2.2. File System

PROFESSORS:

Eduardo Huedo Cuesta

Rubén Santiago Montero

OTHER AUTHORS:

Ignacio Martín Llorente

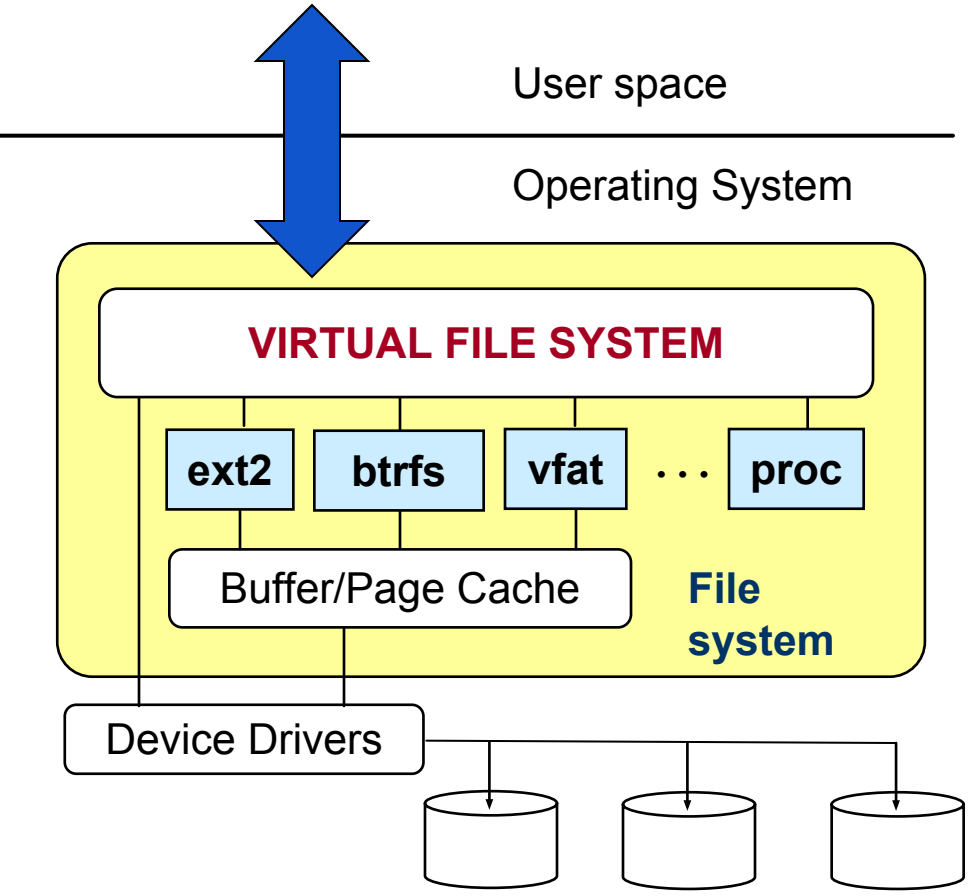
Juan Carlos Fabero Jiménez

Characteristics of File Systems

- **From the user point of view**
  - Collection of files and directories used to store and organize information
- **From the operating system point of view**
  - Set of tables and structures that allow managing files and directories

- **File system types:**
  - **Disk-based:** Stored on physical media such as hard disks, optical disks or flash media
    - Examples: Minix, ext2-3-4, FAT, NTFS, ISO9660, ufs, hpfs, XFS, BTRFS, ZFS…
  - **Network-based (or distributed):** Used to access remote file systems despite their type
    - Examples: NFS (Network File System), SMB...
  - **Memory-based (or pseudo):** Their contents reside in main memory while the operating system is running
    - Examples: procfs, tmpfs...

File System Architecture



- Establishes a well defined **link between the OS kernel** and the different **file systems**
- Provides the different **calls** for file management, **independent of the file system**
- Allows accessing multiple different file systems
- **Optimizes I/O** by means of:
  - Inode and Dentry caches
  - Buffer/Page cache (*sync*)

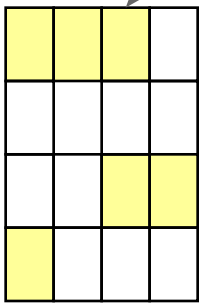
File System Structure: Block Groups

File system evolution: Minix → ext → ext2 → ext3 → ...
Inspired in FFS (Fast File System) from BSD

- Data blocks are close to their inode
- Inodes are close to their directory

Block Group 0	Block Group 1	Block Group 2	...	Block Group n

Super-Block	Group Descriptors	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

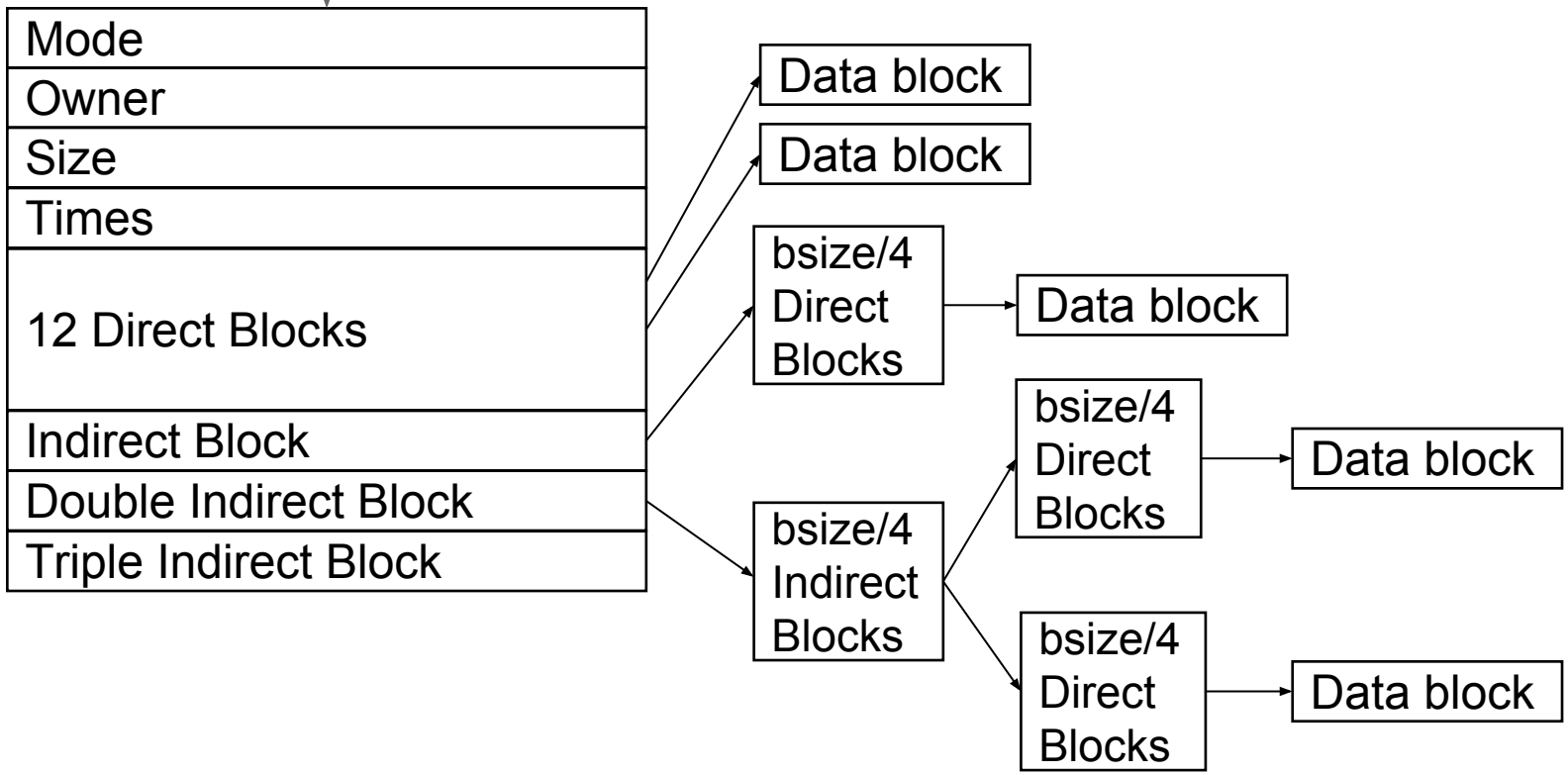


ext2 structure

File System Structure: Inodes

Directory:

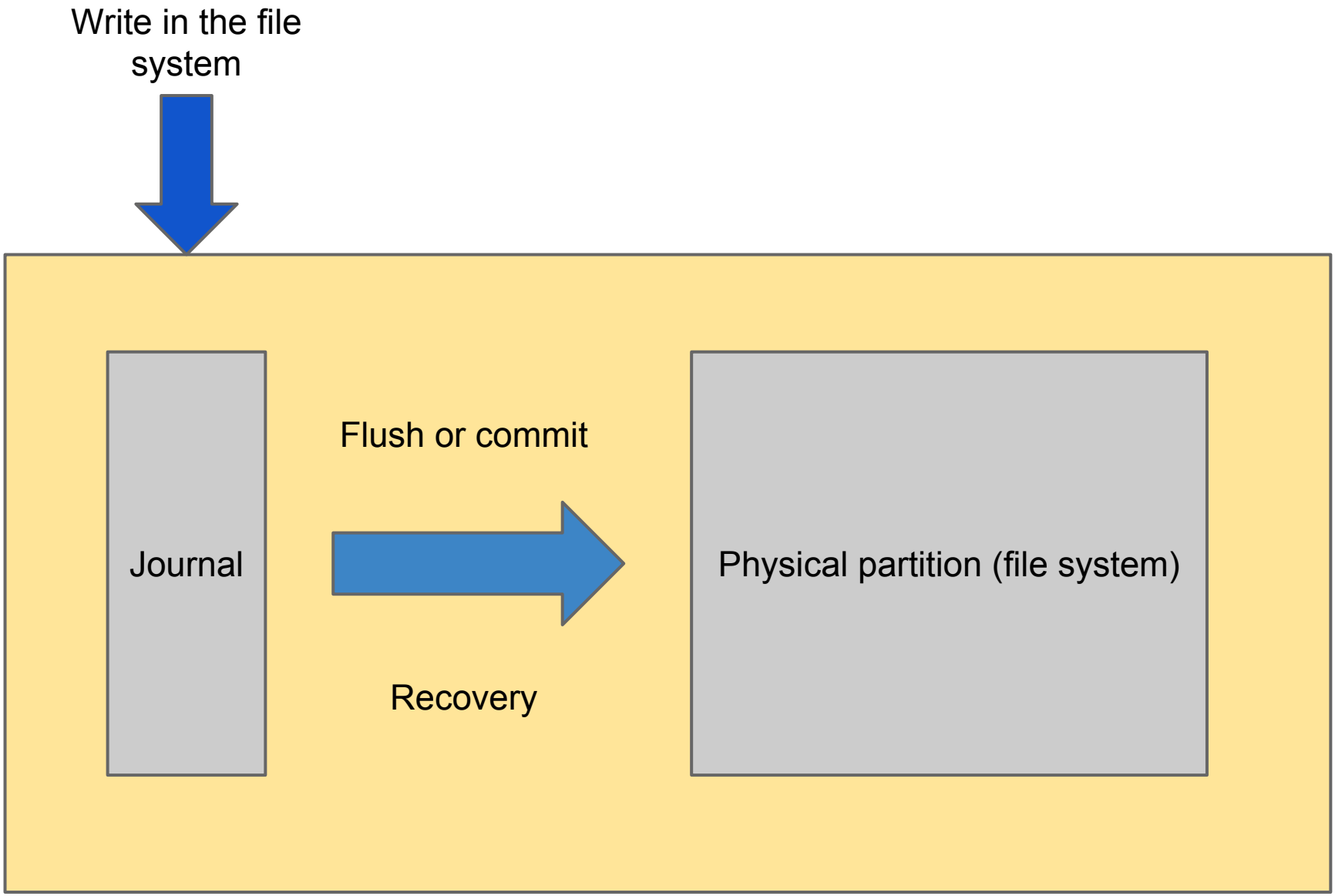
Inode:



Journaling

- When a traditional file system is not appropriately shutdown, the OS must check the integrity and consistency of the file system during the next boot (`fsck` tool)
  - This implies walking all the file system structure in search of orphaned inodes and other inconsistencies, which in large systems it can take a long time
  - In some occasions, it is not possible to automatically repair the structure, and a manual repair must be done
- Modern file systems incorporate a special file, area or device, named *journal* (or *log*), that avoids data corruption
- Some changes in the disk are first written in the journal
- In the event of a system crash or power failure, the journal file can be used to return the system to a consistent state
- ReiserFS, XFS, ext3 and ext4 file systems incorporate journaling

Journaling


Write in the file system
Journal
Flush or commit
Recovery
Physical partition (file system)

Journaling

- Metadata (inodes, bitmaps...) is always immediately written to the journal
- Depending on how data is written, there are three main alternatives:
  - **Writeback mode**: Relies on the standard write process to write file data changes to disk. Ordering is not preserved, so data may be written into the main file system after its metadata has been committed to the journal. This is the fastest journaling mode, but in case of failure data can be lost and files may contain old data or other garbage.
  - **Ordered mode**: File data updates are flushed to disk before committing changes to associated metadata (i.e. ordering is preserved). This is the default journaling mode in ext3.
  - **Journal mode**: All data changes are also written in the journal. This is the slowest journaling mode, but minimizes the chance of losing changes made to any file.
- Journal flushing to commit changes is done periodically or based on the journal size

File Attributes

<sys/types.h>
<sys/stat.h>

SV+BSD+POSIX

- Return information about a file:

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

  - `stat` follows symbolic links, while `lstat` doesn't
  - `fstat` get status of a file specified by a file descriptor
  - Execute/search permission is required on all of the directories in the path that lead to the file (no permissions are required on the file itself)
  - Error codes:
    - `EBADF`: `fd` is bad
    - `ENOENT`: A component of `path` does not exist, or `path` is an empty string
    - `ENOTDIR`: A component of `path` is not a directory
    - `ELOOP`: Too many symbolic links encountered while traversing the path
    - `EFAULT`: Bad address
    - `EACCES`: Search permission is denied for one of the directories of `path`
    - `ENAMETOOLONG`: `path` is too long

File Attributes

```
struct stat {
    dev_t st_dev;     /* ID of device containing the file */
    ino_t st_ino;     /* Inode number */
    mode_t st_mode;   /* Permissions */
    nlink_t st_link;  /* 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 (bytes) */
    unsigned long st_blksize; /* Block size for FS I/O */
    unsigned long st_blocks;  /* Allocated blocks */
    time_t st_atime; /* Last access */
    time_t st_mtime; /* Last modification */
    time_t st_ctime; /* Last status change (inode) */
};
```

- `st_blksize`: Preferred block size for IO operations for optimal performance
- `st_blocks`: Number of 512-byte blocks allocated to the file
- `st_atime`: Modified by `read`, `write`, `mknod`, `utime` or `truncate`
- `st_mtime`: Modified by `write`, `mknod` or `utime`, not when metadata changes
- `st_ctime`: Only modified when metadata (i.e. the inode) changes

File Attributes

POSIX provides a set of macros and flags (defined in `sys/stat.h`) to check the file type and permissions (`st_mode`)

- **File type macros:**
  - `S_ISLNK(mode)`: test for a symbolic link
  - `S_ISREG(mode)`: test for a regular file
  - `S_ISDIR(mode)`: test for a directory
  - `S_ISCHR(mode)`: test for a character device
  - `S_ISBLK(mode)`: test for a block device
  - `S_ISFIFO(mode)`: test for a pipe or FIFO
  - `S_ISSOCK(mode)`: test for a socket
- **Permission flags** (use them with bitwise operators `|` `&` `~` `^`):
  - `S_IRWXU`: read, write, execute/search by owner (`0x00700`)
  - `S_IRWXG`: read, write, execute/search by group (`0x00070`)
  - `S_IRWXO`: read, write, execute/search by others (`0x00007`)

`S_I` { `R` / `W` / `X` } { `USR` / `GRP` / `OTH` } : { Read / Write / Execute } by { Owner / Group / Others }

Permissions

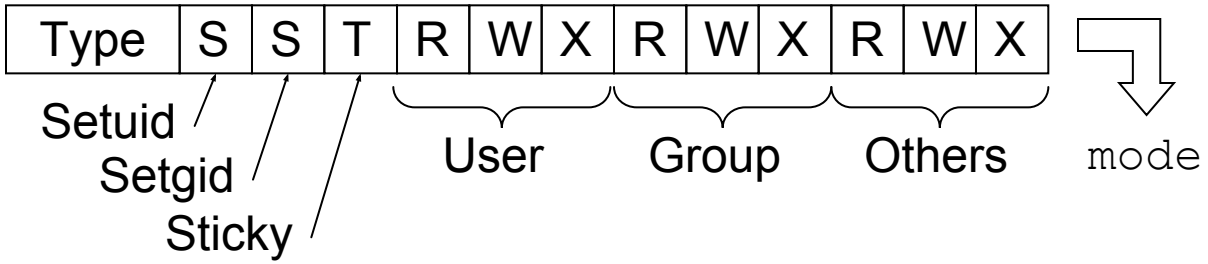
<sys/types.h>
<sys/stat.h>

SV+BSD+POSIX

- Change permissions of a file (file type cannot be changed):

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

  - Modification is usually done by reading current mode and performing bitwise operations
  - The process' effective UID must be 0 (root) or match the owner of the file
  - Error codes:
    - `EPERM`: Permission error
    - `EROFS`: Read only file system
    - `ENOENT`: The file does not exist
    - `EIO`: An I/O error occurred
    - `ENOTDIR`: A component of the path is not a directory
    - `ELOOP`: Too many symbolic links



Permissions

<unistd.h>
SV+BSD+POSIX

- Check user's permissions for a file:

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

  - Argument `mode` is a combination (bitwise OR) of the following flags:
    - `R_OK`: The file exists and grants read permission
    - `W_OK`: The file exists and grants write permission
    - `X_OK`: The file exists and grants execute permission
    - `F_OK`: The file exists
  - The full path is considered
  - The check is done using the calling process's real IDs, rather than the effective IDs as it is done when actually attempting an operation on the file
  - The function call will fail if any of the permissions is denied

File Opening and Creation

<sys/types.h>
<sys/stat.h>
<fcntl.h>

SV+BSD+POSIX

- Open and/or create a file or device:

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

  - `path`: The path of the file or device
  - `flags`: One of the following flags is mandatory:
    - `O_RDONLY`: Read only access
    - `O_WRONLY`: Write only access
    - `O_RDWR`: Read and write access
  - `mode`: Determines the permissions to create the file with (needed with `O_CREATE`). Permissions are modified by the process `umask`
    - In octal notation (preceded by 0 in C/C++)
    - As a bitwise OR (`S_IRWXU = 00700`, `S_IRUSR = 00400`...)
    - (see file attributes)
  - It returns a file descriptor (shared through `fork` and `exec`), setting the access pointer to the start of the file
    - The file descriptor is the lowest one available in the system

File Opening and Creation

- Other flags:
  - `O_CREAT`: Create file if it does not exist (with `mode` permissions)
  - `O_EXCL`: Error if `O_CREAT` is set and the file exists (*exclusively create*)
  - `O_TRUNC`: If the file already exists and is a regular file and the open mode allows writing, it will be truncated to length 0
  - `O_APPEND`: Before each write, the file offset is positioned at the end of the file. This may lead to corrupted files on NFS file systems if more than one process appends data to a file at once
  - `O_NONBLOCK`: When possible, the file is opened in nonblocking mode. Neither the call to open nor any subsequent operations will cause the calling process to wait
  - `O_SYNC`: The file is opened for synchronous I/O, blocking any call to `write` until the data has been physically written to the underlying hardware

File Opening and Creation

<sys/types.h>
<sys/stat.h>

SV+BSD+POSIX

- Set file mode creation mask (`mask & 0777`)

```
mode_t umask(mode_t mask);
```

  - The permissions specified in `open(2)`, are not directly applied to the created file
  - Permissions of a new file are established in the following way:

Permissions = `mode & (~umask)`

Example:

```
0666 & (~022) = 110 110 110 & 111 101 101 = 110 100 100 = 0644
- rw- rw- rw- & (~--- -w- -w-) ⇒ - rw- r-- r--
```

  - Used by `open`, `mkdir`, and other system calls that create files to modify the permissions placed on newly created files or directories
  - This system call always succeeds and the previous value of the mask is returned

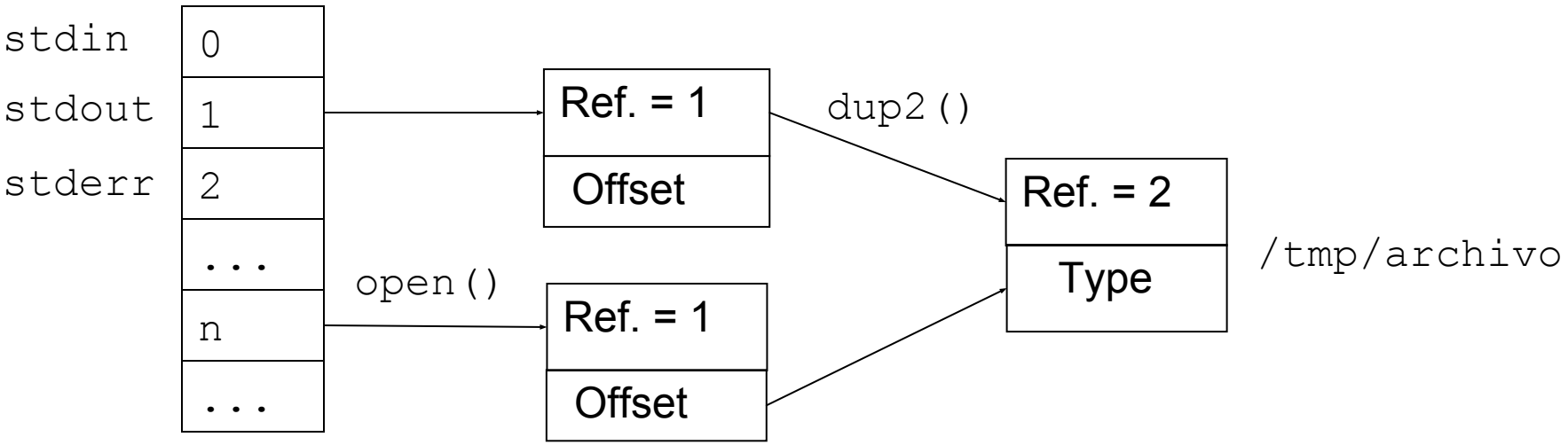
File Descriptor Duplication

<unistd.h>
SV+BSD+POSIX

- Duplicate a file descriptor:

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

  - Both file descriptors will refer to the same open file, and thus share file locks, file offset and file status flags
  - `dup()` returns the lowest-numbered unused file descriptor
  - In `dup2()`, `newfd` will refer to `oldfd` and `newfd` will be closed if open
  - Error codes:
    - `EBADF`: `oldfd` isn't open or `newfd` is out of the allowed range
    - `EMFILE`: Maximum number of open file descriptors has been reached



File Read and Write

<unistd.h>
SV+BSD+POSIX

- Write, read, repositioning and closing files:

```
ssize_t write(int fd, void *buf,
              size_t count);
ssize_t read(int fd, void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close(int fd);
```

  - Do not mix these system calls with library functions (e.g. `fopen`, `fread`, `fwrite`… from `stdio.h`, or streams in C++)

- File writing is done through the Buffer/Page Cache, providing efficient access. To prevent losing information in case of system crash or power failure:
  - Use the `O_SYNC` flag in `open()`, which synchronizes file and disk on every write
  - Perform file synchronization explicitly with the following system call:

```
int fsync(int fd);
```

    - The call blocks until the device reports that the transfer has been completed

Hard and Symbolic Links

<unistd.h>

SV+BSD+POSIX

- Create a new link (hard link) to an existing file:

```
int link(const char *old, const char *new);
```

  - If `new` exists it will not be overwritten
  - Hard links can be made only on files in the same file system

- Make a symbolic link (soft link or symlink) to a file:

```
int symlink(const char *old, const char *new);
```

  - If `new` exists, it won't be overwritten
  - Symbolic links can be made to files in a different file system, or even to a non existing file

- Read the contents of the symbolic link path:

```
int readlink(const char *path, char *b, size_t bs);
```

  - Size of link path can be determined with `lstat`
  - It does not append a `NULL` character to `b`
  - In case `b` is too small, it will truncate the contents to `bs` characters

File Removal

<unistd.h>
SV+BSD+POSIX

- Delete a name and possibly the file it refers to:

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

  - It deletes the directory entry and decrements the number of links (references) in the inode
  - If the number of links reaches 0 and no processes have the file open, the file is deleted and the space it was using is made available for reuse
  - If any processes still have the file open, the file (fifo, socket or device) will remain in existence until the last file descriptor referring to it is closed

File Control

<unistd.h> <fnctl.h>
SV+BSD+POSIX

- Manipulate a file descriptor:

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

  - Argument `cmd` determines the operation to be performed on the file:
    - `F_DUPFD`: Find the lowest numbered available file descriptor greater than or equal to `arg` and make it be a copy of `fd` (similar to `dup2`, which uses exactly the file descriptor specified)
    - `F_GETFD`: Read the file descriptor flags (only *close-on-exec*)
    - `F_SETFD`: Set the file descriptor flags to the value specified by `arg`
    - `F_GETFL`: Get the file access mode and the file status flags
    - `F_SETFL`: Set the file status flags to the value specified by `arg` (on Linux, only `O_APPEND`, `O_ASYNC`, `O_DIRECT`, `O_NOATIME` and `O_NONBLOCK`)

File Control: Locks

- Acquire, release, and test for the existence of record locks:

```
int fcntl(int fd, int cmd, struct flock *lock);

struct flock {
    short l_type;   /* F_RDLCK, F_WRLCK or F_UNLCK */
    short l_whence;/* SEEK_SET, SEEK_CUR or SEEK_END */
    off_t l_start; /* Start of locked region */
    off_t l_len;   /* Length in bytes (0=until EOF) */
    pid_t l_pid;   /* Process blocking it (only F_GETLK) */
    ...
};
```

- Lock types:
  - **Read or shared lock** (`F_RDLCK`): A process is reading the locked region, so it cannot be modified. Any number of processes may hold a read lock on a file region
  - **Write or exclusive lock** (`F_WRLCK`): A process is writing in the locked region, so it can not be read or modified. Only one process may hold a write lock on a file region

File Control: Locks

- Argument `cmd` can take the following values:
  - `F_GETLK`: If the lock described by `lock` could be placed, it returns `F_UNLCK` in the `l_type` field. If one or more incompatible locks would prevent this lock being placed, then it returns details about one of these locks in the `l_type`, `l_whence`, `l_start`, and `l_len` fields and sets `l_pid` to be the PID of the process holding that lock
  - `F_SETLK`: Acquire (when `l_type` is `F_RDLCK` or `F_WRLCK`) or release (when `l_type` is `F_UNLCK`) the lock described by `lock`. If a conflicting lock is held by another process, it returns -1 and sets `errno` to `EACCES` or `EAGAIN`
  - `F_SETLKW`: As for `F_SETLK`, but if a conflicting lock is held on the file, then it waits for that lock to be released
- Locks are advisory, so `read` and `write` do not actually check to see whether there are any locks in place (Linux also provides non-POSIX mandatory locks)
- Locks are associated to processes and are not inherited by child processes (Linux also provides non-POSIX open file description locks, which are inherited)
- Active locks can be consulted in `/proc/locks`
- `flock(2)` and `flock(1)` provide old-style (non POSIX) UNIX file locking
- `lockf(3)` provides an interface for record locking on top of `fcntl`

Directory Access

<sys/types.h> <dirent.h>
SV+BSD+POSIX

- Open a directory:

```
DIR *opendir(const char *name);
```

  - It opens the directory corresponding to `name`, and returns a pointer to a directory stream, which is positioned at the first entry in the directory
  - `DIR` data type is similar to `FILE` type, specified by the standard IO library

- Read a directory:

```
struct dirent *readdir(DIR *dirp);
```

  - It returns the next directory entry (`dirent` structure) in `dirp`, and it returns NULL on reaching the end of the directory stream or if an error occurred
  - The only field mandated by POSIX is `d_name`, of unspecified size, with at most `NAME_MAX` characters preceding the terminating null byte

- Close a directory:

```
int closedir(DIR *dirp);
```

  - It closes the directory stream associated with `dirp`, which is not available after this call

Directory Creation and Removal

- Create a directory:

```
int mkdir(const char *path, mode_t mode);
```

<sys/stat.h>
<sys/types.h>
SV+BSD+POSIX

  - `mode` contains the permissions to create the directory (modified by `umask`)
  - The new directory will be owned by the EUID and EGID of the process, but if the parent directory has the *setgid* bit set, the new directory will inherit the group ownership from it

- Remove a directory:

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

<unistd.h>
SV+BSD+POSIX

- Change the name or location of a file (or directory):

```
int rename(const char *old,
           const char *new);
```

<stdio.h>
BSD+POSIX

  - If `new` exists, it is removed before associating it to `old`
  - If `new` is a directory it must be empty
  - `old` and `new` must be of the same type and belong to the same file system
  - If `old` refers to a symbolic link, the link is renamed; if `new` refers to a symbolic link, the link will be overwritten