



Scuola Superiore Sant'Anna



File System

Giuseppe Lipari



File system



File system interface

- The concept of file
 - A file is a collection of bytes recorded on a mass storage device
 - A file can contain
 - Source code of programs
 - Executable programs
 - Text
 - Various kind of data (binary)
 - The file is the minimum “unit” of allocation of data in a device
 - To access a file we must be able to interpret its content!



Attributes

- Basic attributes of a file
 - Name
 - Type
 - (text, binary, sound stream, video stream, etc)
 - Location
 - Size
 - Access permissions
 - User identification
 - It can include group identification
 - Date and time
 - (of creation, of last modification, of last access)



Basic operations on a file

- The following operations can be performed on a file
 - Creation
 - Find space on the device
 - Allocate space and update structures
 - Writing
 - The size of a file is dynamic! This can create problems in the allocation policy
 - Reading
 - Positioning (lseek)
 - In all cases, we need a pointer to the current position in the file
 - Deletion
 - We need to deallocate the space and update the structure
 - Truncament
 - Reduce size



Opening a file

- What does it mean to open a file?
 - The file is normally memorized on the mass storage
 - Opening a file is needed for:
 - Creating data structures in the operating system for accessing the file (tables, pointers, etc.)
 - Registering that a file is being used by some process
 - Thus, it cannot be deleted by another process!
 - However, a file can be opened by many processes at the same time

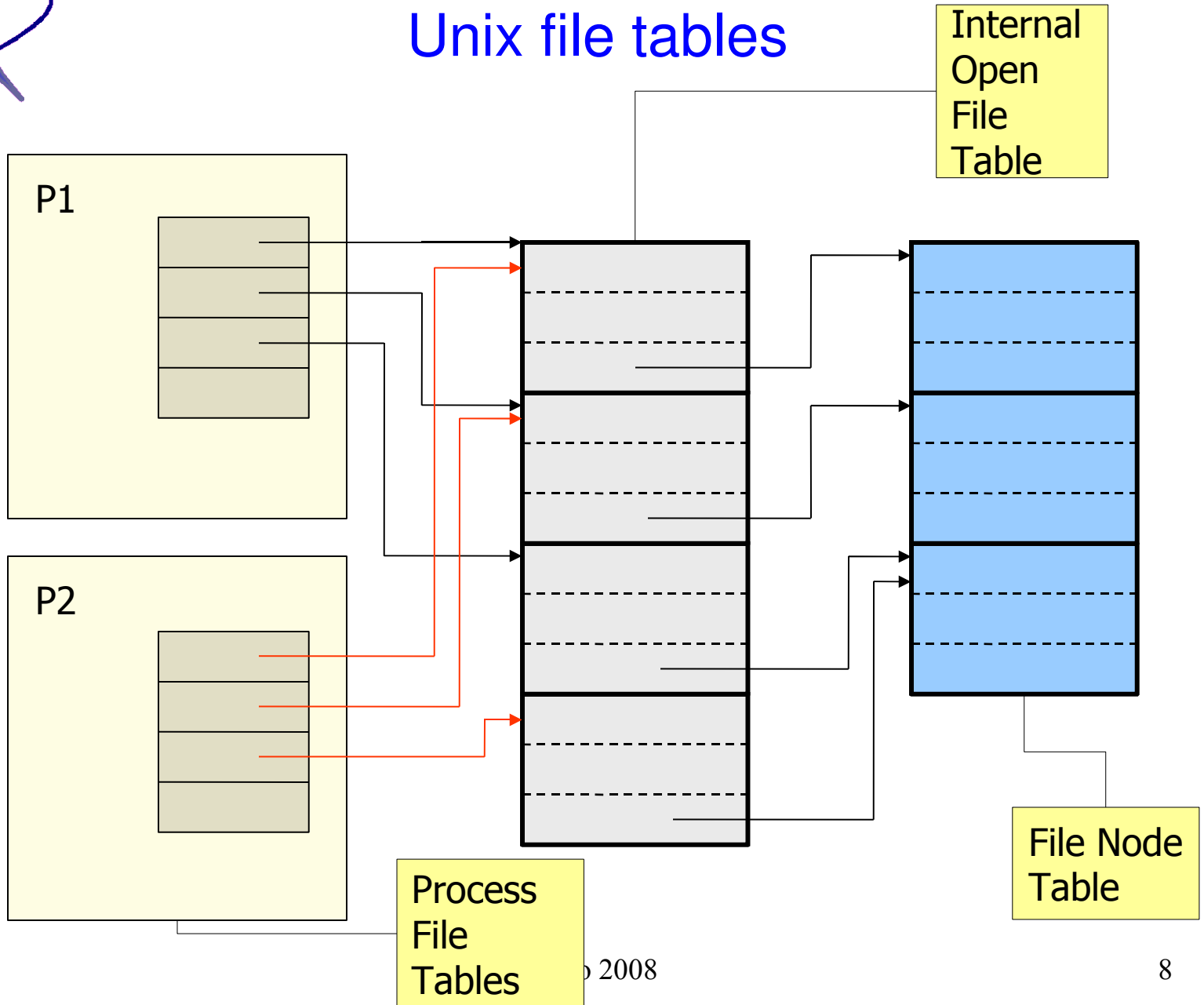


File tables

- Two levels of tables
 - Process's file table
 - There is one such table for each process
 - The table contains “file descriptors”
 - A “file descriptor” contains the pointer to “internal open file descriptor” residing in the open file table
 - Open file table
 - Each time an `open()` system call is invoked, one new “internal open file descriptor” is created and memorized in the “open file table”
 - An “internal open file descriptor” contains a pointer to the “file node descriptor”
 - File node table
 - This table contains one “file node descriptor” for each open file in the system
 - Each file node descriptor contains a counter to the number of “internal open file descriptors” pointing to the node



Unix file tables





File types

- The file type is an indication of how data is stored on the file
 - In windows, each file has an extension that reveals the file type
 - .exe, .bat, .doc, .xls, .avi, etc.
 - A program uses this information to understand the format of the data
 - It is only an indication, not embedded in the OS
 - In Macintosh, each file is associated an identifier of the program that created it
 - When double-click on a file, the appropriate program is automatically invoked
 - This association is embedded in the OS



Binary and text

- Binary files
 - A binary file is simply a stream of bytes
 - The format and the meaning of the bytes are up to the program
- Text files
 - A text file is a sequence of ASCII characters
 - Special characters are CR and LF
 - Not all OS encode text in the same way
 - In DOS and Windows, we need both the CR and the LF
 - In Unix, only LF is enough
 - Therefore, when transferring a file from a Unix system to a DOS system, and viceversa, we can have problems!
 - Some program (e.g. ftp) automatically converts text files among the two formats, if necessary
 - Always specify the format (binary or text) when transferring a file with FTP!!



Access method: sequential

- The simplest method for accessing a file is “sequentially”
 - Each “internal open file descriptor” has an “offset” field
 - records the current position in the file
 - Every read/write operation updates this field
 - The file is seen as a “tape” in which you can do play, rewind, fast forward
 - All OS support sequential access



Direct access

- The file has a particular structure
 - Each file consists of a number of “records” of fixed length
 - A record is a collection of data of some type
 - Example:
 - A record in a phone book file contains:
 - Name, surname, phone, address
 - The file is a collection of records
 - In the direct access method, it is possible to read or write one particular record without reading the previous ones



Direct access

- Example:
 - If every record has a size of 100 bytes:
 - The first record is at offset 0
 - The second record is at offset 100
 - ...
 - The i -th record is at offset $(i-1)*100$
- Direct access is very useful for databases
 - Some OS supports direct access
 - For example, the read function has one more parameter `read(fd, record, i)` where i is the record number
 - In most OS it is necessary to use a sequential file and the `lseek()` and `read()` functions



Indexed access

- Based on direct access, is the indexed access
 - Each record has a “key field”
 - directly from databases concepts
 - Each file contains at the beginning an “index” with the records’ positions and the keys
 - By inspecting the index, it is possible to know at which position a certain record is
 - This feature can be implemented directly in the OS or in the application



Directory structure

- A disk is usually structured in partitions and directories
- Partitions
 - A disk may contain one or many “partitions” or “volumes”
 - A partition is a virtual disk
 - In many OS it is possible to define a partitions that contains many disks
 - Each partition contains a directory structure
- Directory
 - Contains a list of files or of other directories
 - It is organised in a structure (tree or acyclic graph)



Directory

- Directories are organised in a structure
- Tree structure
 - Root directory: contains all files and subdirectories
 - Any directory can contain an arbitrary number of subdirectories
 - To create/delete directories, special system calls are used
 - Every file has a “parent directory”
 - Every directory (except the root) has a “parent directory”
- The directory organisation has nothing to do with how the files are allocated on the disk
 - Two files in the same directory are not necessarily contiguous



Directories in Unix

- In Unix, a directory is a special text file
 - It contains the list of files, one per each line
 - Try to open the directory with vi!
- However, it cannot be read or written as a normal file



Current working directory

- Each process has a current working directory (CWD)
 - When opening a file, we can specify the file name as
 - a relative path (i.e. starting from the CWD)
 - an absolute path (i.e. starting from the root directory)
- File Creation
 - When a file is created, it is added to its parent directory
- File Deletion
 - When a file is deleted, it is removed from the parent directory
- Directory deletion
 - In some OS it is not possible to remove non-empty directories (regular *rm* command in Unix and *del* command in Windows)
 - In others, removing a directory implies to recursively remove all its contents (*rm -f* in Unix and GUI in Windows)

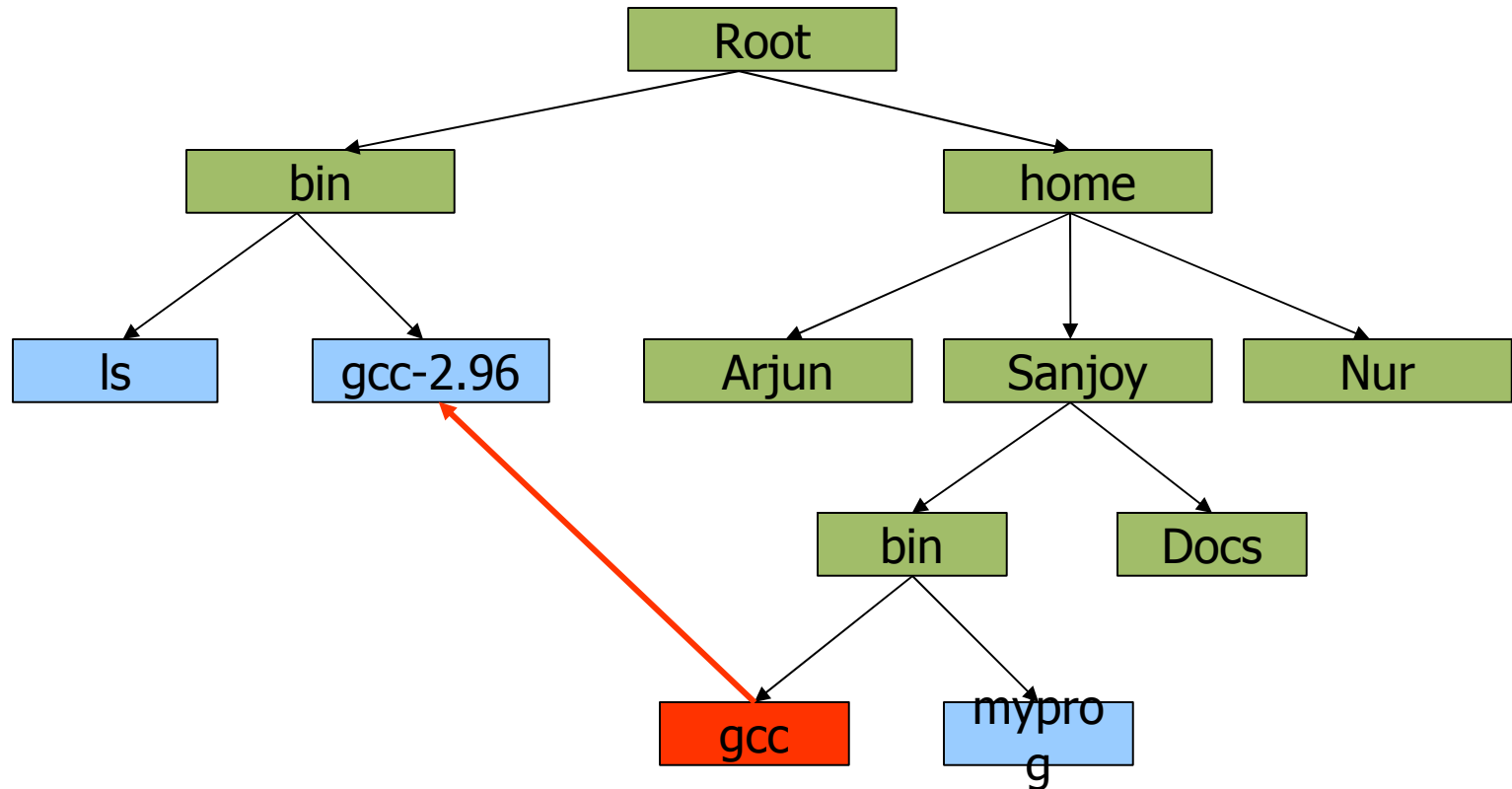


Acyclic directory structure

- In this structure it is possible to establish “links”
 - A file link is a “reference” to a file or a directory
 - It is possible to link any file or directory in any position
 - It is not possible to create cycles
 - A cycle would make the system too complex!
 - It is used for
 - “sharing” files and/or directories
 - to associating a symbolic name for a program



Example



- gcc is a symbolic link to gcc-2.96



Basic operations with links

- Creating a symbolic link
 - In Unix with command `ln`
 - `ln -s <link name> <file name>`
- Deletion of a file
 - Should we also delete the link?
 - We need
 - either a list of backward link (i.e. the linked file points to all symbolic links)
 - or to search the file system for pending links
 - Too much overhead in both cases



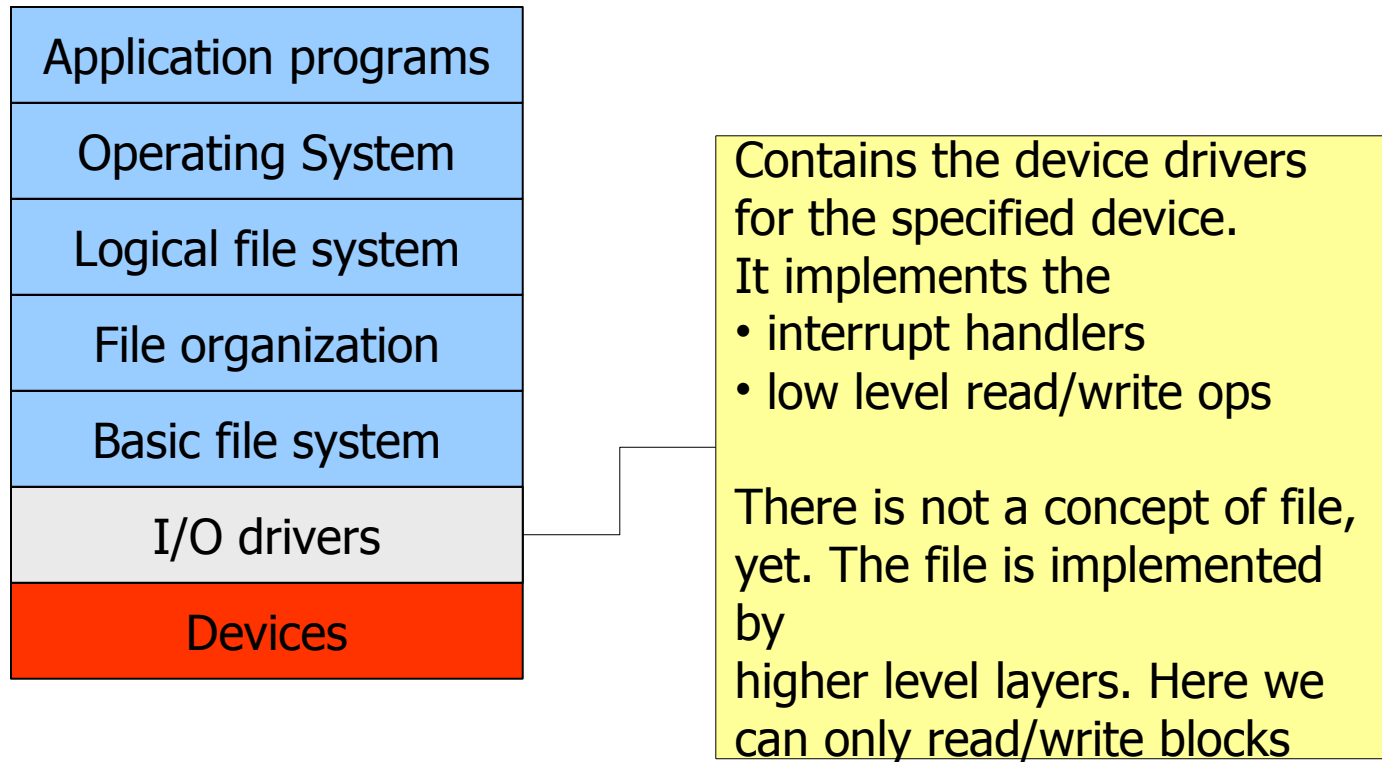
Deletion of a file with links

- Can we delete the file if there is a symbolic link to it?
 - In some implementation,
 - the file keeps a reference counter of how many symbolic links are pointing to it
 - when the reference counter goes to 0, the file is effectively deleted
 - In most implementations, the link remains “pending” and an error is raised if we try to access it



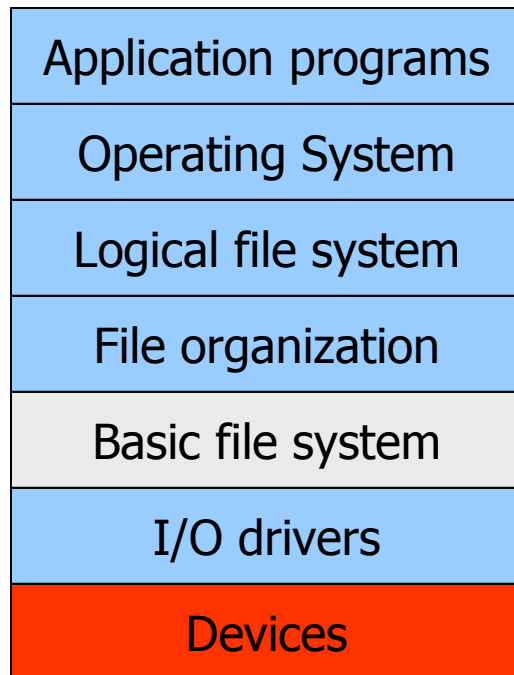
Implementation of a file system

- The implementation is usually done in “layers”





Basic file system layer



It is also called
"Virtual Volume Layer". Every disk
is seen as a linear sequence of
blocks.

This level translates linear block
numbers into disk coordinates for
the I/O driver

Example: block number 2035 is
located on track 12, sector 35,
level 2.

This also depends on the
disk, so it could be considered part
of the device driver



Basic file system layer

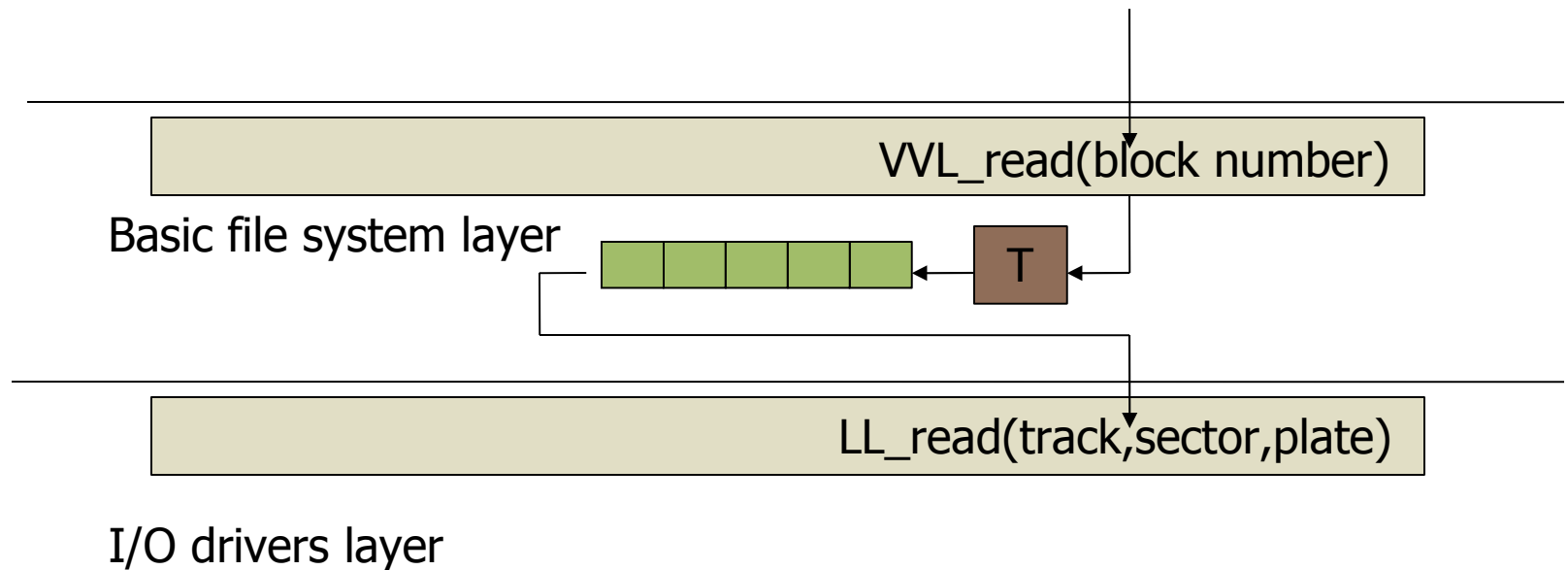
- This layer implements:
 - Translation from linear block number to block coordinates on the disk
 - Scheduling algorithm
 - For example, SCAN
 - The scheduling algorithm can be done only in this level, because here we know the physical of the disk (i.e. arm direction)
 - Therefore, in this level we organize the requests for disk access in one or more queues, for example with SCAN



Interface example

- The block number requested from the upper layer is transformed in a block coordinate. Then, the request is inserted in a SCAN-ordered queue. An internal process, reads from the queue and serves the requests

File organization layer



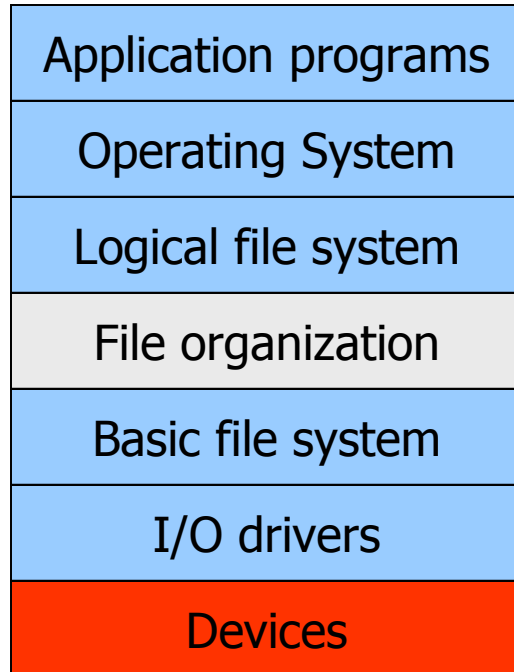


Basic File System Layer

- It provides the following functions
 - VVL_read(block number)
 - translates the block number into coordinates
 - invokes the LL_read()
 - VVL_write()
 - translates the block number into coordinates
 - invokes the LL_write()



File organization layer



In this layer, each file is divided into logical blocks. Each logical block is then allocated on a physical block on the disk. Notice the similarity with the paging system!

Several ways of allocating. Whenever possible, consecutive blocks of a file are allocated as consecutive blocks on the disk.

This layer also implements partition tables



File organization layer

- This layer provides the concept of file
 - However, we do not have the “file name” yet
 - Each file is identified by the pointer to the file descriptor
 - Each file is associated at least the first block on the disk
- From this layer up, we will only see the concept of file!
 - This layer (and the upper ones) manipulate the i-node (the descriptor of the physical file on the disk)



Partitions

- Every partition on the disk is a set of consecutive blocks
- Example
 - A disk contains 10,000,000 blocks (sectors)
 - Partition 1: [0 – 9,999] (swap)
 - Partition 2: [10,000 – 4,999,999] (hda0)
 - Partition 3: [5,000,000 – 9,999,999] (hda2)
- The partition table is stored on the disk itself, in sector 0
- To view/edit a partition table:
 - fdisk



File organization layer

- This layer also provides an allocation algorithm
 - Where the file's blocks are allocated on the disk
 - The algorithm is very important for performance
 - This layer maintains the information on which blocks on the disk are used and which are not
 - This information is stored on the FAT (file allocation table)
 - It is a particular structure that is stored in some fixed position in the disk (for example tracks 0 and 1)
 - The FAT is partially loaded in memory, for faster access



The FAT

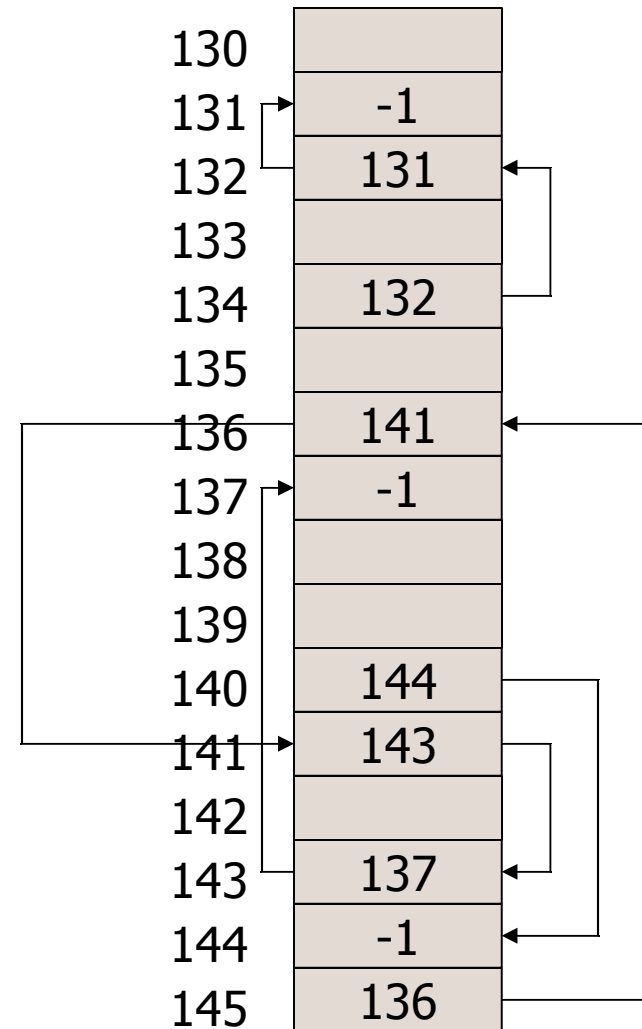
- The MS-DOS FAT is implemented in the following way
 - At the beginning of each partition, we find the FAT
 - The FAT is a table that contains one element for each block in the disk, plus an index table
 - Each entry in the index table contains the name of the file and its first block number
 - The entry in the FAT corresponding to the block number, contains the number of the second block
 - and so on
 - Basically, the FAT is a linked list
 - Non used blocks have 0



The FAT

Index Table

data.txt	134
movie.mpg	145
music.mp3	140





Other implementations

- To store the free blocks we can use:
 - A vector of bits
 - 0 if the block is free
 - 1 if the block is occupied
 - Not very efficient
 - A linked list
 - each free block contains the address of the next free block
 - Group of blocks
 - the first free block contains the list of the first N free blocks
 - Of these, the last contains the address of the next N free blocks
 - and so on...



File organization layer

- Two allocation methods
 - Concatenated allocations
 - Each block contains the information on its next block
 - Disadvantage: only sequential access, to know where logical block 100 is, the system must read all 99 previous blocks
 - Indexed allocation
 - Each file is associated a block table (usually in the first block)
 - The table contains the physical block number of each logical block of the file
 - Allows direct access to the file
 - Disadvantage: requires some extra space for each file (1 block?)

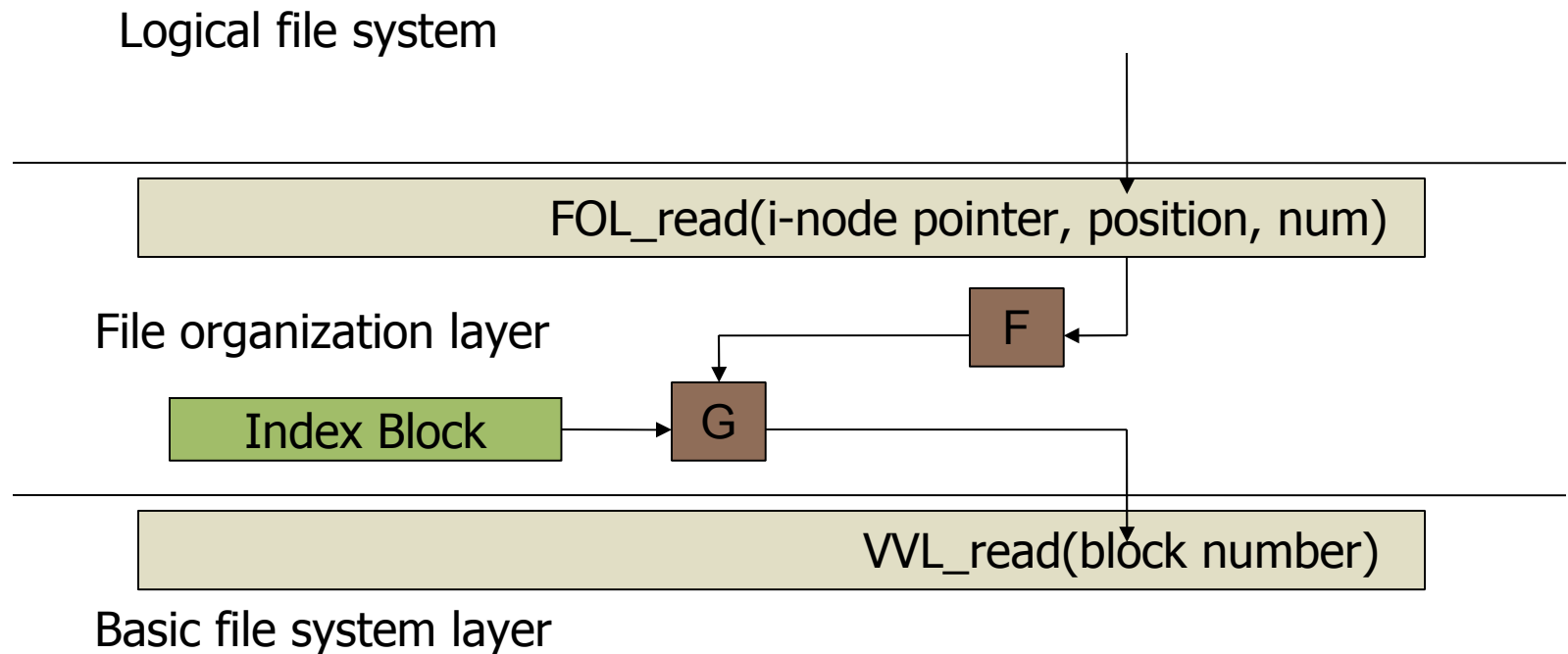


Indexed allocation

- A file have one (or more) index blocks
 - usually the first block of the file does not contain data but the index block
 - In case more index blocks are needed (for very long files), index blocks can be concatenated
- This allocation is the most used
 - althought in many different flavours

Interface

- Function F computes the logical blocks of the file to be read
- Function G computes the physical blocks on the disk (reading from the index block of the file)



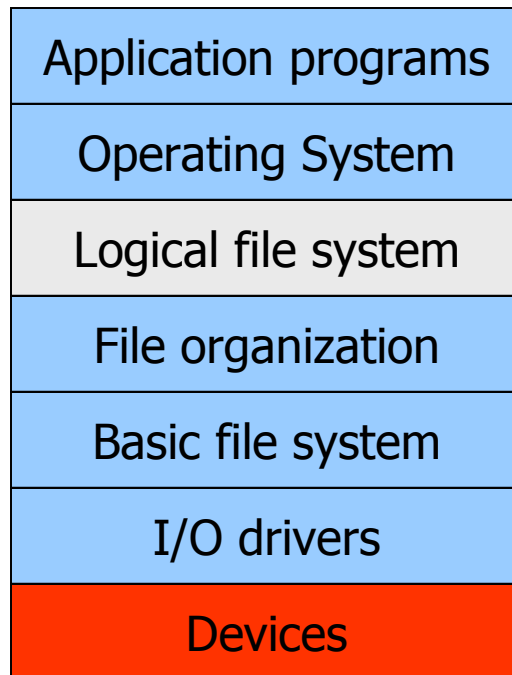


Interface

- Functions provided by this layer
 - FOL_create()
 - creates a new file, allocating some space for it on the disk
 - FOL_remove()
 - removes an existing file, freeing the space
 - FOL_open()
 - loads the first block of a file in memory for indexed access
 - FOL_close()
 - free the memory occupied by that file
 - FOL_read() and FOL_write()



Logical file system layer



This layer provides several functionalities:

- access control and locking
- naming
- catalogs (directories)
- file tables

This layer provides an interface to the application programs!

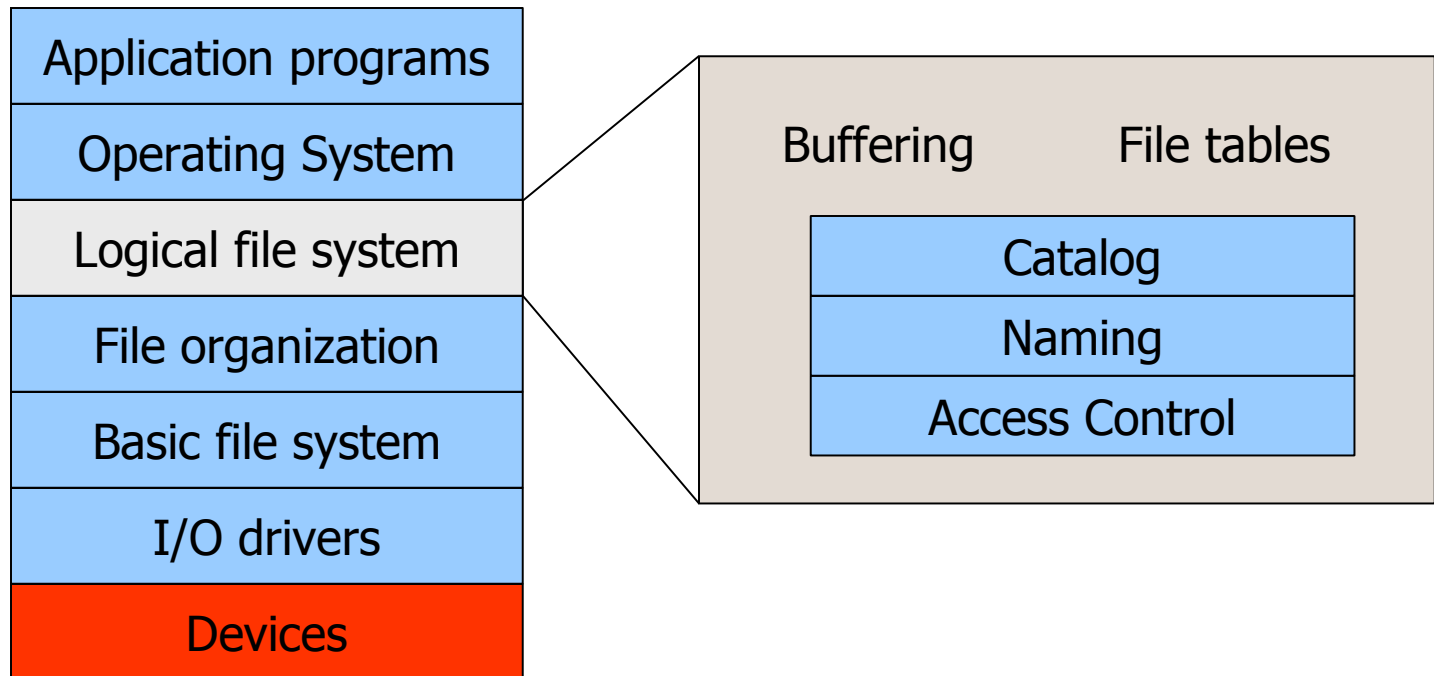


Logical file system layer

- We divide it into 3 sub-layers
 - Access control and locking
 - It is used to check if a certain operation on a file can actually be done
 - Naming
 - Each file is associated with a name
 - The name is not unique!
 - Catalog
 - Provides the “directory” facility
- Then, the Logical File System Layer itself provides additional features
 - buffering
 - file tables



Sub-layers





Access control sublayer

- It is used to check if a certain operation on a file can actually be done
 - First, read/write permissions are checked
 - If the permission is not granted, the function returns with an error
 - Then, locks are checked
 - A lock can be on a file, or on part of a file
 - If the file (or part of a file) is locked by some process, access by other users is not possible
 - An error is returned or the process is blocked



Interface

- The following functions are defined
 - `ACL_lock_r()`
 - Locks a file for reading
 - Other process can read
 - No other process can write
 - `ACL_unlock_r()`
 - Unlocks the file for reading
 - `ACL_lock_w()`
 - Locks a file for writing
 - No other process can access the file
 - `ACL_unlock_w()`
 - Unlocks a file for writing



Naming sublayer

- This sub-layer associates names to files
 - the symbolic names are stored in the i-node itself
- This sublayer provides the LSF_open() function
 - the LSF_open() returns an id (the index in the “internal open file table”)
 - It creates the “internal open file table”, and calls the FOL_open() that will create the i-node (if it does not exists already)
 - All the other functions (LSF_read(), LSF_write(), etc) will refer to the id



Directories

- A third sublayer (catalog layer) is used to implement directories
 - Each directory is seen as a special file
 - This special file contains the list of the files and subdirectories
 - It can also be implemented as a more complex structure
 - This sublayer provides functions for manipulating directories



Interface

- The following functions are defined
 - CL_create_dir()
 - creates a new directory in the specified place
 - adds the directory name in the parent directory
 - CL_remove_dir()
 - removes the directory, also from the parent directory
 - CL_create_file()
 - adds the file to its parent dir
 - CL_remove_file()
 - removes the file from its parent dir



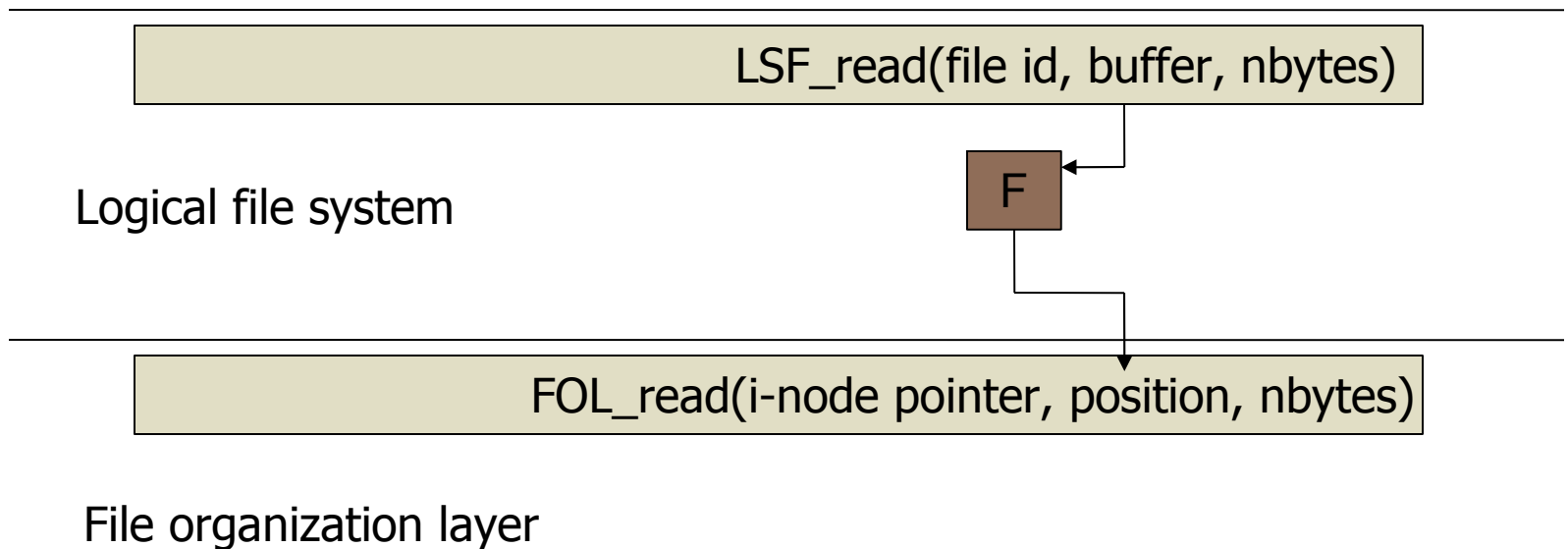
Interface of the Logical File System

- In addition to the sub-layer functions, the LFS layer provides:
 - LSF_read(file_id, buffer, nbytes)
 - this function, after obtaining the i-node address, invokes the FOL_read()
 - it also updates the current offset
 - LSF_write(file_id, buffer, nbytes)
 - this function, after obtaining the i-node address, invokes the FOL_write()
 - it also updates the current offset and the current size
 - LSF_seek(file_id, offset, how)
 - this function only updates the current offset
 - LSF_close(file_id)
 - This function calls the FOL_close and decrements the reference counter in the “internal file descriptor”. If the counter is 0, deallocates the file descriptor



Interface example

- Function F transforms the file id into a i-node pointer
- The position is the current offset





Logical File system layer

- This layer can also provide buffering
 - A LSF_read may or may not involve a call to FOL_read(), depending on the content of the buffer
 - Also, to fill the buffer, a LSF_read() of one byte can be transformed in a FOL_read() of many blocks
 - Finally, a double buffering mechanism can be provided here
 - an internal process can block on a circular buffer waiting to load a sequence of blocks from the disk



Operating system functions

- OS primitives
 - open()
 - creates the entry in the process' file descriptor table
 - calls the LSF_open() that creates the “internal open file descriptor”
 - read()
 - translates from file descriptor to file id
 - simply calls the LSF_read()
 - write()
 - translates from file descriptor to file id
 - simply calls the LSF_write()
 - lseek()
 - ...



Formatting

- A disk needs to be formatted before starting with operations
 - A formatting creates all structures in the disk that are used to
 - allocate blocks
 - index files
 - For example,
 - a clean FAT is created
 - a linked list of free blocks is created on the disk
 - The formatting also “marks” bad blocks (i.e. blocks that cannot be used safely)
 - another way of marking bad blocks in Unix is with *chkdisk*
 - In some special system, the formatting creates the structures for the error corrections
 - For example, in RAID



Swap space

- A portion of the disk is allocated to the swap space
 - To swap out processes
 - To swap out modified pages
- The swap space is usually in a separate partition
 - To avoid interference with other files
 - The swap space can be managed as
 - A single direct access file
 - A collections of files
- If the swap space is in a separate partition
 - The normal functions used for the files are not used for the swap operations
 - Special dedicated functions will load and unload pages
 - These functions use algorithms optimized for speed



Transaction – based file systems

- Some modern file systems are based on the concept of transactions
 - In old file systems, a crash of the OS, or another catastrophic event could leave the disk in an inconsistent state
 - For example, a file is half written
 - Some element of the FAT is not correctly linked
 - Some block is neither free nor occupied
 - etc.
 - Usually, when the system restarts, it check the consistency of the File tables on the disk and, in case it is not correct, runs a recovery algorithm
 - The recovery is long and tedious
 - Moreover, some other file could be damaged
 - To overcome these problems, the concept of Transaction-Based file systems is used
 - Taken directly from the database theory

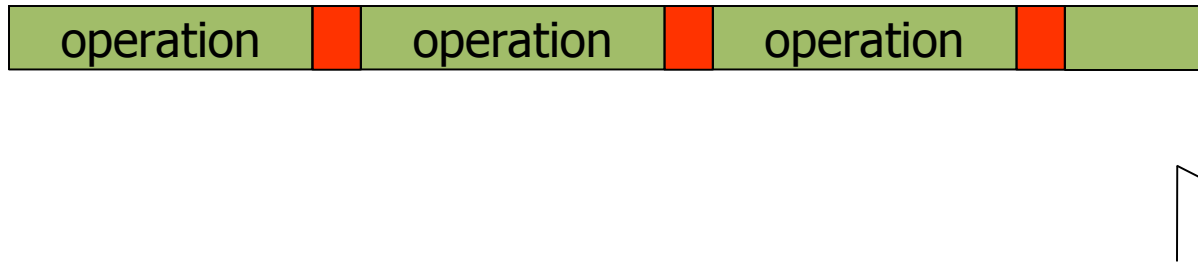


Transaction – based file systems

- In a transaction-based file system
 - read and write operations are “transactions”
 - Every operation consists of two parts
 - The operation itself
 - A commit phase
 - The commit phase is very short and can be considered “atomic”
 - i.e. it cannot be interrupted
 - In other words, operations cannot be left in a “incomplete” state
 - Only committed operation are valid



Transaction-based file system



- Only the first three operations are valid
 - The last operation is like it had never been started
- To implement this, we need to implement a write operation in a particular way
 - New blocks are written on the disk
 - The update in the “FAT” is part of the commit phase
- Implementation changes a lot from FS to FS
 - Examples in Linux: ReiserFS, ext3, etc.



Virtual file system

- In many operating systems the file interface is used also for accessing devices and sockets
 - e.g. Linux
- Depending on the name used in the open, it is possible to associate a file descriptor to a device
 - This is done in the naming sub-layer



Virtual file system

- In Linux the i-node is substituted by a v-node
 - virtual-node
- The v-node is a generalization of a i-node
 - The i-node is for files
 - The v-node can also be for devices
- The v-node contains a pointer to a device descriptor table
 - The device descriptor table contains appropriate functions pointers for open/read/write/close, etc.



Special files in Linux

- The `/dev/` directory contains special files
 - They are not actual files on the hard disk
 - Each file in `/dev/` is a device
 - In most cases,
 - by writing directly on the file, we can send data to the device
 - by reading directly from the file, we can read data from the device
 - Example:
 - `/dev/null` is a particular “device” whose “write” operation does nothing
 - Sending data to `/dev/null` is equivalent to sending them nowhere



The /proc directory

- Another special directory is /proc/
 - Again, it is not a real directory as it does not contain files
 - It contains information on the Operating system
 - For example, it contains a directory for each process
 - The directory contains several files, like the process id, the memory currently used by the process, etc
 - The ps and the top commands read their information from the /proc file system
 - How does it work?



The /proc directory

- Every time we “open” a file in the /proc directory
 - The naming subsystem understands it is a special file
 - It opens a special v-node that contains a pointer to a table of pointers to functions
 - open/read/write/etc.
 - Each call to read on that descriptor will be translated on a call to the appropriate function pointer
 - The function pointer points to a function that looks inside the operating system structures and returns the needed informations