ESCI upf. School of International Studies

Input / Output management

THEORY OF INFORMATION, ARCHITECTURE OF COMPUTERS AND OPERATING SYSTEMS

Bioinformatics

Course 2022/23 T3

UPC Departament d'Arquitectura de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA

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)

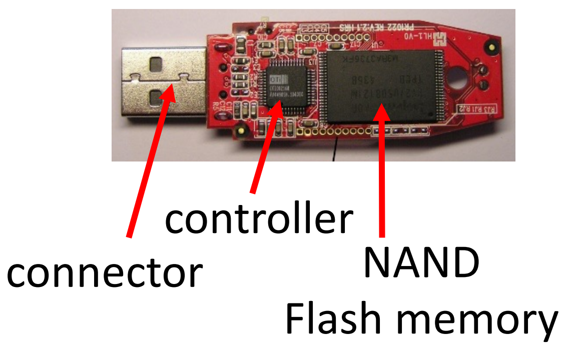
- In fact, basically, processes perform computation and/or I/O
- Sometimes, even, I/O is the main task of the process:
  for instance, web browsing, shell, word processor

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

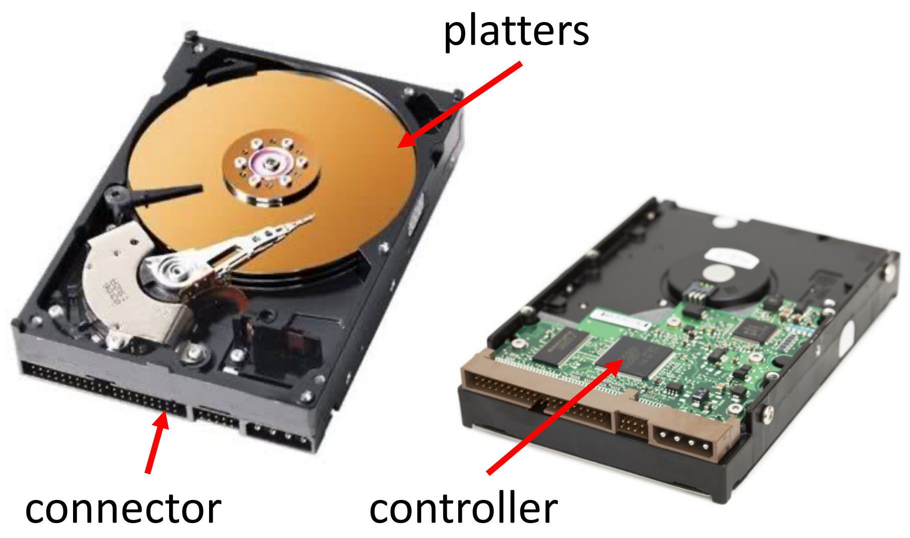
I/O Devices

Physical Storage Devices

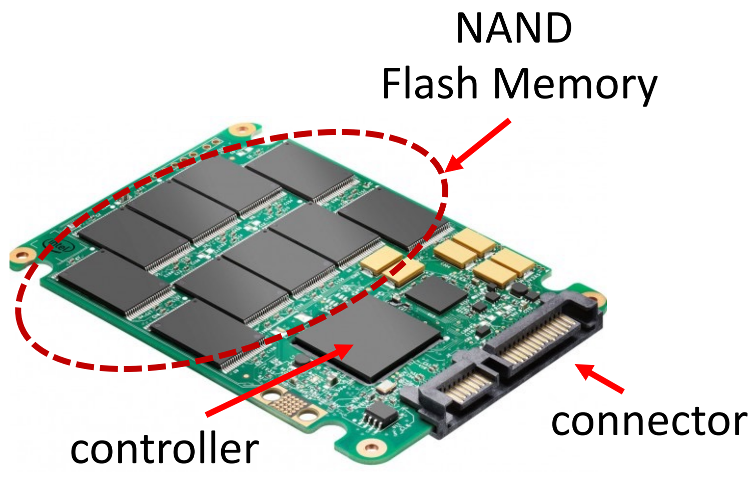
- Non-volatile memory to save data
- Similar vs Different components
  - Impact on performance and capacity



USB Drive

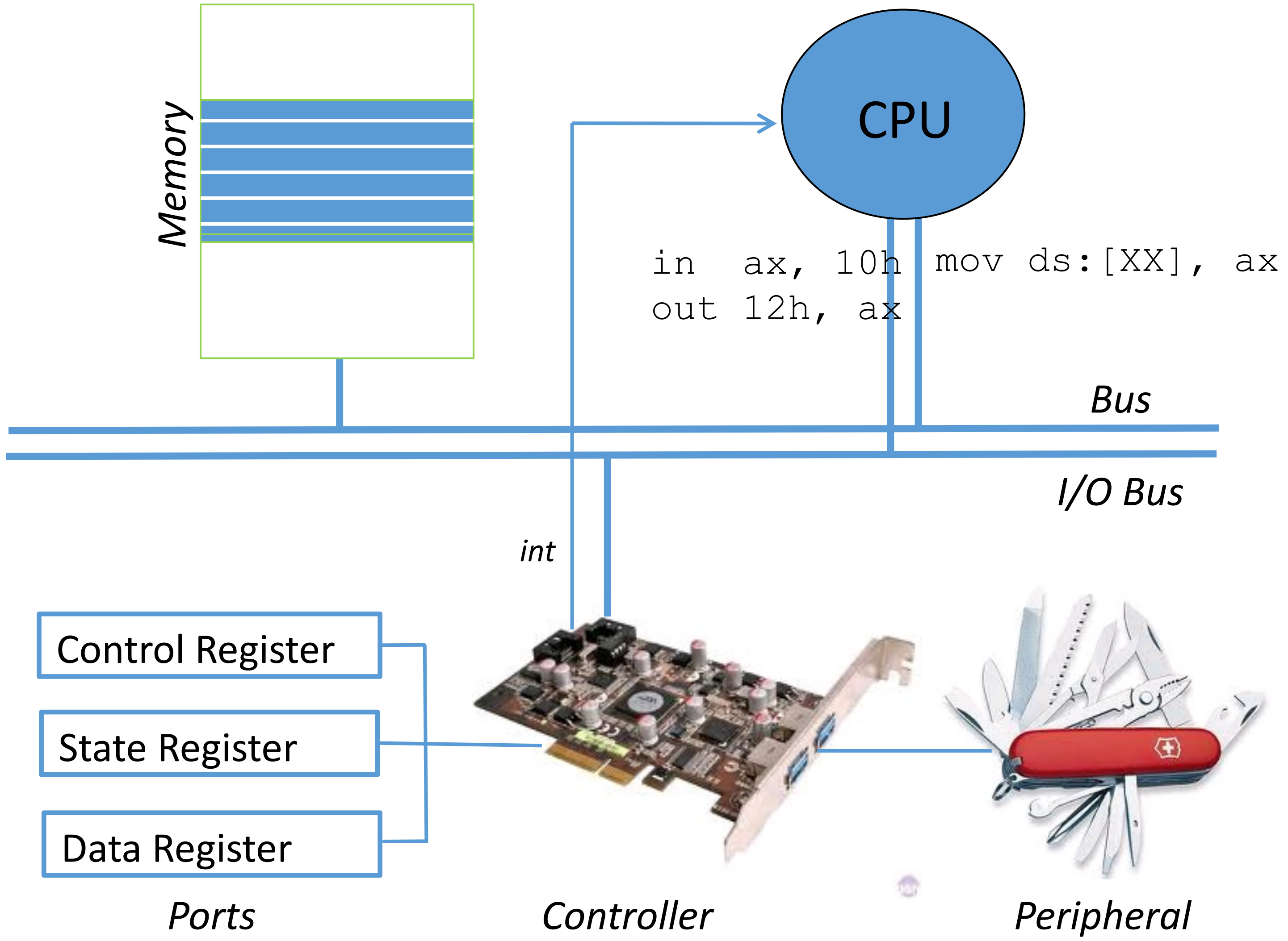


Hard Disk



Solid State Drive (SSD)

HW view : Accessing physical devices

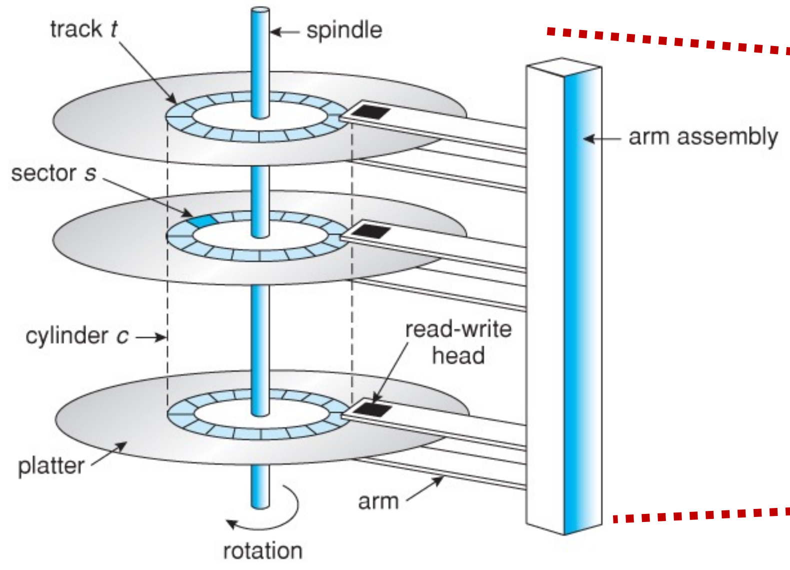


How are data physically saved?

- 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



Devices: virtual, logical, PHYSICAL

Device types

- To interact with users: display, keyboard, mouse. To store data: hard disk, Bluray, pendrive. To transfer data: modem, network, WI-FI or even more specialized (plane controllers, sensors, robots) … many possible characterizations

- Classification criteria:
  - Device type: logical, physical, network
  - Access speed: keyboard vs hard disk
  - Access flow: mouse (byte) vs DVD (block)
  - Access exclusivity : disk (shared) vs printer (dedicated)
  - And so on …

CONCEPT:
Device independence

- Trade-off: standardization vs new device types

Independence: principles of design

- ►The goal is that processes (code mainly) be independent of the device that is being accessed
- ►Uniform I/O operations
  - ►Access to any device with the **same system calls**
  - ►Increase portability and simplicity of user’s processes
- ►Use of **virtual devices**
  - ►Process does not specifies the physical device, but it uses an identifier and a later translation.
- ►**Device redirection**: use different devices with no code changes
  - ►The Operating System allows a process to change the allocation of its virtual devices

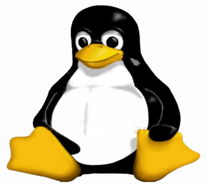
```
% programa < disp1 > disp2
```

Independence: design principles

- ►Hence, usually, design in three levels: **virtual**, **logical** and **physical**
- ►First level gives us independence, the **process works on virtual devices** and does not need to know what's behind.
- ►The second level gives us **device sharing**. Different concurrent accesses to the same device.
- ►Therefore, it's possible to write programs performing I/O on (virtual) devices without specify which (logical) ones
- ►In execution time, process dynamically determines on which devices it is working. It can be a program argument or "inherited" from its parent.
- ►Third level separates operations (software) from implementation. This code is quite low-level, most of times in assembler.

Virtual Device

- **Virtual level**: isolates user from the complexity of managing physical devices.
  - It sets correspondence between symbolic name (filename) and user application, using a virtual device.
    - A symbolic name is the representation inside of the Operating System
      - `/dev/dispX` or `.../dispX`
    - A virtual device represents a device in use by a process
      - Virtual Device = channel = file descriptor. It is a **number**
      - Processes have 3 standard file descriptors
        - Standard Input file descriptor 0 (stdin)
        - Standard Output file descriptor 1 (stdout)
        - Standard Error descriptor 2 (stderr)



  - I/O System calls use the identifier of the virtual device

Logical Device

- **Logical level**:
  - It sets correspondence between virtual device and physical(?) device
  - It manages devices with or without physical representation
    - For instance, a virtual disk (on memory) or a null device
  - It deals with independent size data blocks
  - It brings a uniform interface to physical level
  - It offers shared access (concurrent) to physical devices that represents (if so)
  - In this level permissions, such as access rights, are checked.
  - Linux identifies a logical device with a file name

Physical Device

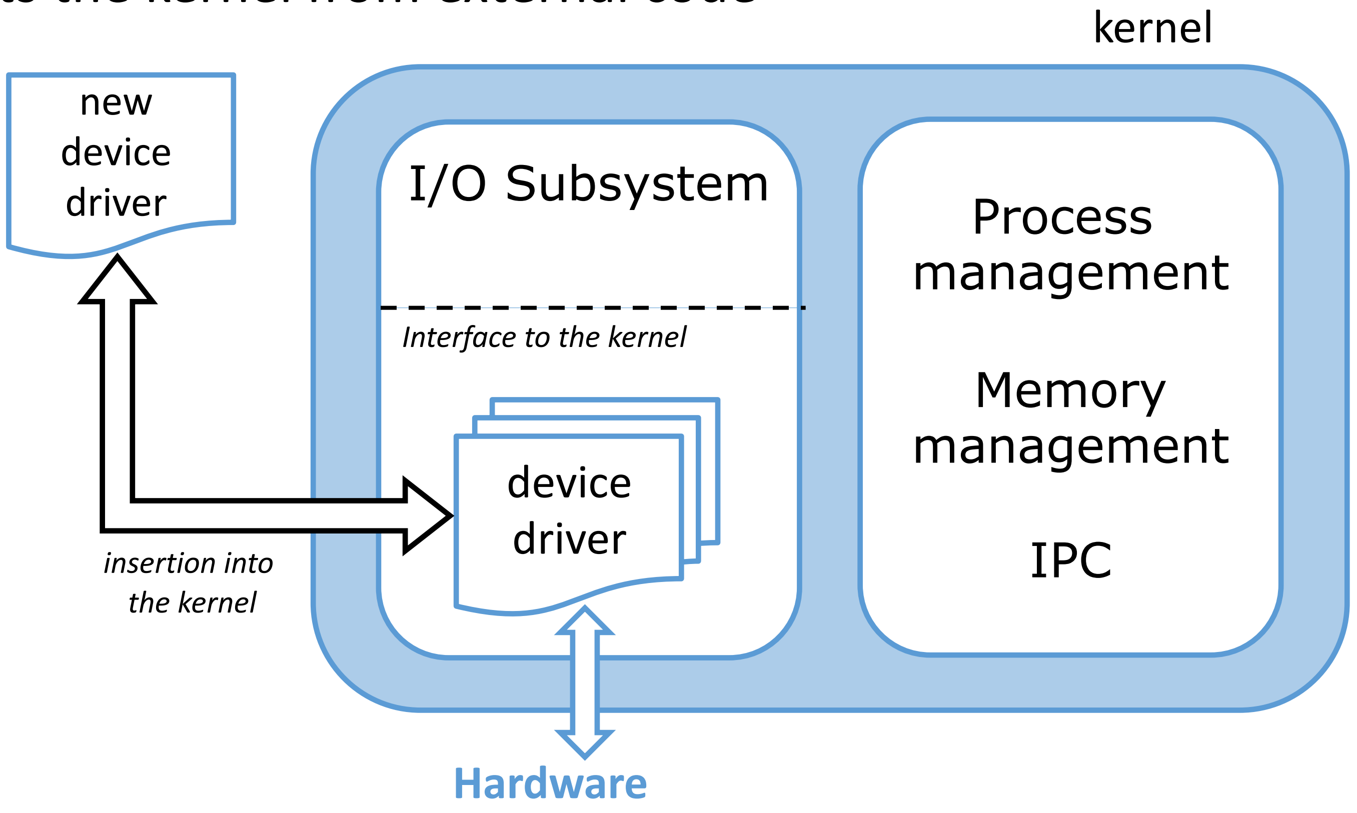
- **Physical Level**: implements logical level operations in low-level.
  - Translates parameters from the logical level to specific parameters
    - For instance, on a hard disk, translates file offset to cylinder, platter, sector and track
  - Device initialization. Check if it is free, otherwise it enqueues a request
  - It performs the request programming operation
    - It could mean state checking, hard disk engine init, ...
  - Waits (or not) the operation ending
  - If successful, return the results or report any possible error
  - In Linux, a physical device is identified by three parameters:
    - Type: **Block/Character**
    - And two numbers: major/minor
      - **Major**: tells the kernel which family device to use (DD)
      - **Minor**: tells the kernel which one inside the family

Device Drivers

- In order to offer independence, a set of operations is defined for all devices
  - It is a superset of operations that could be offered to access to a physical device.
  - Not all devices can offer all operations
  - In translation time (from virtual to logical) the available operations is set
- OS programmers can’t generate code for all devices, models, etc.
- Manufacturers should provide the low-level routines set that implements device functionality
    - The code + data to access device is known as Device Driver
    - It follows the interface specification of accessing to I/O operations defined by the OS
- To add a new device
  1. Option 1: with kernel recompilation
  2. Option 2: without kernel recompilation
     - OS must offer a mechanism to add kernel code/data dynamically
       - *Dynamic Kernel Module Support*, *Plug & Play*

Device Driver

- Common operations (interface) are identified and specific differences are encapsulated inside OS modules: Device Driver.
  - Isolates the kernel from device management complexity
  - Protects the kernel from external code



I/O Linux Modules: +details

- Steps to create and start using a new device:

Phy.

- **Compile** DD, if needed, in kernel object format (.ko)
  - Type (block/character) and IDs are especified inside the code
- **Install** (insert) at runtime driver routines
  - insmod file_driver
  - The driver is related to a given major id

Log.

- **Create a logic device** (filename) and link it to the physical device
  - New file ←→block | character + major & minor
  - **Command** mknod
    - mknod /dev/mydisp c major minor

Virt.

- **Create the virtual device**
    - open(“/dev/mydisp”, ...);

Kernel data structures

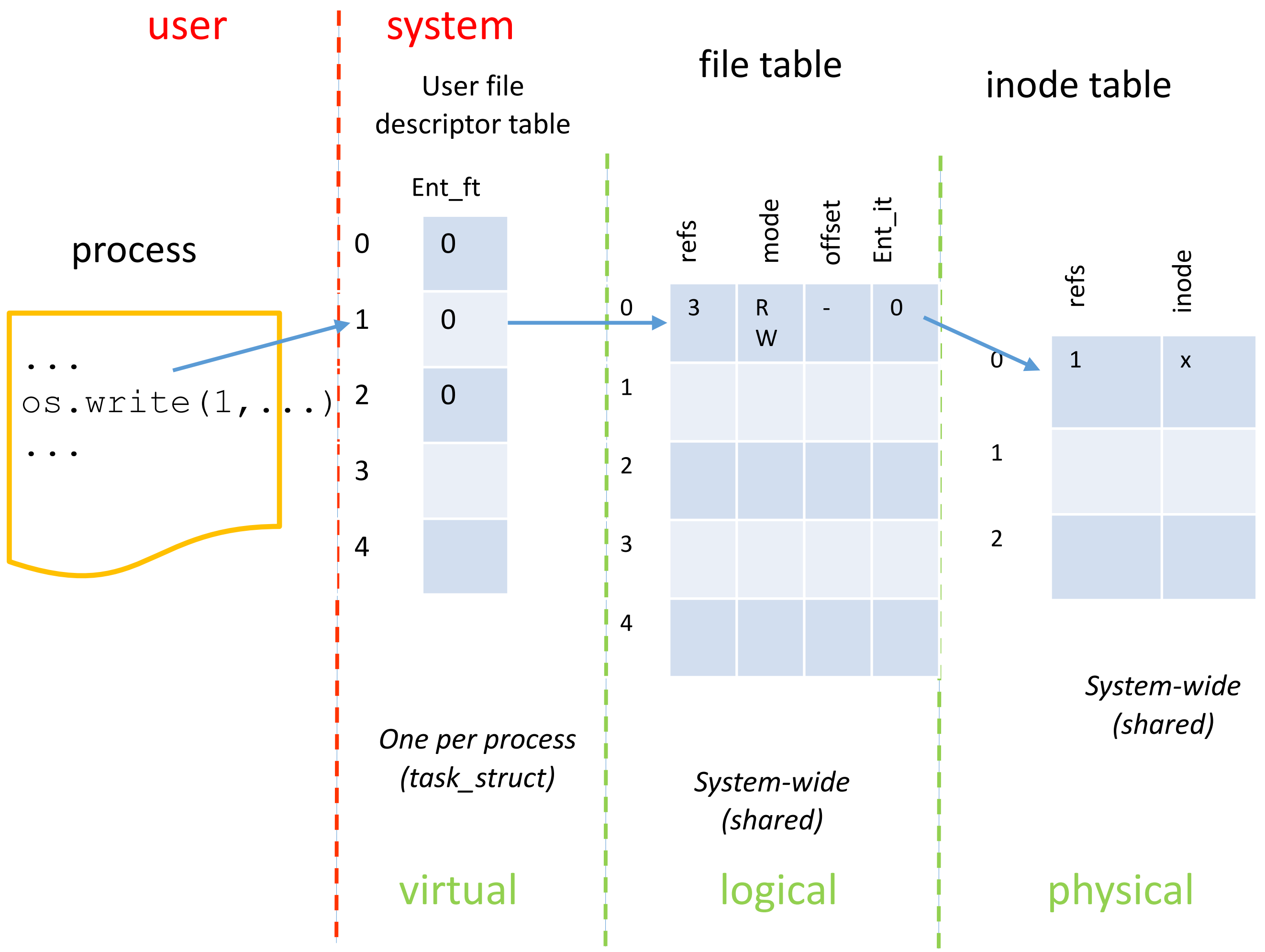
Kernel data structures: inode

- 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

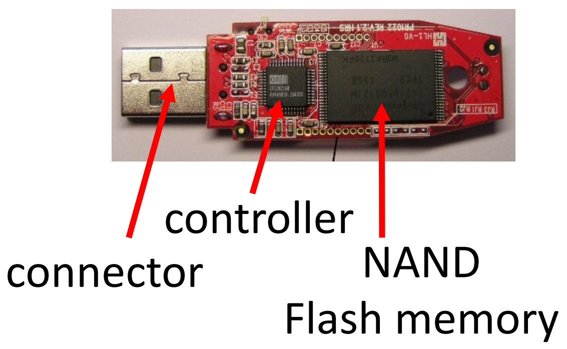
Kernel data structures



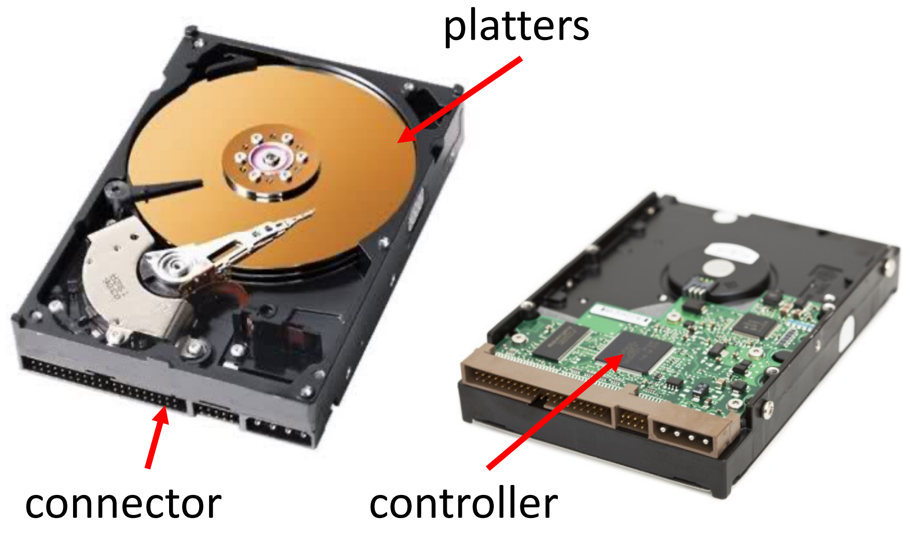
File system

Physical Storage Devices

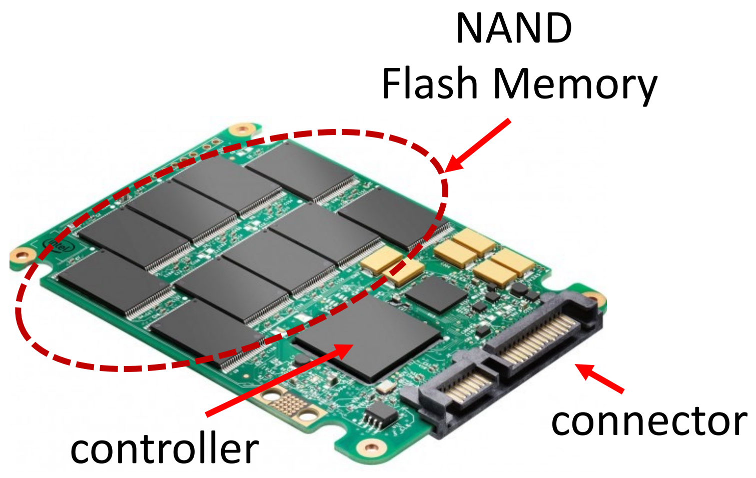
- Non-volatile memory to save data
- Similar vs Different components
  - Impact on performance and capacity



USB Drive



Hard Disk



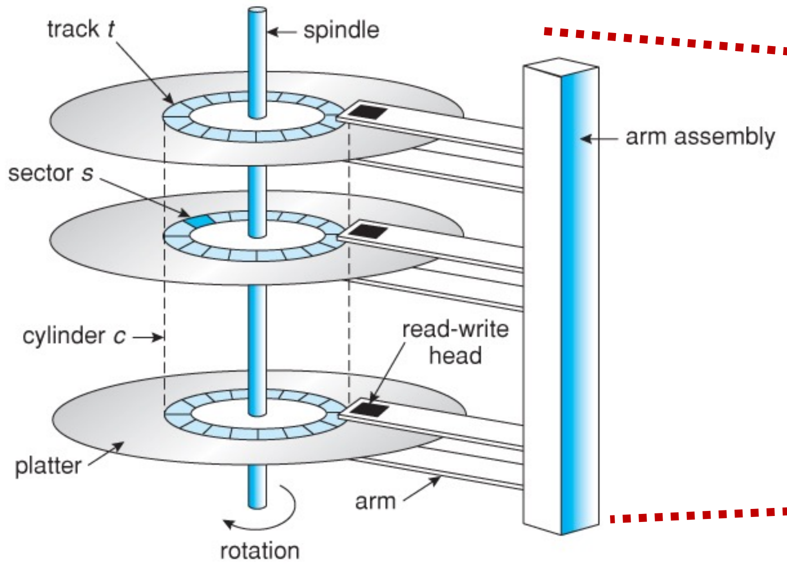
Solid State Drive (SSD)

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)

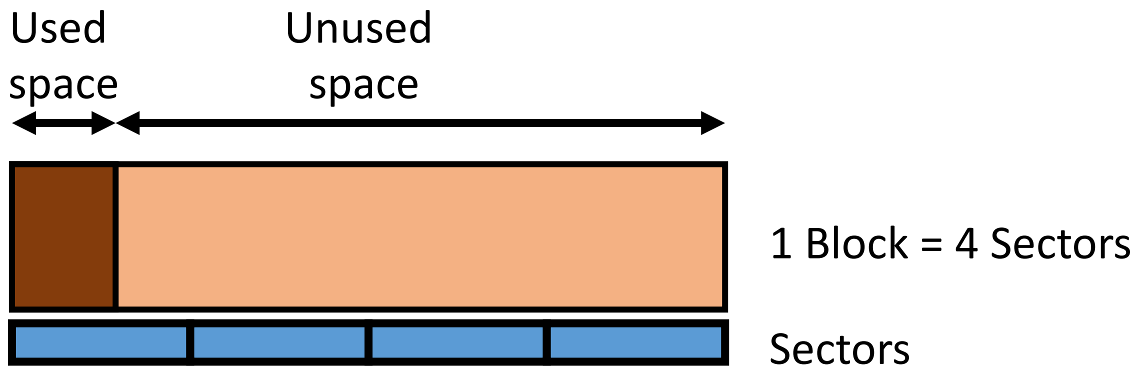
Some parameters that impact on performance

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



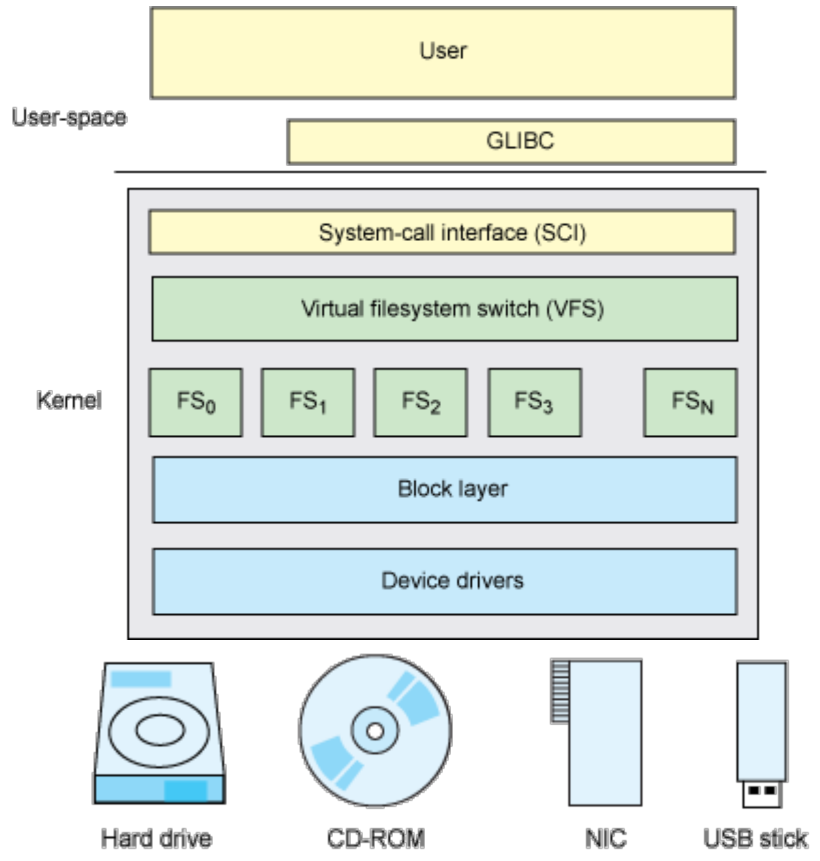
- What is the impact of a bad block size selection???
  - Too large: fragmentation (waste of space)
  - Too short: degrade performance too many accesses to the device

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

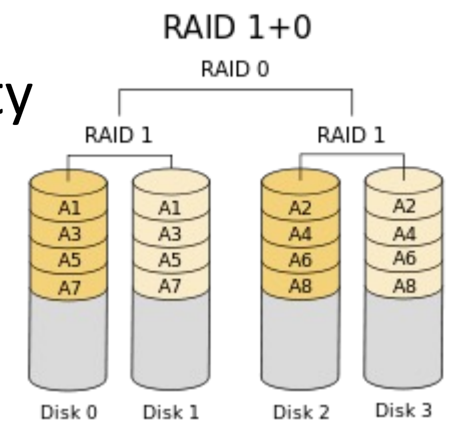
- Swap space: in UNIX/Linux OS is a special file system that extends main memory
  - Windows implements swap space in a single resizable file
- FUSE: Filesystem in Userspace
  - Let’s non-priviledge users implement their own file system without modifying the kernel (it is executed in user space rather 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

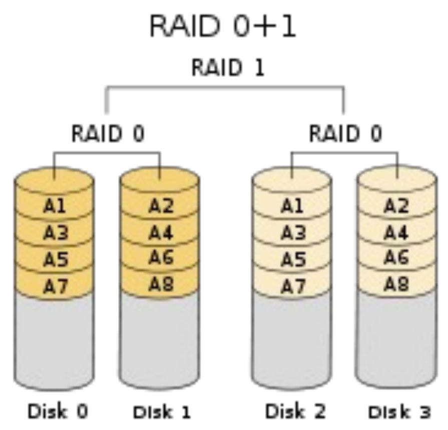
Journaling

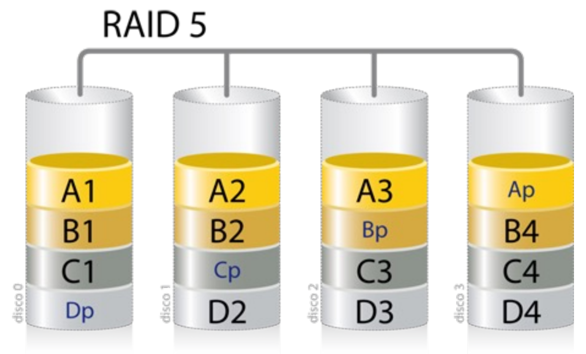
- Transaction based File System
  - Keep track of changes not yet committed to the file system
  - It records the changes in a “journal” file
    - It has a dedicated area in the file system
- In case of system failure or outage…
  - The file system can be brought back fast and with lower likelihood 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







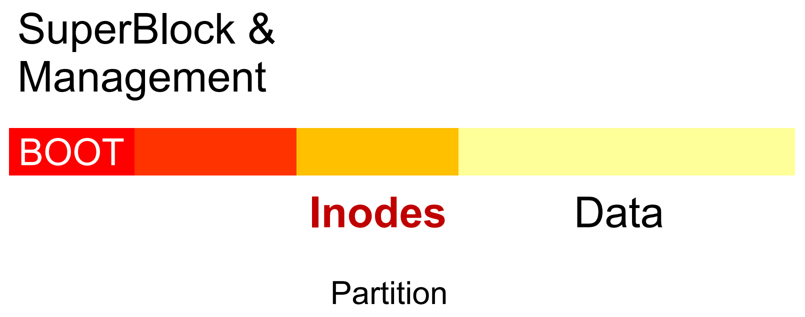
Performance Impact: handling blocks

- 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

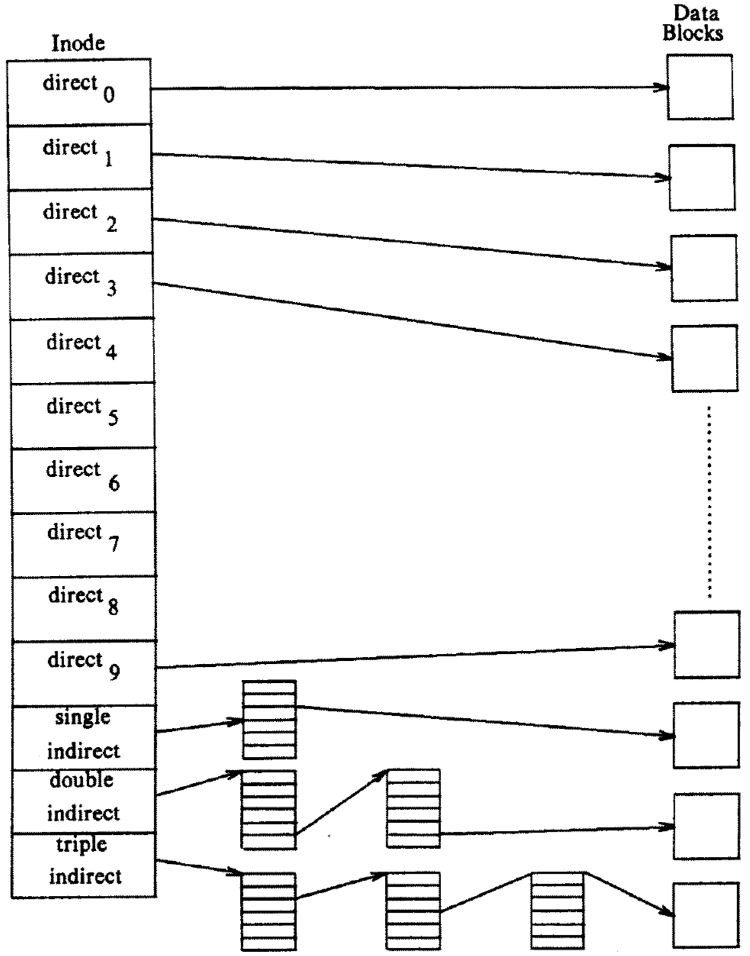
I-Node Based File System

- 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.



Managing used space

- For each file, OS must know where are located its blocks.
  - Indexed allocation: pointers to data blocks assigned to a file
  - Inode contains the indexes. How many?
    - multilevel index
- Indexes' blocks (B = 1KB, @ = 32b)
  - 10 direct blocks
    - (10 blocks = 10/KB)
  - 1 indirect block
    - (256 KB)
  - 1 double indirect block
    - (64 MB)
  - 1 triple indirect block
    - (16GB)
  - Classical AT&T System V UNIX approach (1983).



Directories: Organizing files

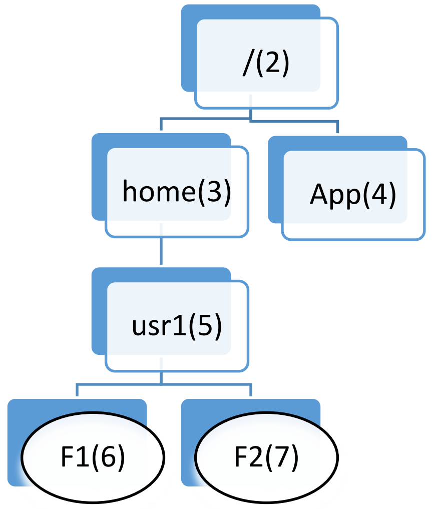
- **Directory:** Logical structure to organize files
  - Pathname: Relative (from any folder) vs Absolute (from the root folder)
- It is a particular type of file managed by the OS
  - "/" is the root folder
    - Every partition has its own root folder **(I-node ID 2)**
  - "." and ".." are mandatory 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

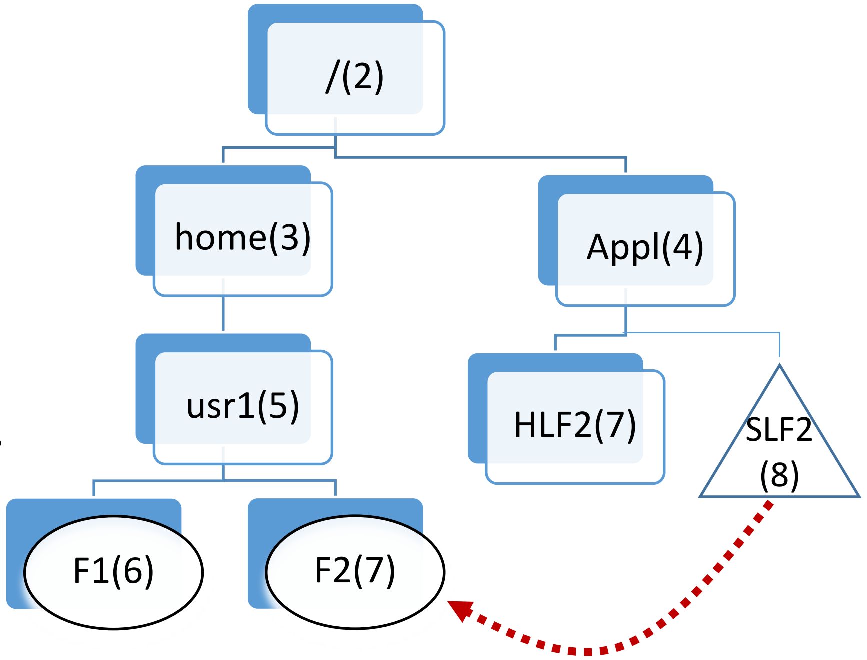
Directory: /home/usr1



Directories: Organizing files

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

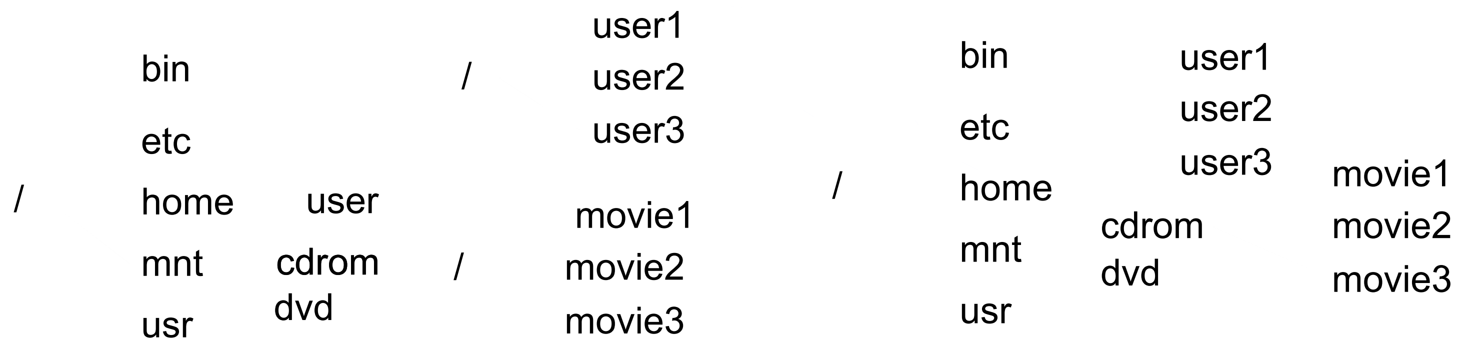
- Sharing files
  - Hardlinks
    - It only needs a new entry in a directory (name→I-node ID)
  - **Softlinks**
    - A **new file** that comprises a pathname
      - Similar to shortcuts in Windows
  - Pros/Cons/restrictions lead to use one or the other



Mount

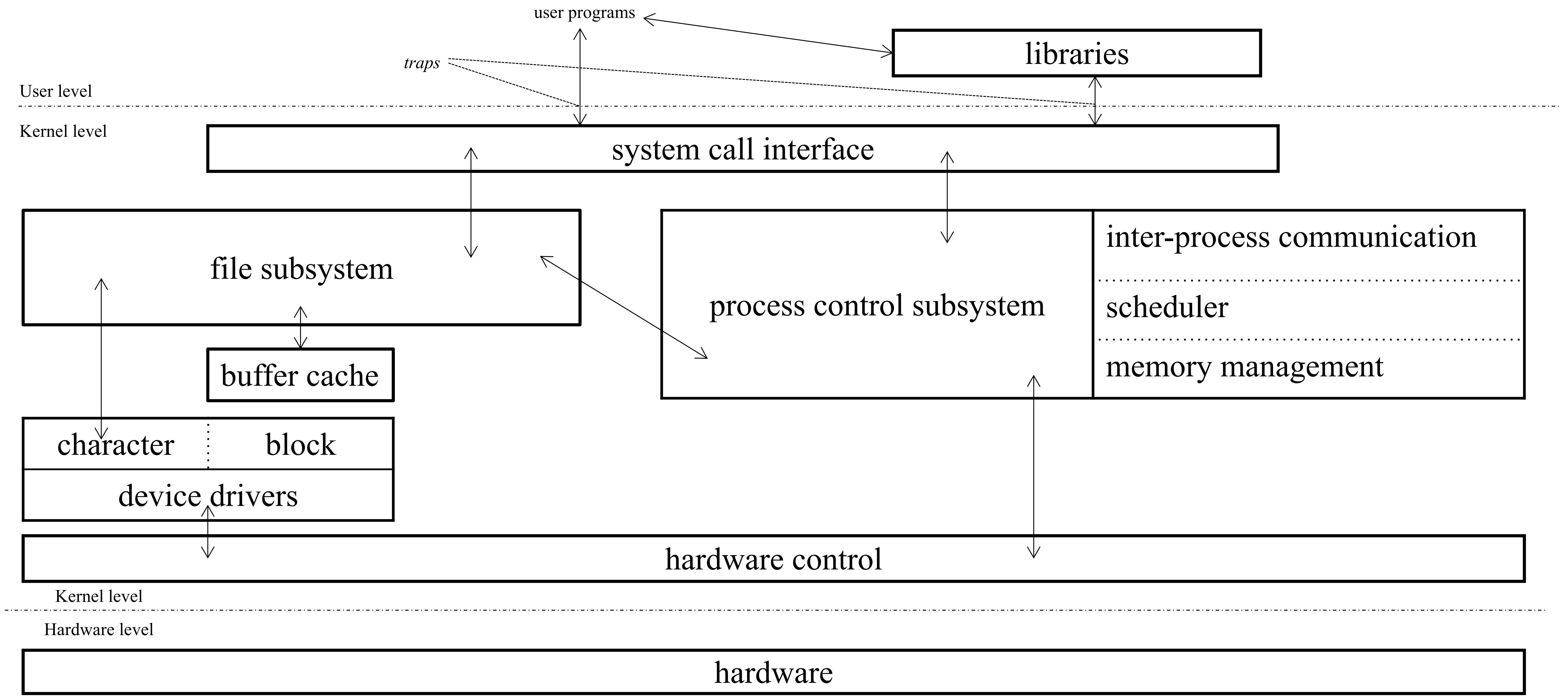
- Publishing the contents of a disk partition on the file system
  - Linux command line:

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



Basic system calls

Block diagram of the Unix System Kernel



Source: Bach, M. J. (1986). *The design of the UNIX operating system.* Prentice-Hall

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

Blocking operations

- **Blocking** I/O: A process ask for a transfer of N bytes and waits for the call completes. Returns the number of bytes transferred.
  - If there are data available (even if the number of bytes is smaller than requested) transfer is made and process returns immediately
  - If there are not data available the **process is blocked**
    1. Process state changes from RUN to BLOCKED
       1. Process leaves CPU and is queued in a waiting processes list
       2. The first process from the READY queue is moved to RUN (if Round Robin)
    2. When data are available an interrupt arrives
       - The ISR (Interrupt Service Routine) sends data to the CPU and enqueue the blocked process in the READY list (if Round Robin)
    3. When the turn is over, the process will be put into RUN again

Non-blocking operations

- **Non-blocking** I/O operation
  - A process ask for a transfer of N bytes. Data available at that time are sent and immediately returns whether there are data or not.
  - Control returns immediately with the data have been transferred.

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

`os.open(path, flags, mode=0o777, *, dir_fd=None)`

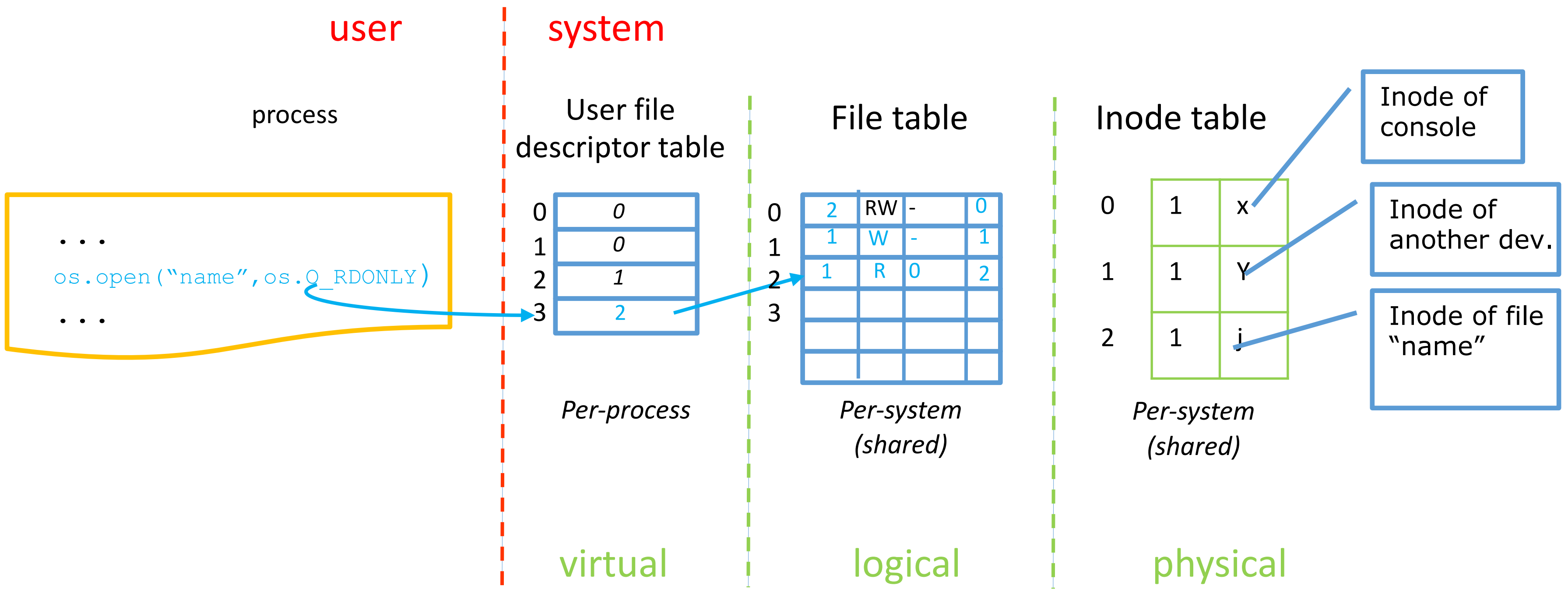
- Open the file `path` and set various flags according to `flags` and possibly its mode according to `mode`. When computing *mode*, the current `umask` value is first masked out. Return the file descriptor for the newly opened file. The new file descriptor is non-inheritable.
- Links a device (file name) to a virtual device (field descriptor)
  - `path` is a file name.
  - `flags` indicate the type of open. At least, one of them
    - `os.O_RDONLY` (reading)
    - `os.O_WRONLY` (writing)
    - `os.O_RDWR` (reading & writing)
  - `mode` gives the file permissions if the file is being created.
- Raises an auditing event open with arguments path, mode, flags.

Open: creating

- Regular files can be created with the system call `open` adding O_CREAT in the **`flags`** argument. The system call `mknod` creates special files.
- Argument **`mode`** is mandatory
  - It is the file permissions as **octal** number .
- There is not a syscall for parcilly remove data in a file. However, file can be truncated to length 0 adding O_TRUNC in the **`flags`** argument.
- Examples:
  - Ex1: `os.open("X",os.O_RDWR|os.O_CREAT,0o664)` → If file X did not exist it's created, but otherwise `O_CREAT` has no effect.
  - Ex2: `os.open("X",os.O_RDWR|os.O_CREAT|O_TRUNC,0o777)`→ If file X did not exist it's created, but otherwise file X is empty now.

Open: 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



```
...
os.open("name",os.O_RDONLY)
...
```

Read

`os.read(`*`fd`*`, `*`n`*`)` Read at most *n* bytes from file descriptor *fd*.

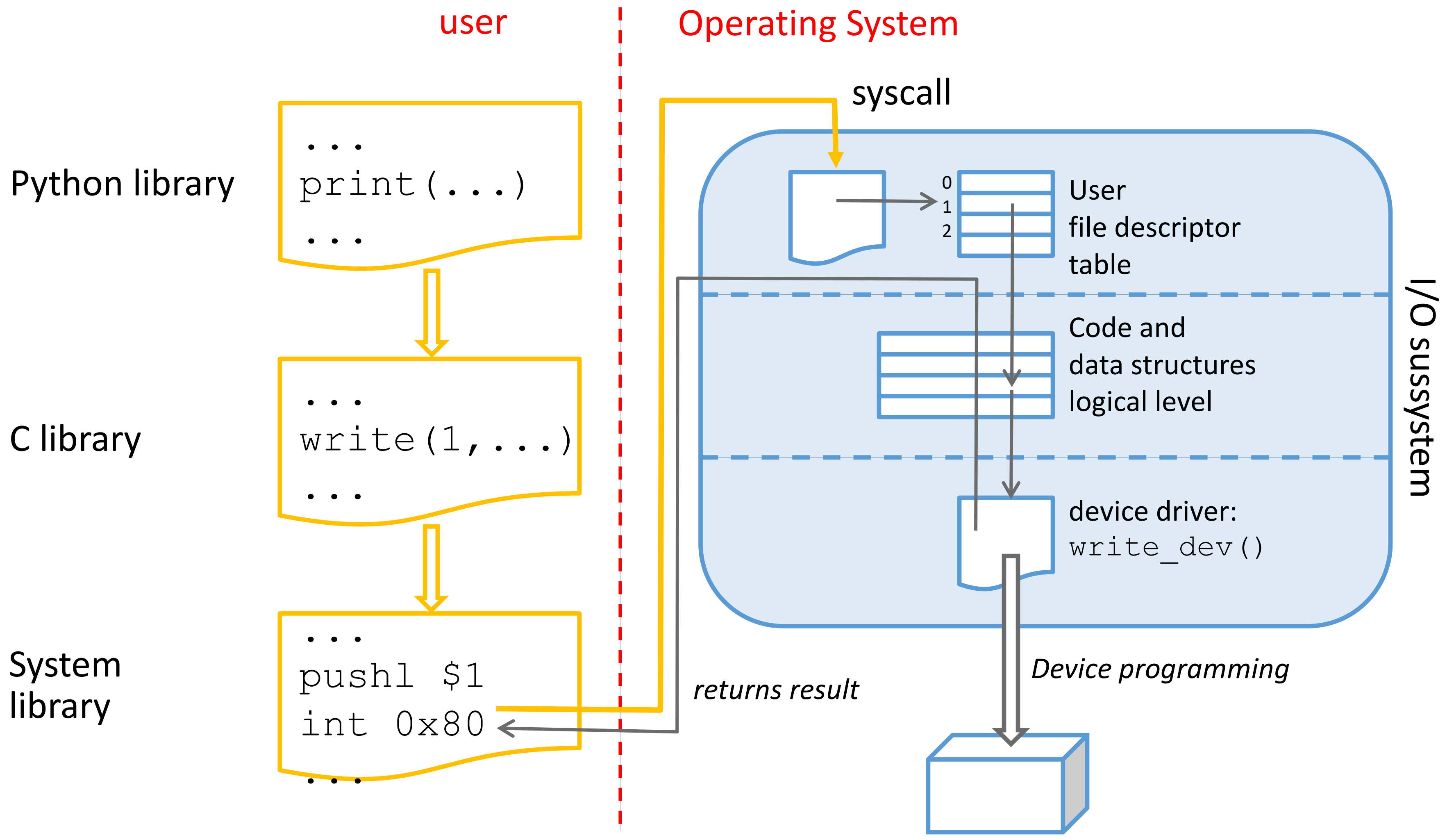
- Return a byte string containing the bytes read.
- If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.
- The kernel updates the offset in the file table to the n; consequently, successive reads of a file deliver the file data in sequence

Write

`os.write(fd, str)`

- Write the byte string in `str` to file descriptor `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
- Return 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



```
...
print(...)
...
```

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

```
...
pushl $1
int 0x80
...
```

Close

```
os.close(fd)
```

- Close file descriptor `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).

dup/dup2

```
newfd = os.dup(fd)
```

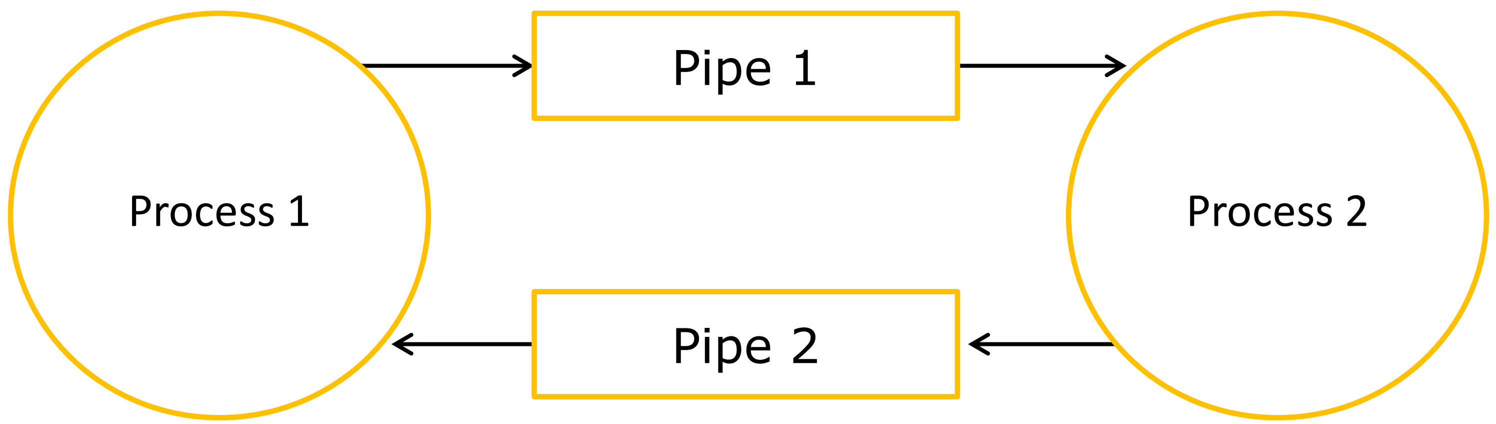
- ►Return a duplicate of file descriptor *fd*.
  - ►Where *fd* is the file descriptor being duped and *newfd* is the new file descriptor that references the file.
  - ►Copies a file descriptor into the first free slot of the user file descriptor table.

```
os.dup2(fd, fd2)
```

- ►Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary.
  - ►On success, return *fd2*.
  - ►Similar to *dup*, but the free slot is forced to be *newfd*

Pipes

- Interprocess communication by message passing



Unnamed pipes

```
fd = os.pipe()
```

- ►Create a pipe.
- ►Return a pair of file descriptors (r, w) usable for reading and writing, respectively.
- ► Only related processes, descendants of a processes that issued the pipe call can share access to unnamed pipes
- ► The kernel allocates 2 entries in the File Table and 1 in the Inode Table.

Named pipes

`os.mkfifo(`*`path, mode=0o666`*`)`

- ►Create a FIFO (a named pipe) named *path* with numeric mode *mode*
- ►The current umask value is first masked out from the mode.
- ► FIFOs are pipes that can be accessed like regular files.
- ► It does not open the FIFO.
  - ►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.

Pipes

- 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

Characteristics of pipes

- Reading
  - If there are data, a process reading from the pipe reads what it wants in a transient way
  - If there are no data, a process reading from the pipe is blocked until other process writes to the pipe.
  - If the pipe is empty and there are no writer processes (ie, all file descriptors opened for writing are now closed), reader process receives and EOF
  - Therefore, processes must close all unused file descriptors as soon as possible.
- Writing
  - If there is room for the data to be written, the kernel writes the data (or as many as possible)
  - If the pipe is full, writer process goes to sleep waiting for data to drain from the pipe
  - If there are no more readers, writer process receives a signal *SIGPIPE*
- Behaviour of pipes, like blocking, can be modified by syscalls (`fcntl`)

lseek

`new_offset = os.lseek(`*`fd, pos, how`*`)`

- ▶NO accesses to disk → only update I/O ptr (the file table offset)
- ▶ Subsequent `read` or `write` system calls use the file table offset as their starting byte offset.
- ▶`how`: os.SEEK_SET, os.SEEK_CUR, os.SEEK_END

examples

Byte by byte access

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

```
c=os.read(0,1)
while (len(c)>0):
    os.write(1,c)
    c=os.read(0,1)
```

byteBYbyte.py

- Reading while there are data (`len(c)>0`), that's up to the device. The total amount of syscalls 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.

Buffer in user space access

- Reading blocks of bytes

blockBYblock.py

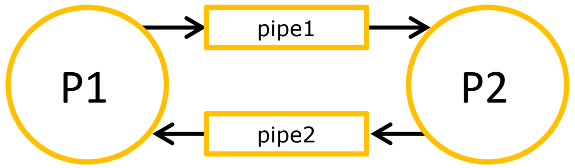
```
#!/usr/bin/python3 -s
import os

SIZE=256
c=os.read(0,SIZE)
while (len(c)>0):
    os.write(1,c)
    c=os.read(0,SIZE)
```

- What about performance? How many system calls are executed?

Data communication using pipes

- Program a process schema equivalent to the figure:



- 2 pipes
- P1 sends to pipe1 and receives from pipe2

```
def func_p1(fdin, fdout):
        buff = os.read(fdin,16)
        os.write(fdout,buff)
        return 0
```

- P2 the opposite symmetrically

```
def func_p2(fdin, fdout):
buff = b'Hello world\n'
os.write(fdout,buff)
buff = os.read(fdin,16)
os.write(1,buff)
return 0
```

dataComm.py

```
pipe1 = os.pipe()
pipe2 = os.pipe()
pidp1 = os.fork()
if (pidp1==0):
        os.close(pipe1[0])
        os.close(pipe2[1])
        func_p1(pipe2[0],pipe1[1])
        sys.exit(0)
os.close(pipe1[1])
os.close(pipe2[0])
pidp2 = os.fork()
if (pidp2 == 0):
        func_p2(pipe1[0],pipe2[1])
        sys.exit(0)
os.close(pipe1[0])
os.close(pipe2[1])
os.wait()
os.wait()
```

Random access and size evaluation

- What does this code do?

```
fd = os.open("Helian.fasta",os.O_RDONLY)
c = os.read(fd,1)
while (len(c)>0):
  os.write(1,c)
  os.lseek(fd,4,os.SEEK_CUR)
  c = os.read(fd,1)
```

rnd-access.py

- And this one?

```
fd =os.open("Helian.fasta",os.O_RDONLY)
sz = os.lseek(fd,0,os.SEEK_END)
print (sz)
```

fsz.py

Pipes and blocking

```
fd = os.pipe()
pid = os.fork()
if ( pid == 0): # child process
  c = os.read(0,1) # reads from stdin
  while (len(c)>0):
    os.write(fd[1],c) # and writes to the pipe
    c = os.read(0,1)
else: # parent process
  c = os.read(fd[0],1) # reads from the pipe
  while (len(c)>0):
    os.write(1,c) # and writes to the stdout
    c = os.read(fd[0],1)
    os.wait()
```

rnd-access.py,

- Be careful! The parent process must close `fd[1]` to prevent blocking.

Sharing the offset

- What does this code do?

```
fd = os.open("Helian.fasta",os.O_RDONLY)
nAs = 0
pid = os.fork()
c = os.read(fd,1)
while (len(c)>0):
  if (c == b'A’):
    nAs +=1
  c = os.read(fd,1)
print ("Number of A:",nAs)
os.close(fd)
```

sh-offset.py,

Non sharing the offset

- What does this code do?

```
nAs = 0
pid = os.fork()
fd = os.open("Helian.fasta",os.O_RDONLY)
c = os.read(fd,1)
while (len(c)>0):
  if (c == b'A’):
    nAs +=1
  c = os.read(fd,1)
print ("Number of A:",nAs)
os.close(fd)
```

nonsh-offset.py,

Bibliography

- Computer Systems. A programmers perspective
  - Randal E. Bryant, David R. O'Hallaron 2015. Chapter 10.
  - https://upfinder.upf.edu/iii/encore/record/C__Rb1318766
- Operating System
  - Silberschatz, A; Galvin, P. B; Gagne, G. 2019. Chapters (11-15)
  - https://cataleg.upf.edu/record=b1498664~S11*cat
- 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