UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Input / Output management

COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

Bioinformatics

2025/26 Spring Term

Jordi Fornés

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Contents

Basics Concepts of I/O

What’s I/O?

- **Definition**: information transfer between a process and the outside.
  - Data Input: from the outside to the process
  - Data Output: from the process to the outside

  (always from the process point of view)

- In fact, basically, processes perform computation and/or I/O
- Sometimes, even, I/O is the main task of the process:
  for instance, web browsing, shell, word processor

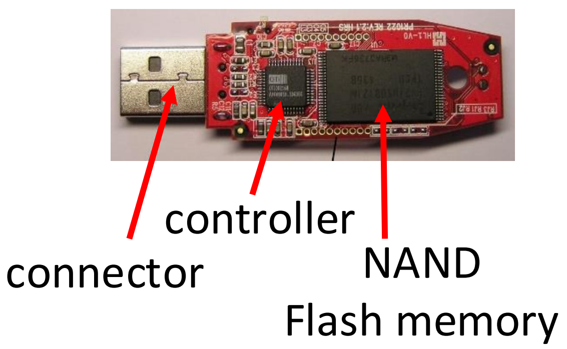
- **I/O management**: Device (peripherals) management to offer an usable, shared, robust and efficient access to resources

I/O Devices

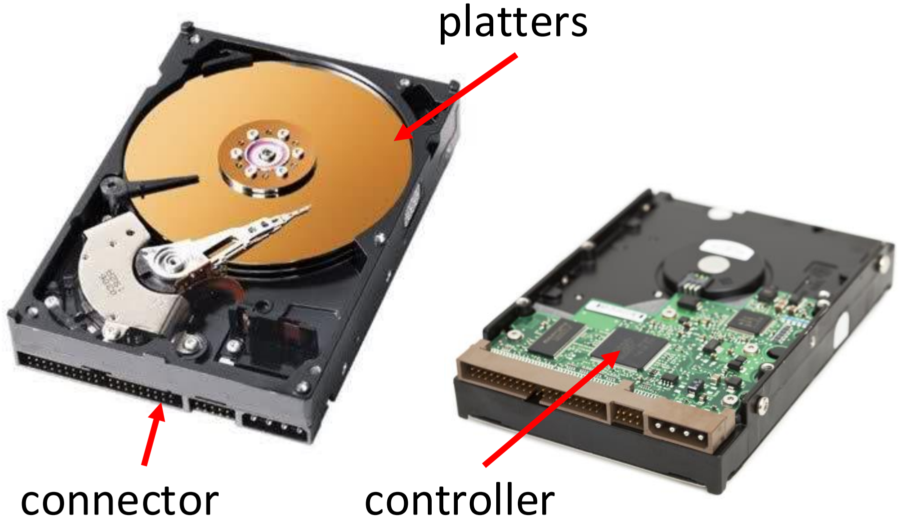
```
                    Windows

A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.

*  Press any key to terminate the current application.
*  Press CTRL+ALT+DEL again to restart your computer. You will
   lose any unsaved information in all applications.

              Press any key to continue _
```

Physical Storage Devices

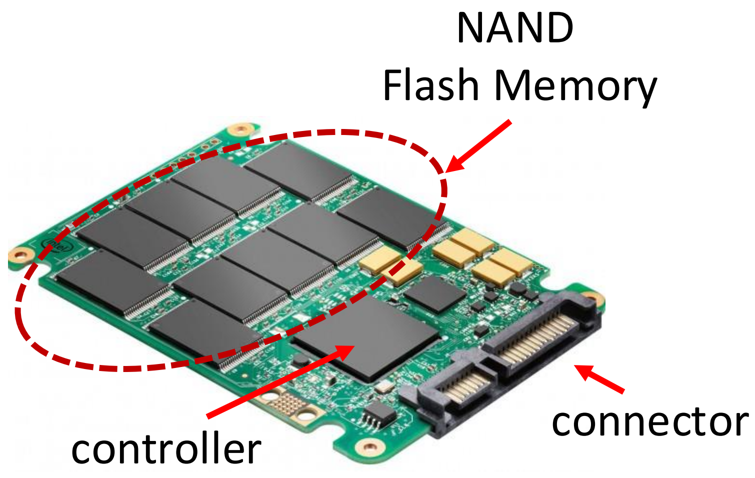
- Non-volatile memory to save data
- Similar vs Different components
  - Impact on performance and capacity



USB Drive

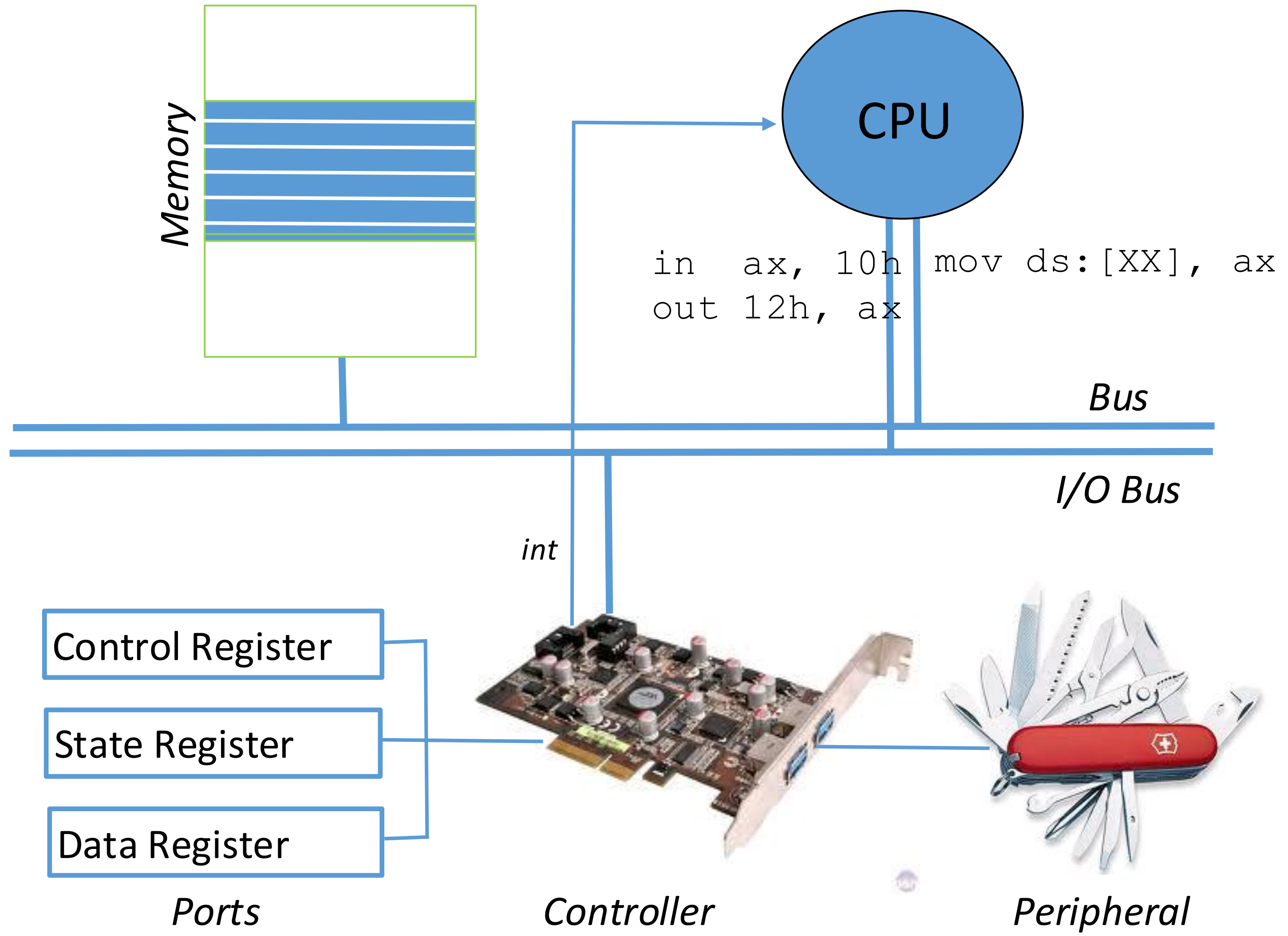


Hard Disk



Solid State Drive (SSD)

HW view : Accessing physical devices

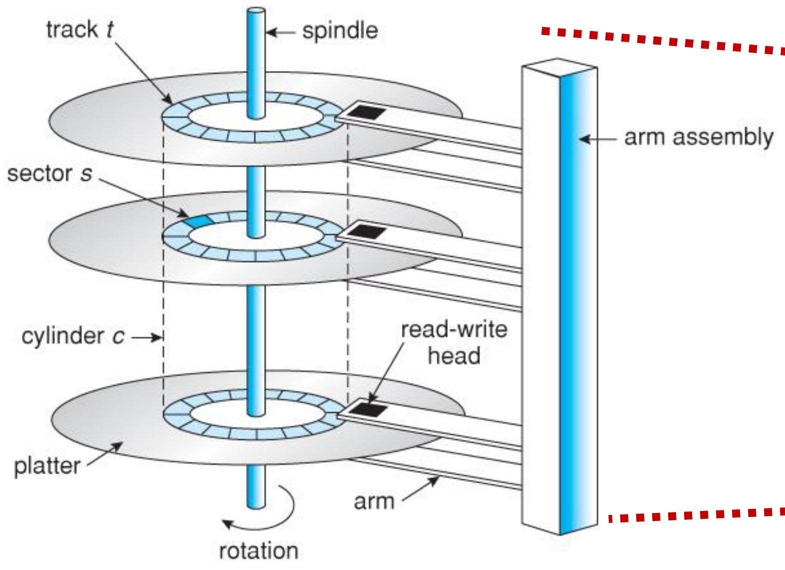


How are data physically saved?

- Any storage device needs to organize the pool of memory
  - E.g.: DVD, hard-disk, pen-drive, etc.
- Sector: The smallest unit of data that can be read/written
  - **Defined by the hardware**
  - Fixed size (tipically 512 Bytes)

Some parameters that impact on performance

Speed: rpm
Connector Bandiwth: Gbps



Basic system calls

Blocking and non-blocking operations

- Some I/O operations are time consuming
- A process cannot be idle in the CPU
  - OS blocks the process (RUN→BLOCKED)
- Default behaviour can be modified with the flag O_NONBLOCK

I/O system calls

os.open: Given a *pathname, flags* and *mode* returns an integer called the user *file descriptor*

os.read: Reads *n* bytes from a device (identified by the file descriptor) and saved in memory

os.write: Reads a bytestring from memory and writes them to the device (identified by the file descriptor)

os.close: Releases the file descriptor and and leaves it free to be reused

os.dup/dup2: Duplicates the file descriptor. Copies a file descriptor into the first free slot of the user file table. It increments the count of the corresponding file table entry, which now has one more fd entry that points to it.

os.pipe: Allows transfer of data between processes in a first-in-first-out manner

os.lseek: Changes the offset of a data file (an entry in the File Table pointed by the fd).

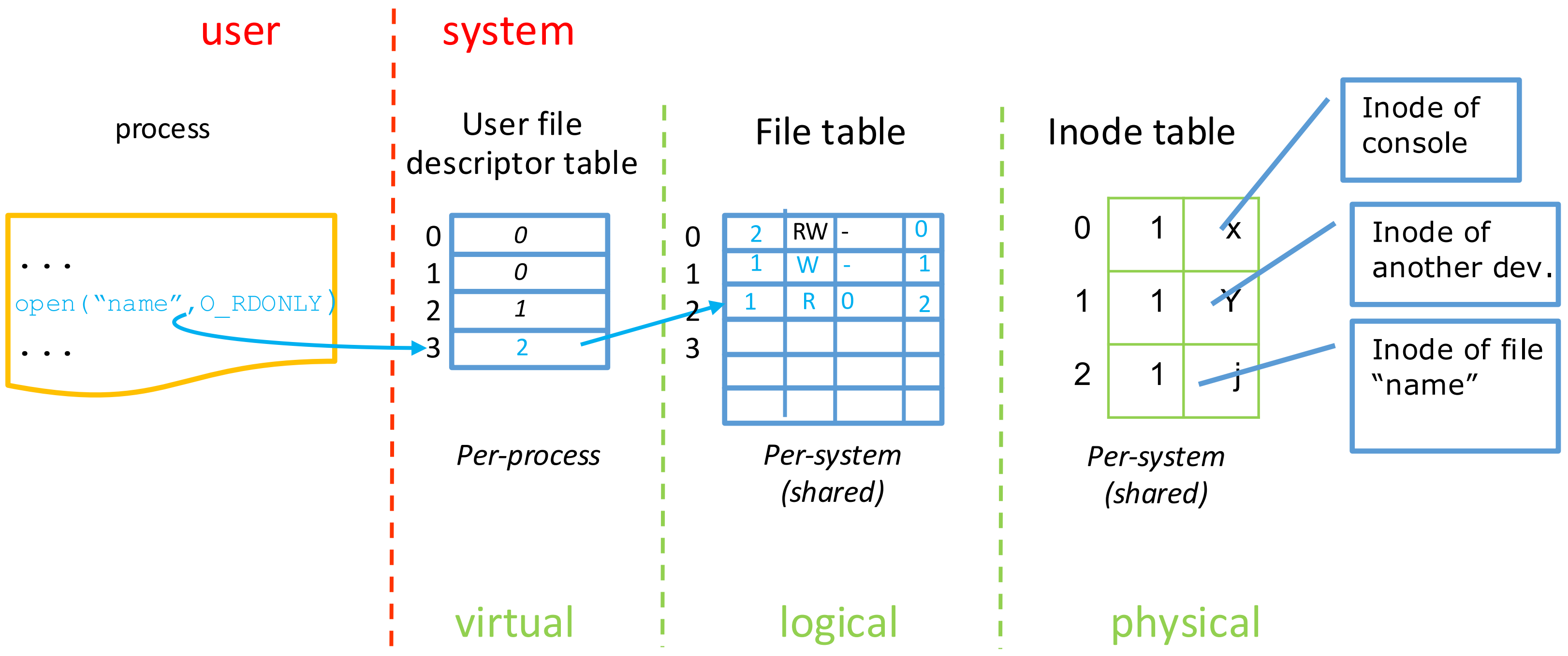
Syscalls open, read & write are blocking

Open

- So, how do you associate a name with a virtual device?
- **`fd = open(pathname, flags [, mode]);`**
  - **`open`** syscall links a device (file name) to a virtual device (field descriptor)
    - Is the first step that a process must take to access file data. It checks permissions. After correct completion, process can call *read/write* multiple times without check permissions again.
    - **`open`** returns a file descriptor. Other file operations, such as reading, writing, seeking and closing the file use the file descriptor.
  - **`pathname`** is a file name.
  - **`flags`** indicate the type of open. At least, one of them
    - `O_RDONLY` (reading)
    - `O_WRONLY` (writing)
    - `O_RDWR` (reading & writing)
  - **`mode`** gives the file permissions if the file is being created.

Open: data structure

- Open (cont): effects on the kernel data structures
  - The kernel allocates an entry in the file descriptor table. **<u>It will always be the first free entry.</u>** The kernel records the index of the File Table in this entry
  - The kernel **<u>allocates an entry in the file table</u>** for the open file. It contains a pointer to the in-core inode of the open file, and a field that indicates **<u>the byte offset</u>** in the file where the kernel expects the next read or write to begin.
  - The kernel associates these structures in the corresponding DD (`MAJOR` of the symbolic name). It may happen that different entries of the FT point to the same DD



Read

Num. of bytes actually read

File descriptor returned by open

Address of a data structure in the user process

Num. of bytes the user wants to read

- `n = read(fd, buffer, count);`
  - Asks for reading *count* bytes (characters) from the device pointed by *fd*
    - If there is great or equal *count* bytes available, it reads *count bytes*
    - If there is less than *count* bytes, it reads all of them
    - If there is no bytes, it's up to the device behaviour:
      - Blocking process until data available
      - Returns 0 immediately
    - If *EOF*, returns 0 immediately
      - The meaning of *EOF* it's up to the device behaviour
  - Returns n, the number of bytes actually read
  - The kernel updates the offset in the file table to the n; consequently, successive reads of a file deliver the file data in sequence

Write

Num. of bytes actually written

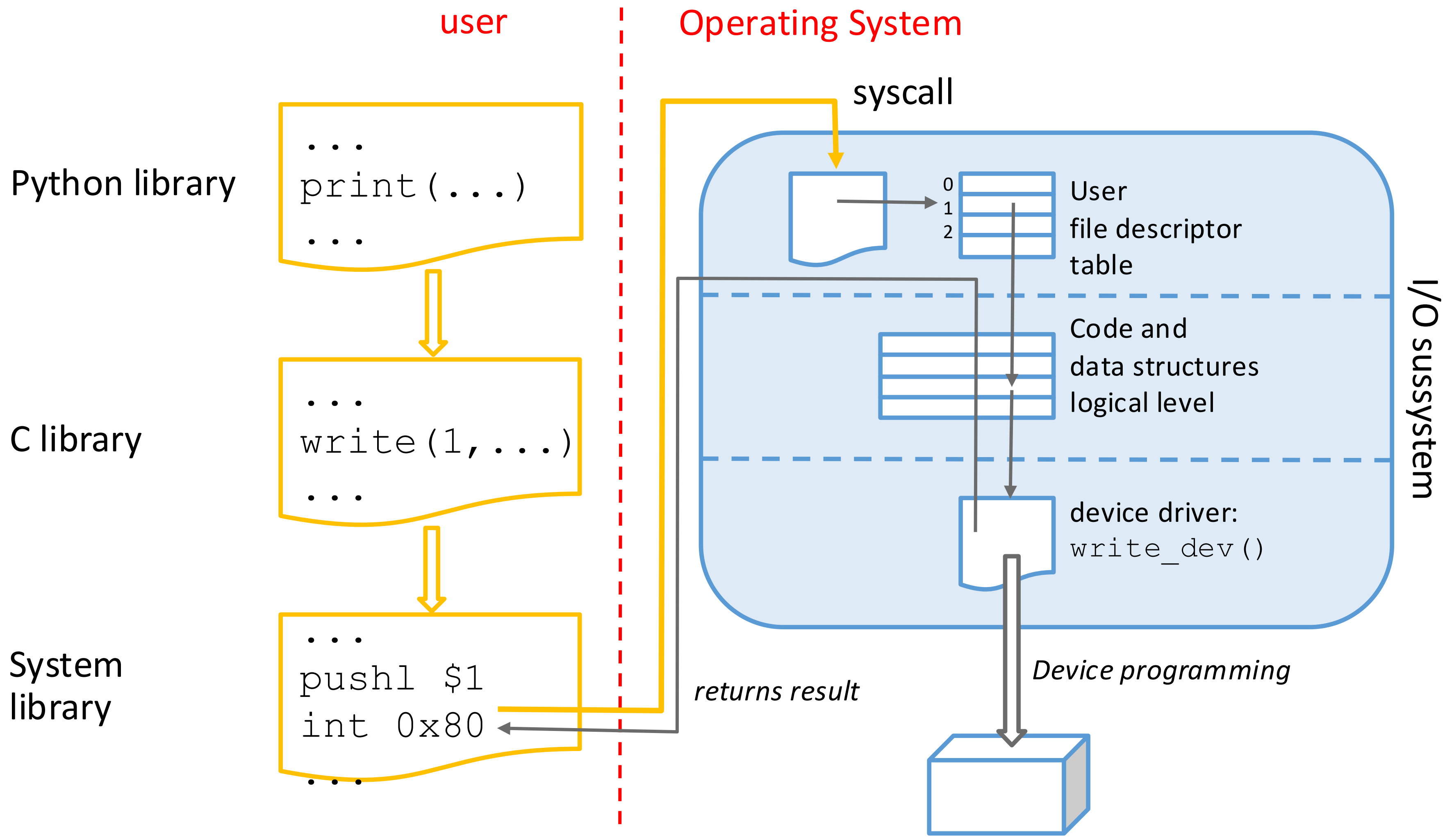
File descriptor returned by open

Address of a data structure in the user process

Num. of bytes the user wants to read

- `n = write(fd, buffer, count);`
  - Asks for writing *count* bytes (characters) to the device pointed by *fd*
    - If there is space on device for *count* bytes, it writes *count* (the kernel allocates a new block if the file does not contains a block that corresponds to the byte offset to be written)
    - If there is less, it writes what fits
    - If there is no space left on device, it's up to the device behaviour:
      - Blocking process until space available
      - Returns 0 immediately
  - Returns n, the number of bytes actually written
  - The kernel updates the offset in the file table to the n; consequently, successive writes of a file update the file data in sequence (when the write is complete, the kernel updates the file size entry in the inode if the file has grown larger)

Example: writing to a device



Dup/dup2/close

- `newfd = dup(fd);`
  - Where *fd* is the file descriptor being duped and *newfd* is the new file descriptor that references the file.
  - Copies a file descriptor into the first free slot of the user file descriptor table.
  - Returns *newfd*
- `newfd = dup2(fd, newfd);`
  - Similar to *dup*, but the free slot is forced to be *newfd*
  - If *newfd* already refers to an open file, it is closed before duped
- `close(fd);`
  - Where *fd* is the file descriptor for the *open* file.
  - The kernel does the *close* operation by manipulating the file descriptor and the corresponding file table and inode table.
  - If the reference count of the file table entry is greater than 1 (*dup, fork*) then the kernel decrements the *count* and the *close* completes.
  - If the table reference count is 1, the kernels frees the entry and releases the in-core inode (If other processes still reference the inode, the kernel decrements the inode reference count but leaves it allocated).

pipe

- Pipes allow transfer of data between processes in a first-in-first-out manner and they allow also synchronization of process execution.
  - `pipe(fd_vector); // Device for FIFO communications`
    - Creates an unnamed pipe. Returns 2 file descriptors `fd_vector[0]` for reading, `fd_vector[1]` for writing the pipe (and allocates corresponding File Table entries).
    - There is **no name** in the VFS, so there is no any call to `open`.
    - Only related processes, descendants of a processes that issued the pipe call can share access to unnamed pipes
  - Named pipes are identical, except for the way that a process initially accesses them
  - `mknod("my_pipe", S_IFIFO | 0600, 0);`
    - Creates a pipe, named "my_pipe", in the VFS and, hence, processes that are not closely related can communicate.
    - Processes use the `open` syscall for named pipes in the same way that they open regular files.
    - The kernel allocates 2 entries in the File Table and 1 in the Inode Table.

pipe

- Usage
  - Processes use the `open` system call for named pipes, but the `pipe` system call to create unnamed pipes.
  - Afterwards processes use regular system calls for files, such as `read` and `write`, and `close` when manipulating pipes.
  - Pipes are bidirectional, but ideally each process uses it in just one direction. In this case the kernel manages synchronization of process execution.
- Blocking device:
  - Opening: a process that opens the named pipe for reading will sleep until another process opens the named pipe for writing, and vice versa.
  - Reading: if the pipe is empty, the process will typically sleep until another process writes data into the pipe.
  - If the count of writer processes drops to 0 and there are processes asleep waiting to read from the pipe, the kernel awakens them, and they return from their read calls without reading any data.
  - Writing: if a process writes a pipe and the pipe cannot hold all the data, the kernel marks the inode and goes to sleep waiting for data to drain from the pipe.
    - If there are no processes reading from the pipe, the processes that writes the pipe receives a signal SIGPIPE → the kernel awakens the sleeping processes
  - Processes should close all non-used files descriptors, otherwise -> Blocking!
- Data structures
  - 2 entries in the user File Descriptor Table (R/W)
  - 2 entries in the File Table (R/W)
  - 1 entry in the in-core Inode Table

lseek

- `lseek` changes the File Table byte offset (the read-write pointer) . It allows direct access by position in data files (or even sequential devices, like tapes).
    - Offset is 0 after an `open` system call (except with `APPEND` flag).
    - Offset is increased by `read` and `write` system calls.
    - Offset can be modified by the user with lseek system call
- `new = lseek(fildes, offset, origin)`
- The value of the pointer depends on `origin`:
    - `SEEK_SET`: pointer = `offset`. Set the pointer to offset bytes from the beginning of the file.
    - `SEEK_CUR`: pointer += `offset`. Increment the current value of the pointer by offset.
    - `SEEK_END`: pointer = file_size + `offset`. Set the pointer to the size of the file plus offset bytes.
    - `offset` can be negative.

examples

Byte-by-Byte access

- Reading from the standard input and writing to the standard output

```
while ((n = read(0, &c, 1)) > 0)
    write(1, &c, 1);
```

- Note:
    - Reading while there are data (`n==0`), that's up to the device. The total amount of syscall depends on the number of bytes to be read
    - Processes conventionally have access to three files: its standard input (0), its standard output (1) and its standard error(2).
    - Processes executing at a terminal typically use the terminal for these three files.
    - But each may be "redirected" independently to any logical device that accepts the operations of reading and/or writing.
    - For instance:

```
#example1 →                   input=terminal, output=terminal
#example1 <disp1 →            input=disp1, output=terminal
#example1 <disp1  >disp2 →    input=disp1, output=disp2
```

Buffer in user space access

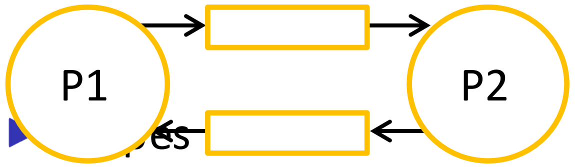
- The same, but reading blocks if bytes (chars in this case)

```
char buf[SIZE];
...
while ((n = read(0, buf, SIZE)) > 0)
    write(1, buf, n);
```

- Note:
  - You must write `n` bytes
    - Process is asking for `SIZE` bytes, however it reads `n` bytes
  - What about performance? How many system calls are executed?

Data communication using pipes

- Program a process schema equivalent to the figure:



- P1 sends to pipe1 and receives from pipe2
- P2 the opposite symmetrically

```
void p1(int fdin,int fdout);
void p2(int fdin,int fdout);
```

```
int pipe1[2], pipe2[2],pidp1,pidp2;
pipe(pipe1);
pipe(pipe2);
pidp1=fork();
if (pidp1==0){
      close(pipe1[0]);
      close(pipe2[1]);
      p1(pipe2[0],pipe1[1]);
      exit(0);
}
close(pipe1[1]);
close(pipe2[0]);
pidp2=fork();
if (pidp2==0){
      p2(pipe1[0],pipe2[1]);
      exit(0);
}
close(pipe1[0]);close(pipe2[1]);
while(waitpid(-1,null,0)>0);
```

Random access and size evaluation

- What does this code do?

```
fd = open(“abc.txt”, O_RDONLY);
while (read(fd, &c, 1) > 0) {
    write(1, &c, 1);
    lseek(fd, 4, SEEK_CUR);
}
```

- And this one?

```
fd = open(“abc.txt”, O_RDONLY);
size = lseek(fd, 0, SEEK_END);
printf(“%d\n”, size);
```

pipes and blocking

```
int fd[2];
...
pipe(fd);
pid = fork();
if (pid == 0) {                                   // child
    while (read(0, &c, 1) > 0) {
        // Reads, process and send data
        write(fd[1], &c, 1);
    }
}
else {                                            // parent
    while (read(fd[0], &c, 1) > 0) {
        // Receives, process and send data
        write(1, &c, 1);
    }
}
...
```

- Be careful The parent process must close `fd[1]` if it does not want to be blocked!

Sharing the read-writer pointer

- What does this code do?

```
...
fd = open(“fitxer.txt”, O_RDONLY);
pid = fork();
while ((n = read(fd, &car, 1)) > 0 )
       if (car == ‘A’) numA++;
sprintf(str, “El número d’As és %d\n”, numA);
write(1, str, strlen(str));
...
```

Non shared read-write pointer

- What does this code do?

```
...
pid = fork();
fd = open("fitxer.txt", O_RDONLY);
while ((n = read(fd, &car, 1)) > 0 )
       if (car == 'A') numA++;
sprintf(str, "El número d'As és %d\n", numA);
write(1, str, strlen(str));
...
```

Redirection of standard input and output

- What does this code do?

```
...
pid = fork();
if ( pid == 0 ) {
     close(0);
     fd1 = open("/dev/disp1", O_RDONLY);
     close(1);
     fd2 = open("/dev/disp2", O_WRONLY);
     execv("programa", "programa", (char *)NULL);
}
...
```

Redirection and pipes

```
...
pipe(fd);
pid1 = fork();
if ( pid1 != 0 ) {                                  // parent
    pid2 = fork();
    if ( pid2 != 0 ) {                              // parent
        close(fd[0]); close(fd[1]);
        while (1);
    }
    else {                                          // child 2
        close(0); dup(fd[0]);
        close(fd[0]); close(fd[1]);
        execlp("programa2", "programa2", NULL);
    }
}
else {                                              // child 1
        close(1); dup(fd[1]);
        close(fd[0]); close(fd[1]);
        execlp("programa1", "programa1", NULL);
}
```

Kernel data structures

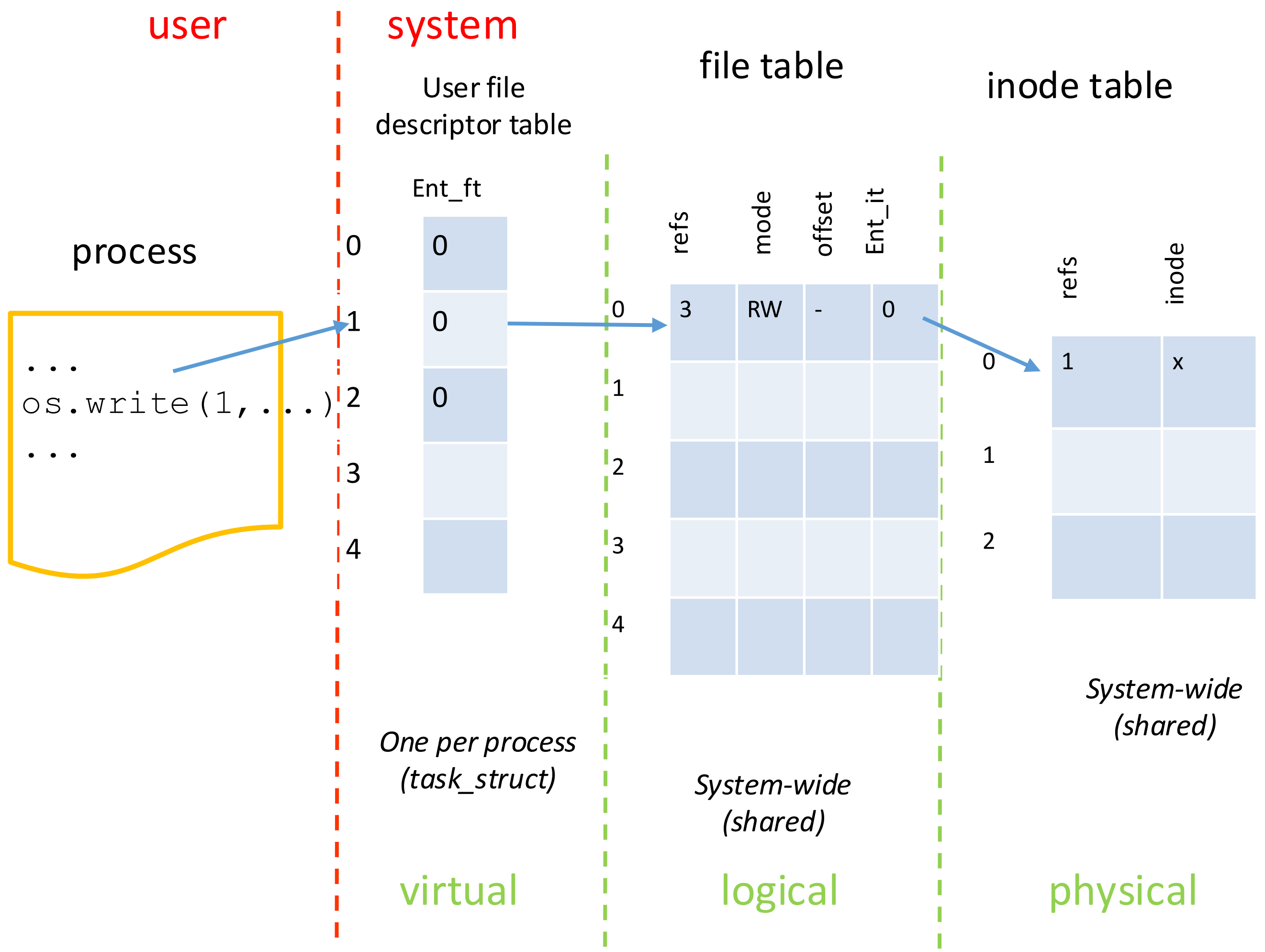
Kernel data structures: inode

- Data structure for storing file system metadata with pointers to its data. Each inode represents an individual file. It stores:
  - size
  - type
  - access permissions
  - owner and group
  - file access times
  - number of links (number of file names pointing to the inode)
  - pointers to data (multilevel indexation) → see below, at the end of this section
- All information about a file, except file names
- Stored on disk, but there is an in-core copy for access optimization

Kernel data structures

- Each process
  - User Field Descriptor Table (FDT): per-process open-file table (saved in the task_struct, ie, PCB)
    - Records to which files the process is accessing
    - The file is accessed through the file descriptor, which is an index to the **FT**
    - Each file descriptor is a virtual device
    - Each field descriptor points to an entry in the Open File Table (FT)
    - Fields we'll assume: `num_entry_OFT`
- Global:
  - Open File Table (FT):
    - System-wide open-file management
    - One entry can be shared among several processes and one process can point to several entries.
    - One entry of FT points to one entry of the Inode Table (IT)
    - Fields we'll assume: `num_links, mode , offset, num_it_entry`
  - Inode Table (IT):
    - Active-inode table. One entry for each opened physical object. Including DD routines.
    - Memory (in-core) copy of the disk data for optimization purposes,
    - Fields we'll assume: `num_links, inode_data`
  - Buffer Cache
    - Memory zone to hold any I-node and data block transfer from/to the disk
    - If the requested I-node or block is in the cache, the access to the disk is not performed

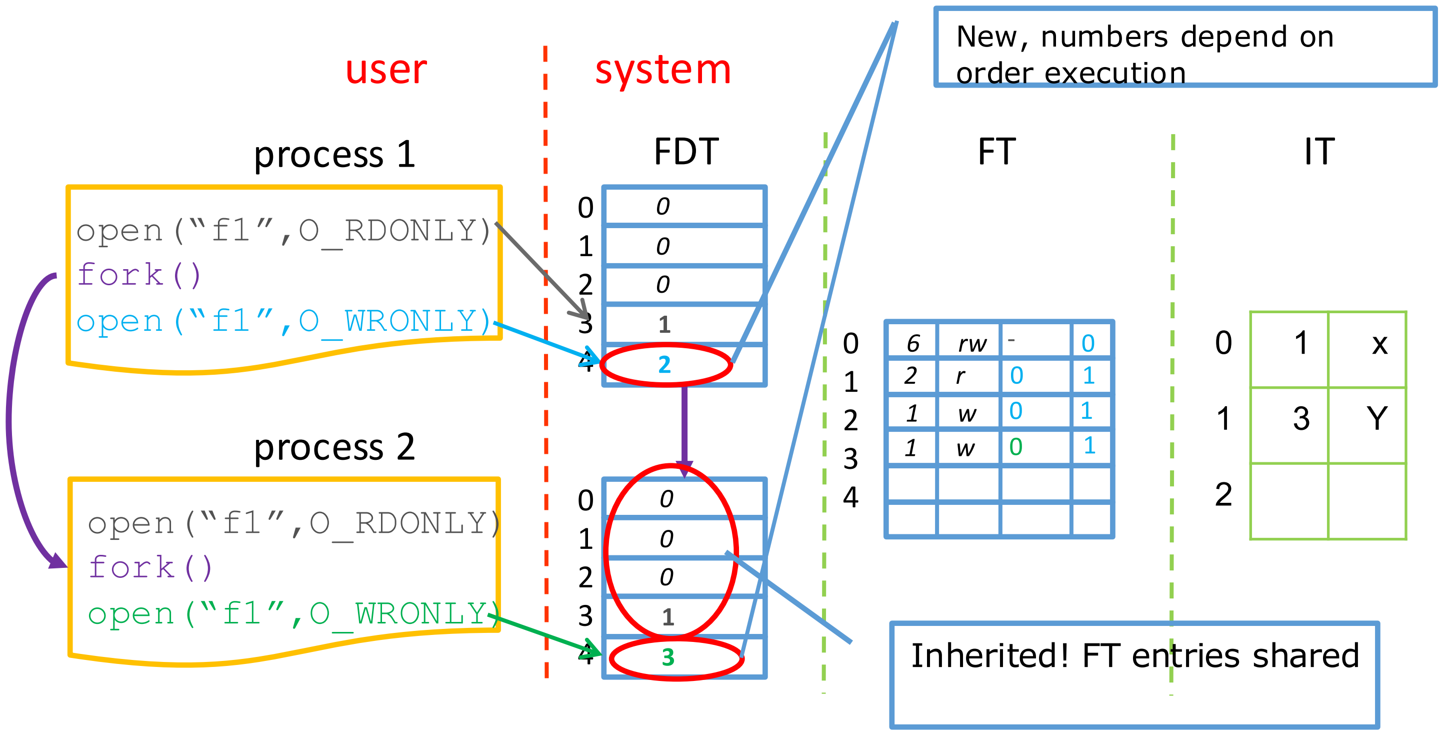
Kernel data structures



```
...
os.write(1,...)
...
```

I/O and concurrent execution (1)

- I/O and *fork*
  - Child process inherits a **copy** of the parent process file descriptor table.
    - All open entries point to the same File Table entries
  - Parent and child sharing devices opened before *fork* system call
  - Next calls to open will be independent



user
system
process 1
process 2
FDT
FT
IT
New, numbers depend on order execution
Inherited! FT entries shared

process 1:

```
open("f1",O_RDONLY)
fork()
open("f1",O_WRONLY)
```

process 2:

```
open("f1",O_RDONLY)
fork()
open("f1",O_WRONLY)
```

FDT (process 1):

0	0
1	0
2	0
3	1
4	2

FDT (process 2):

0	0
1	0
2	0
3	1
4	3

FT:

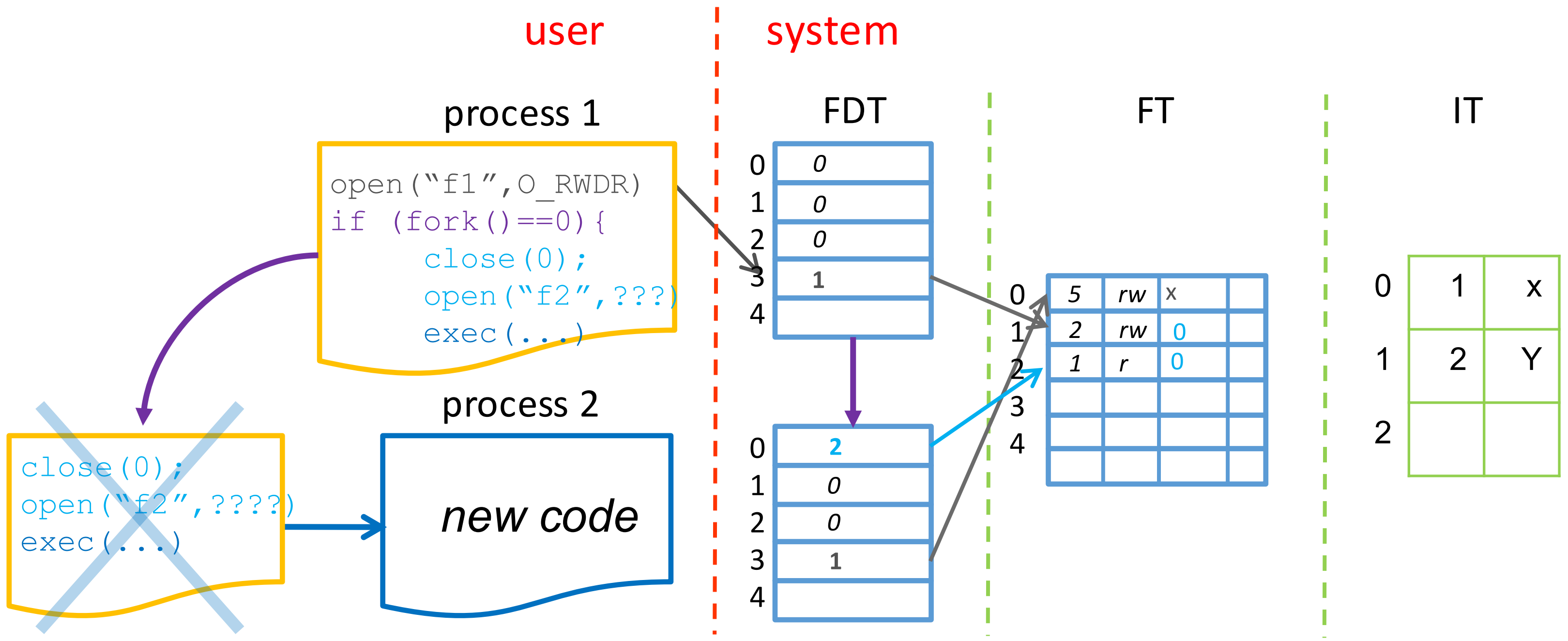
0	6	rw	-	0
1	2	r	0	1
2	1	w	0	1
3	1	w	0	1
4				

IT:

0	1	x
1	3	Y
2		

I/O and concurrent execution (2)

- I/O and *exec*
  - New process image **keeps** the same process' I/O internal structures
  - *fork+exec* allows I/O redirection before process image change



```
open("f1",O_RWDR)
if (fork()==0){
      close(0);
      open("f2",???)
      exec(...)
```

```
close(0);
open("f2",????)
exec(...)
```

	FDT (process 1)
0	0
1	0
2	0
3	1
4	

	FDT (process 2)
0	2
1	0
2	0
3	1
4	

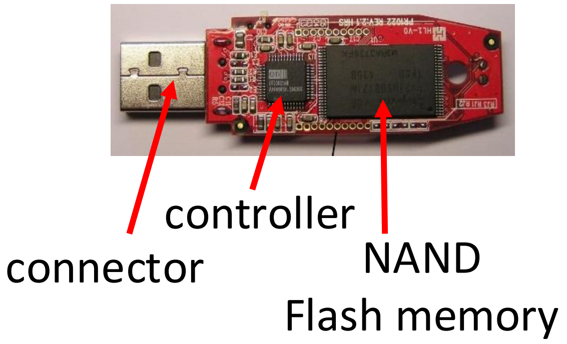
	FT			
0	5	rw	x	
1	2	rw	0	
2	1	r	0	
3				
4				

	IT	
0	1	x
1	2	Y
2		

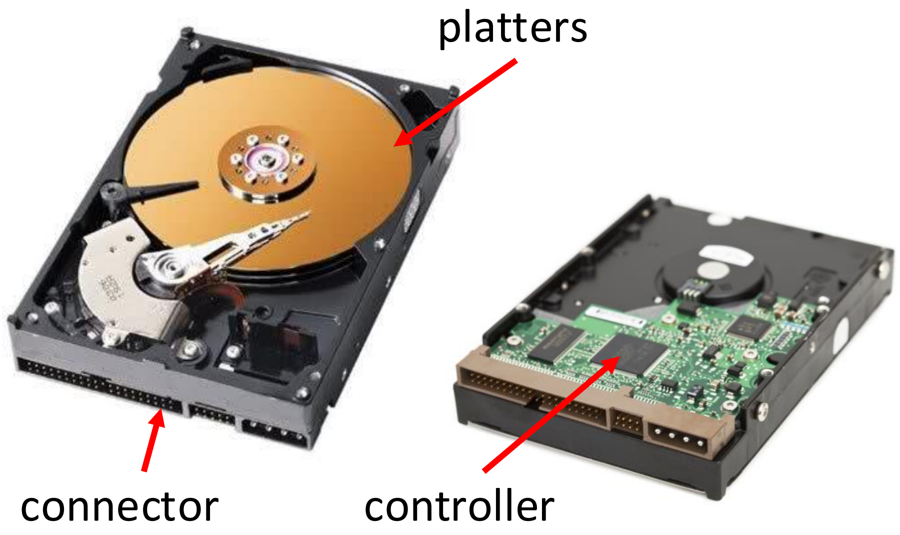
File system

Physical Storage Devices

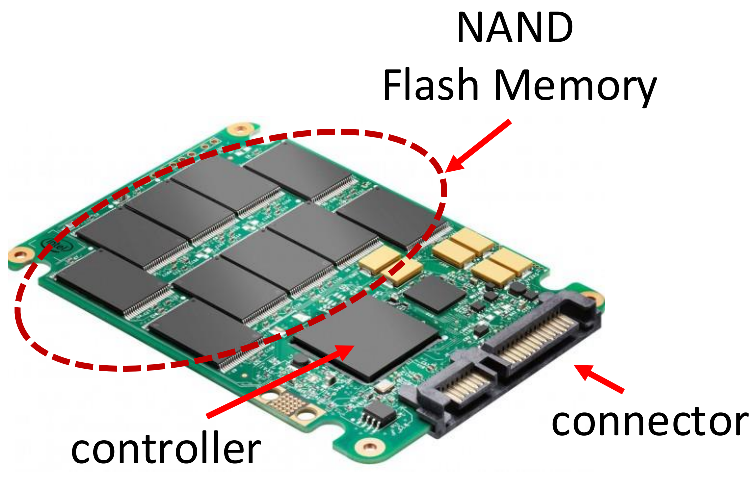
- Non-volatile memory to save data
- Similar vs Different components
  - Impact on performance and capacity



USB Drive



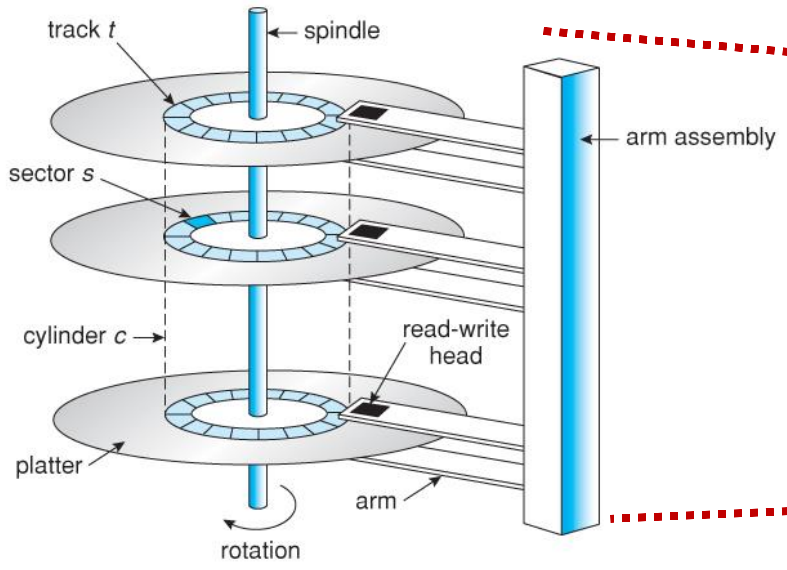
Hard Disk



Solid State Drive (SSD)

How are data physically saved?

- Any storage device needs to organize the pool of memory
  - E.g.: DVD, hard-disk, pen-drive, etc.
- Sector: The smallest unit of data that can be read/written
  - **Defined by the hardware**
  - Fixed size (tipically 512 Bytes)

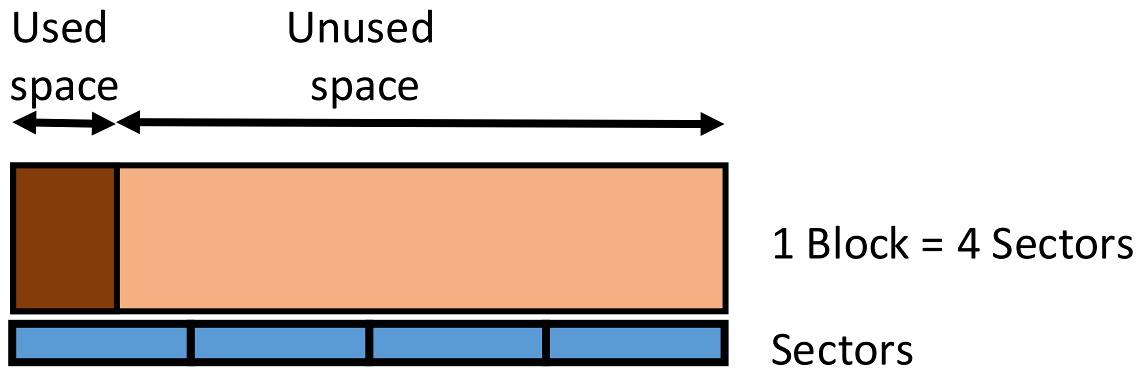


Some parameters that impact on performance

Speed: rpm
Connector Bandiwth: Gbps

How is a storage device organized?

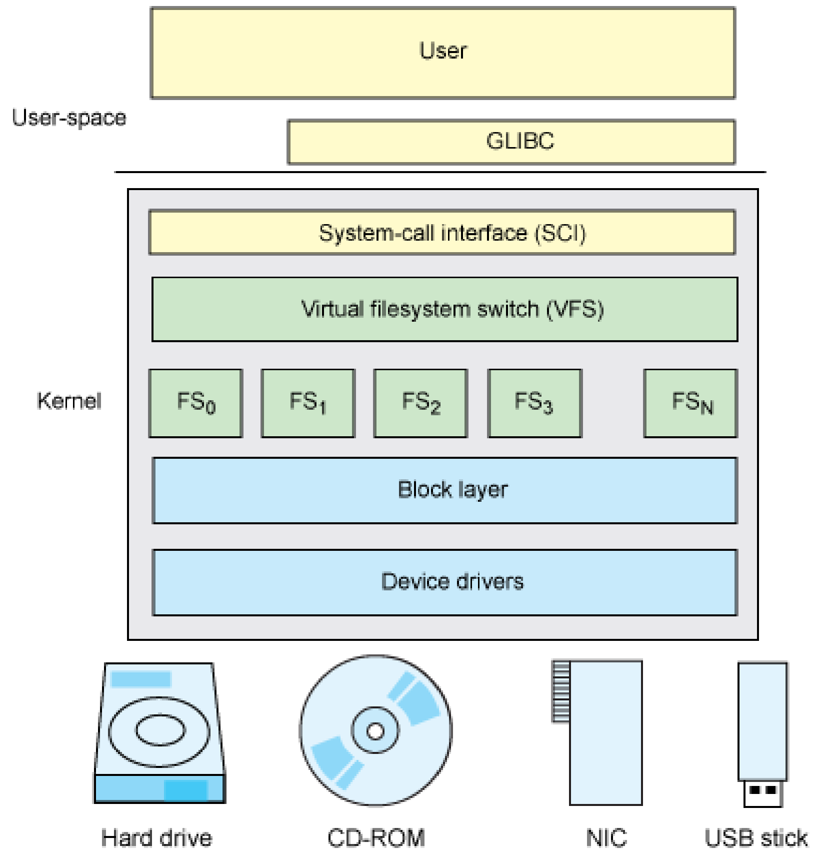
- Block: A group of sectors (the smallest unit to allocate space)
  - **Defined by the OS (when formatting the device)**
- But, what is the best block size????
  - If it is likely to use large files…
    - Large blocks
  - If it is likely to use short files…
    - Short blocks



- What is the impact of a bad block size selection???
  - Too large: fragmentation (waste of space)
  - Too short: degrade performance too many accesses to the device

Virtual File System (VFS)

- An abstraction layer to manage different types of file system
  - It provides a single system call interface for any type of file system
  - VFS for UNIX/Linux. Other Oses use similar approaches
- Types of File System
  - FAT (File Allocation Table)
    - Removable drives
  - exFAT (Extended FAT)
    - Removable drives larger tan 4GB
  - NTFS (New Technology Transfer)
    - Windows
  - **I-node based file systems (UNIX/Linux)**
    - Ext3, ext4, Reiser4, XFS, F2FS
  - Cloud File System
    - GlusterFS, Ceph, HadoopFS, ElasticFileSystem (Amazon)



https://en.wikipedia.org/wiki/Comparison_of_file_systems

File Systems

- Swap space: in UNIX/Linux OS is a special file system that extends main memory
  - Windows implements swap space in a single resizable file
- FUSE: Filesystem in Userspace
  - Let’s non-priviledge users implement their own file system without modifying the kernel (it is executed in user space rather tan kernel space)
    - E.g.: GDFS (Google Drive), WikipediaFS, propietary File Systems
- Different File Systems offer different features that impact on performance, reliability, resilience, security, etc

Journaling

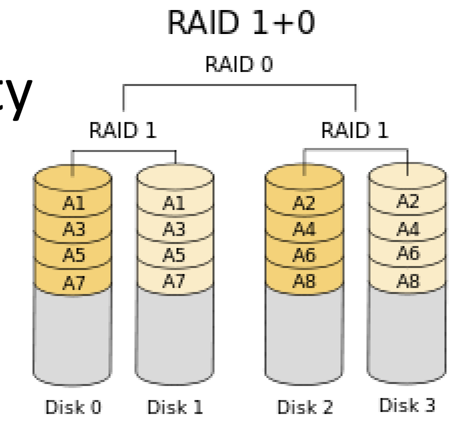
- Transaction based File System
  - Keep track of changes not yet committed to the file system
  - It records the changes in a “journal” file
    - It has a dedicated area in the file system

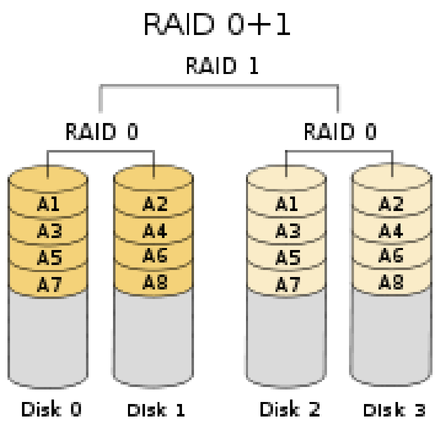
- In case of system failure or outage…
  - The file system can be brought back fast and with lower likelihood of errors
    - E.g.: After a crash, replay the last updates from the “journal”

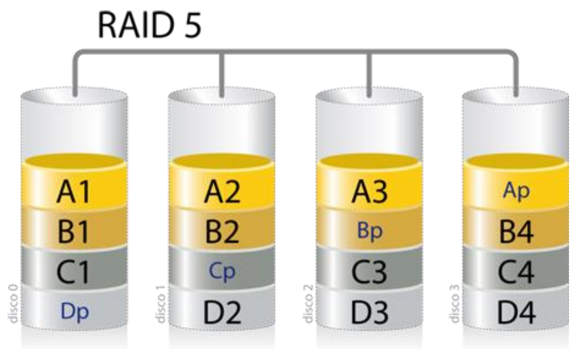
- Some File Systems that implement Journaling
  - Ext3, ext4, ReiserFS, XFS, JFS

RAID: Redundant Array of Independent Disks

- ►Storage virtualization technology that combines multiple physical storage drives into a single logical unit
  - ►Software driver vs hardware controller
  - ►Impact on performance and effective capacity
- ►Several approaches (can be combined)
  - ►RAID 0: Stripping → distributed data
  - ►RAID 1: Mirroring → replicated data
    - ►RAID 1+0 vs RAID 0+1
  - ►RAID 5: with distributed parity blocks







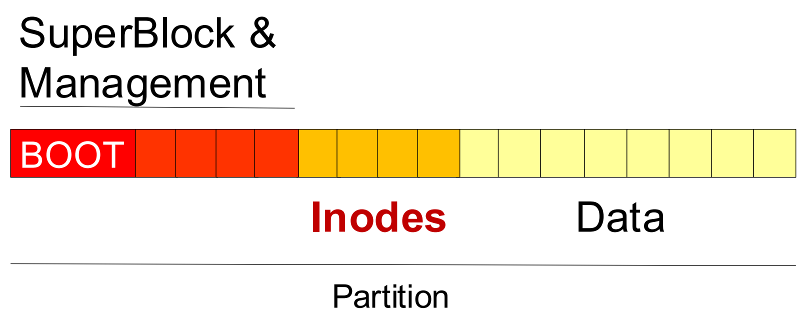
Performance Impact: handling blocks

- Every file system has its own mechanism to handle…
  - Occupied/free blocks
  - The blocks of a given file (i.e. how to access the contents of a given file)

- Depending on the file system, the implementa
  - For example:
    - FAT: has a global table with as many entries as blocks has the drive
      - Linked-list based file access
    - I-node based: has a structure called I-node to hold all the information to manage a file
      - Index based file access (a.k.a. multi-level index). The index is hold by every I-node

I-Node Based File System

- Some fields of the I-Node:
  - I-Node ID
  - Size
  - Type of file (regular file, directory, named pipe, socket, etc.)
  - Protection (Read / Write / Execute (RWX)   for   Owner, Group, Others)
  - Ownership
  - Timestamps
  - Number of Links (# direct relations between a symbolic name and I-Node ID)
  - Pointers to blocks of data (multi-level index). It use to has 12-13 pointers.



Directories: Organizing files

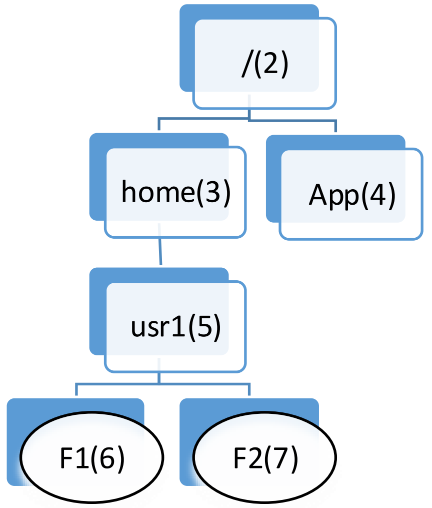
- **Directory:** Logical structure to organize files
  - Pathname: Relative (from any folder) vs Absolute (from the root folder)
- It is a particular type of file managed by the OS
  - "/" is the root folder
    - Every partition has its own root folder **(I-node ID 2)**
  - "." and ".." are mandatory entries in any directory
    - Even though the directory has no files
- **Hardlink:** A direct relation between name and I-node ID

Name	I-node
.	2
..	2
home	3
App	4

Directory: /

Name	I-node
.	5
..	3
F1	6
F2	7

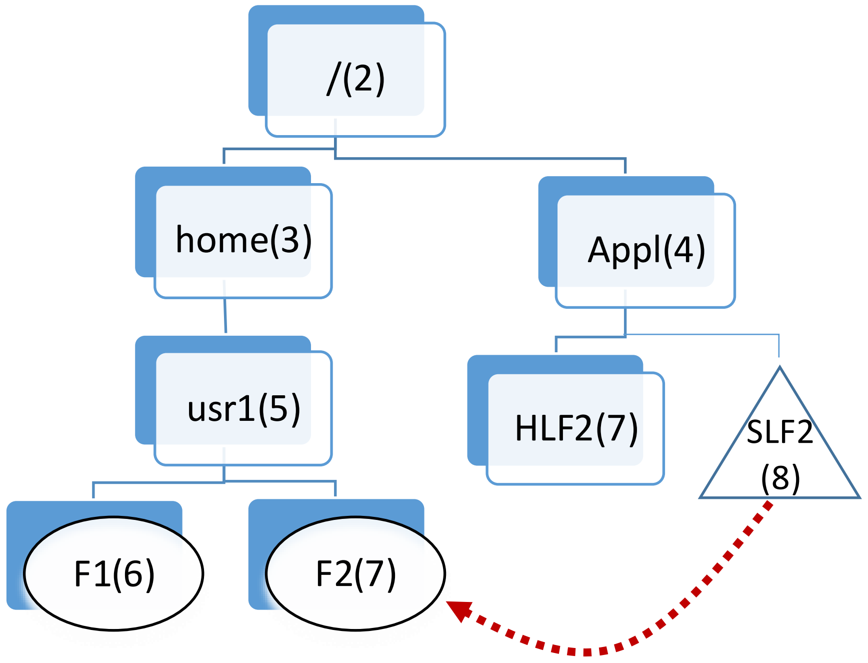
Directory: /home/usr1



Directories: Organizing files

- Directories are organized as graphs
  - A given file can be accessed from different directories

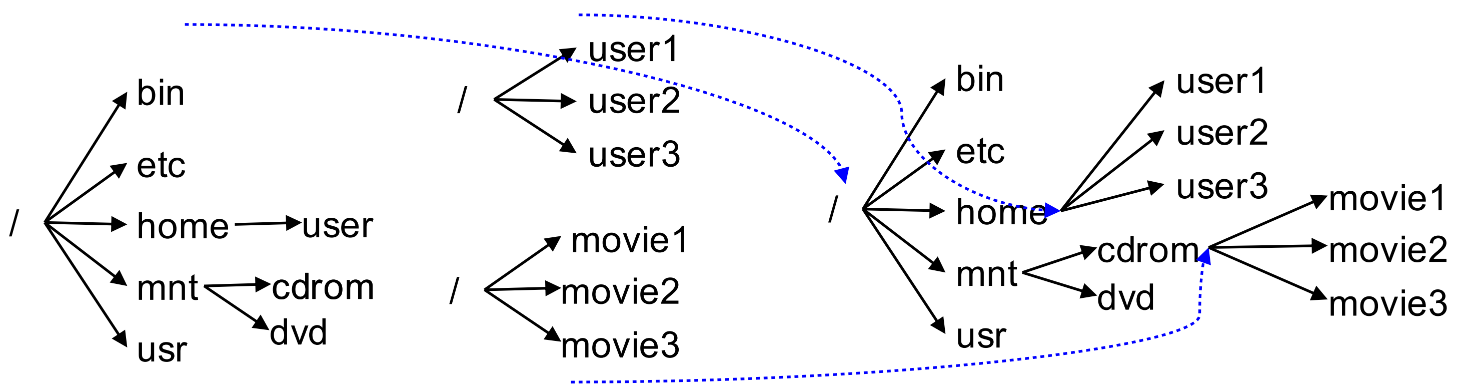
- Sharing files
  - Hardlinks
    - It only needs a new entry in a directory (name→I-node ID)
  - **Softlinks**
    - A **new file** that comprises a pathname
      - Similar to shortcuts in Windows
  - Pros/Cons/restrictions lead to use one or the other



Mount

- Publishing the contents of a disk partition on the file system
  - Linux command line:

```
$ mount -t ext4 /dev/hda1 /home   #mounting the home partition
$ unmount /dev/hda1               #unmounting the home partition
```



Bibliography

- Randal E. Bryant and David R. O'Hallaron,
  - Computer Systems: A Programmer's Perspective, Third Edition (CS:APP3e), Ch. 10.
  - https://discovery.upc.edu/permalink/34CSUC_UPC/1q393em/alma991004062589706711
- Operating System
  - Silberschatz, A; Galvin, P. B; Gagne, G. 2019. Chapters (11-15)
  - https://discovery.upc.edu/permalink/34CSUC_UPC/rdgucl/alma991004148389706711
- Python documentation
  - https://docs.python.org/3/library/os.html
  - https://docs.python.org/3/library/sys.html
  - https://docs.python.org/3/library/signal.html
  - https://docs.python.org/3/tutorial/errors.html