
    - `dup(p[0])`
    - `execve("more", ...)`
- Short-hand: `$ man 3 popen`



FIFO Special Files

Named Pipe

- Special file created with `mkfifo()` (front-end to `mknod()`)
  - See also `mkfifo` command
- Does not store anything on the file system (beyond its inode)
  - Data is stored and forwarded in memory (like an unnamed pipe)
- Supports a rendez-vous protocol
  - Open for reading: blocks until another process opens for writing
  - Open for writing: blocks until another process opens for reading
- Disabled when opening in `O_NONBLOCK` mode