

PYTHON3:

PARSER → Demanar i procesar els input en un programa.

LLISTA: Conjunt de dades que poden ser de diferents tipus (python).

TUPLA: Llista d' elements on cada registre té més d'un tipus de dada.

LIST COMPRENSION: Forma de crear una llista a partir d'un diccionari ?

DICCIONARI: A partir d'una key que no es pot repetir trobem un valor. Llista d'elements amb associació key-valor. On el valor pot ser qualsevol tipus de dades. I la key un string?

Tot programa comença amb aquestes 2 líneas:

```
# /usr/bin/python
#-*- coding: utf-8-*
```

Són l'interpret a utilitzar de les ordres i la codificació de les dades on en 8 bits podem guardar i representar qualsevol caracter .

```
import sys
import argparse
```

Aquesta llibreria (sys) ens permet fer:

~~fileIn=sys.stdin ← guardar l'entrada estandar sigui fitxer o text.~~

Tant si es fitxer com text, es pot anar processant linea a linea (el \n es qui delimita)

Recordar que sempre que obrim un fitxer, **I'HEM DE TACAR.:**

```
fitxer=open(sys.argv[1],"r") → o "w" write o "a" append
fitxer.close()
```

argv és una llista on el element [0] és el nom del programa cridat per python i els succesius són el parametres passats per l'usari.

break serveix per interrompre un bucle (un while o for)

Posar les constants sempre en majuscles: **MAXLIN=10**

En un print:

```
print(line, end='')
```

end="" es per imposar que no afegeixi un salt de linea (ja que line ja en porta un ! i per defecte el print en posa un altre)

ACABAR SEMPRE EL PROGRAMA AMB **exit(0) == Resultat exitós !!**

exit(1) == resultat no exitos sense especificar.

MOLT IMPORTANT PROVAR LES COSES EN EL SHELL DE PYTHON3 !!!!!
MOLT DE COMPTE AMB LES TABULACIONS I NO DEIXAR **CAP SALT DE LINEA SENSE CODI.**

```
python3  
>>>
```

```
parser.add_argument("-f","--fit",type=str,\n                    help="fitxer a processar", metavar="file",\n                    default="/dev/stdin",dest="fitxer")
```

SI UN ARGUMENT ES POSICIONAL (NO TE GUIÓ) NO TE SENTITN POSAR DEFAULT
PQ NO L'AGAFARÀ.

Apunts Ruben:

mkfifo --> creem estructures de dispositius (first input first output)
ES COM UNA REDIRECCIÓ DEL FLUXE DE DADES

mkfifo permet crear un **named pipe (es com un fixer que redirecciona)**

prw-r--r-- 1 marc marc 0 Jun 8 19:51 /tmp/dades

Ex:

mkfifo /tmp/dades (aquest "fitxer no existeix fins ara ")

ls -la / > /tmp/dades (i el pipe queda retingut)

grep samba < /tmp/dades (en un altre terminal)

=

I farà el ls- la / | grep samba (sense que el 'fitxer' pipe ocupi memoria alguna !!)

tail -f /tmp/dades ALTRE EXEMPLE

Senyals: (Per mirar llistar les senyals de kill→: **kill -l**)

- 1 - SIGHUP (Senyal que reinicia el procés)
- 2 - SIGINT (Senyal que plega el programa --> equivalent a CTRL + C)
- 9 - SIGKILL (Senyal que mata el procés **NO POT SER IGNORADA !!**)
- 10 - SIGUSR1 (Senyal sense funció assignada, està així perquè l'usuari el defineixi)
- 12 - SIGUSR2 (Senyals sense funció assignada, està així perquè l'usuari el defineixi)
- 14 - SIGALRM (Senyal que envia una alarma)
- 15 - SIGTERM (Senyal que li demana al programa que plegui ordenadament)
- 18 - SIGCONT (Senyal que demana al procés que continuï == a CTRL + Q)
- 19 - SIGSTOP (Senyal que para el procés --> equivalent a CTRL + S / CTRL + Z)

IPC - Inter proces Communication.

Comunicació entre procesos. Recordar un dimoni és un programa fill desvinculat d'un procés pare que queda actiu en segon pla (background, ja que no es interactiu) a la espera de events (alarmes..)/ordres.

Exemple:

~~El procés pare podria ser Nginx o Apache o IIS i un dimoni seria:~~

~~httpd → Dimoni que es queda a l'escolta de peticions del port 80/443~~

LLIBRERIA	QUE PERMET FER	METODES TIPICS PYTHON3
Import sys	Manipular fitxers/stdin	
Import argparse	Definir arguments opcionals o obligat.	<pre>parser = argparse.ArgumentParser(description=\ "Exemple popen") parser.add_argument("ruta",type=str,\ help="directori a llistar") args=parser.parse_args()</pre>
From subprocess import Popen,PIPE	<p>Popen es un constructor de PIPES en python (tuberies).</p> <p>Permet executar ordres i enviar el resultat via pipe cap a...</p>	<pre>command=[ordre,arg] # tupla amb mínim l'ordre i després els arguments (si n'hi ha) pipeData = Popen(command, stdout=PIPE) for line in pipeData.stdout: print(line.decode("utf-8"), end="")</pre>
Import os	<ul style="list-style-type: none"> - Podem obtenir el pid del proces actual (python) - Dividir el proces actual en 2 (pare i fill). - 	<pre>-os.getpid() →retorna pid del proces actual -os.fork() → Crea fill -os.execl(dicc) →???? Mirar cap l'ex 18 -os.execv →accepta llista o tuples on passem ordre i arguments (path no). -os.execl →no accepta llistes ni tuples per argument (per tant es sempre path, ordre i arguments.. -os.exece →últim valor serà variables entorn -os.execp →programa sense path</pre>
Import signal	Utilitzar events	<pre>def myHandler (signum,frame): (funcio) sys.exit(0) Típica funció que s'executa quan es produeix un event concret que li pasem com a argument/parametre → signal.signal(signal.SIGUSR1,myhandler) Aquí diem que si rep un event signal 10, executa la funció. signal.signal(signal.SIGTERM,signal.SIG_IGN) signal.alarm(20)</pre>
Import socket		Cal sempre tant a client com servidor

Popen crida a altres executables/ordres UNIX desde Python3

És un constructor de classes

Este método permite la ejecución de un programa como un proceso hijo.

El segundo argumento que es importante comprender es shell, que por defecto es False.

En Unix, cuando necesitamos ejecutar un comando que pertenece al shell, como ls -la, tenemos que configurar **shell=True**

<https://pharos.sh/comandos-popen-de-subprocesos-y-so-de-python/>

FORKING

pàgina 35 HOW TO

Definicions:

programa: El código escrito cuando no se está ejecutando .

proceso: El código en ejecución.

Cuando el programa se ejecuta en **os.fork ()**, el sistema operativo creará un nuevo proceso (proceso hijo) y luego copiará toda la información del proceso padre al proceso hijo.

Entonces, tanto el proceso padre como el proceso hijo obtendrán un valor de retorno de la función fork (), en el proceso hijo este valor debe **ser 0**, y el proceso padre es el número de identificación del proceso hijo.

pid=os.fork El fill bifurcat dona pid=0 i el pare bifurcat retorna el pid del fill.

Si fem get.pid llavors el pare retorna el seu PID i el fill el seu propi.

INFO. Ha de morir el pare perquè el fill tingui un PID diferent de 0 que serà el PID del pare+1.

exec:

Serveix per executar ordres UNIX en un programa Python.

El procés que s'està executant a python passa a convertir-se en el programa que se li ha passat (li passem el path del binari del programa), i després li podem passar una llista amb el **path del programa o binari ([0])**, l'ordre **([1])** i on s'executarà **aquesta ordre ([2] o més)**,

MAI executarà el codi que hi hagi sota seu. O sigui fet el exec.. Morirà el procés/programa de python.

Si no es fa un **os.fork()** no es generà un procés que es pugui convertir amb **os.exec** .!!!!!! i no funcionarà !! només un procés fill es pot transformar amb un procés/funció UNIX i després al executar-se morirà sense acabar el codi !!!

os.execl(path, **arg0**, **arg1**, ...) **l (literal)** no s'accepten llistes ni tuples com a argument

os.execl**e**(path, **arg0**, **arg1**, ..., **env**) **e(entorn)** passem variables d'entorn (exemple sql pasariem {nom:"marc", edat:17})

os.execl**p**(**file**, **arg0**, **arg1**, ...) **p(program)** no cal posar ruta absoluta, directament el programa

os.execv**e**(path, **args**, env) → **v(vector)** Els arguments poden ser tuples() o llistes[]

Exemples:

os.execvp("ls",["ls","/etc","/home/marc"]) → **Ull en la llista va dins també l'ordre que es l'element 0 de la llista**

os.execlp("ls","ls","-ls","/") --> farà **ls -l /**

os.execvpe("ls",["ls","/etc","/home/marc"],{nom:"joan",edat:"13"}) FUNCIONA

os.execv("/usr/bin/python3", ["/usr/bin/python3", "16-signal.py", "60"]) ->

Es com si posesim a manija : **python3 16-signal.py 60**

COMENCEM SOCKETS:

Per comunicar-nos necessitem regles(protocolos) per saber quan enviar/rebre la informació.

Sabem que una comunicació és única o la comunicació queda definida :

'IP origen:Port origen --> (Socket origen) + IP destí:Port destí --> (Socket destí)

La combinació SocketOrigen + SocketDestí és ÚNICA !!

Comandes Python:

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

opcions: SOCK_STREAM == TCP o SOCK_STREAM == UDP

s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

AQUEST LINEA D'ADALT LA POSEM SEMPRE PER EVITAR PROBLEMES.

Popoen:

SI ENVIEM EL RESULTAT D'UNA ORDRE UNIX SEMPRE ES FA PER POPEN !!! A CLIENT O SERVIDOR

SI S'ENVIA TEXT SOLAMENT NO CAL (UN ECHO o un print) !!!

shell=True es posa si hem d'enviar via Popen una ordre que necessiti accedir al shell (tipus ls -la)!!

SI VOLEM FER UN CLIENT SENCILL, QUE NOMES REBI I REPRESENTI TEXT, PODEM HABILITAR UN telnet o netcat en local.

SERVIDOR (PQ ESCOLTA):

nc -l 51000 (escoltem peticions del port X, si el client finalitza la connexió acaba)

(si posem l'opció **-k** no finalitzarà la connexió tot i que el client tanqui sessió)

CLIENT:

Obrim connexió amb aquest port i li passem un fitxer:

nc localhost 50001 < fitxer.txt Així rebrem per stdout el contingut (text) del fitxer.

o si volem fer un xat:

nc localhost 50001 (com no hem especificat esperarà stdin, i cada cosa que escrivim depés d'un enter l'enviarà cap al port)

EX	CLIENT	SERVIDOR	Serv aten
21	Obre connexió, envia 1 sol string. Espera 1 resposta. Finalitza connexió.	Rep string i el renvia cap a client el mateix string. Fi	Simple
22	Obre connexió, i espera rebre dades que va printant en blocs de màx 1024 bytes, quan no rep més dades, tanca socket.	Es queda escoltant si rep 1 sola connexió, fa Popen amb un cat i un argument, envia linea a linea, tanca connexió i mor.	Simple
23	NO HI HA	Es queda escoltat si conexions. Rep conexio Executa Popen fent "id" amb l'usuari actual l'envia pel pipe, quan ha enviat tot tanca ell connexió.	OneBOne
24	Es conecta espera resposta fins que servidor digui "se fini", printa tot el rebut, fins a 1024 bytes per linea i tanca socket	Es queda escoltat si conexions. Rep conexio Executa Popen fent "cal" amb arguments i l'envia pel pipe, quan ha enviat tot tanca ell connexió.	OneBOne
25	Obre socket. Inicia una connexió, executa comanda en local i envia via Popen al pipe cap al servidor. Envia linea a linea, quan acaba tanca el socket de connexió.	Es queda escoltant si conexions. Rep connexió i obre un fitxer en mode W, on va grabant la info rebuda. Quan no rep més dades (client ha finalitzat), tanca fitxer i tanca connexió amb aquest socket. I segueix escoltant..	OneBOne
26	Obre socket. Inicia una connexió. Inicia una especie de cli. (si l'usuari no envia comanda tanca socket.) Codifica la el string amb comanda i l'envia pel socket. Client es queda rebent resposta fins que rebí un byte x04 (hex) que voldra dir que era l'ultima linea a rebre. I tornara a esperar una altre ordre del usuari.	Es queda escoltant si conexions. Rep connexió. Li passen 1 comanda, (si no rep comanda tanca connexió) Executa la comanda UNIX mitjançant Popen i envia el resultat (linea a linea) pel pipe. Quan acaba de transmetre envia un x04 al final de l'última cadena per indicar al client que s'ha acabat d'enviar el resultat. I segueix escoltant peticions de connexió.	OneBOne
27	NO HI HA Podem provar fent per cada client: telnet localhost 50007	Va escoltant conexions. Cada connexió nova la guarda i la té en compte de cara rebre info. Rep fins a 1024 bytes per cada client conectat y els renvia cap a	Multi

		client a cada tanda. Que vol dir ? Que escolta fins a 1024 d'un client, els reenvia a aquest client; passa al següent client amb connexió i fa el mateix.	
28	NO HI HA	Igual pero customitzant port i ara rebem una comanda que executem i enviem el resultat (i els errors) al client amb Popen i enviat el famós x04 per indicar que hem acabat de transmetre.	Multi

PROGRAMA 21 I OBSEVACIONS: ES COM FER UN TELNET (o nc)

CLIENT:

El client envia un 'Hello world' passat a binari, que el tornarà a rebre rebotat i el printarà.

```
#docstring: Programa client que envia el text Hello
HOST = ''
PORT = 51000 # port per on enviarem la petició
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT)) # Ens connectem TOC,TOC (establim conex)
s.accept() # si obtenim resp seguim
s.send(b'Hello, world') # Enviem el missatge 'Hello' (b dades binàris)
data = s.recv(1024) # Espera fins rebre dades que com a molt tindrà 1024 bytes i el guarda a data (per tant sabem que data trunrarà a 1024).
s.close() # Llibera el socket
print('Received', repr(data)) # Printem el que hem rebut repr era per passat a text el binari rebut
sys.exit(0)
```

`data = s.recv(1024)` → Vol dir que escoltarem fins a 1024 bytes (buffer) i seguirem el programa cap abaix. Si el missatge és més llarg quedarà truncat a no ser que estiguem en un bucle escoltant tota l'estona.

RECORDAR: Un print() per defecte afegeix un salt de línia al final !!

SERVIDOR:

Tornarà el mateix "Hello world"

```
#docstring: Programa servidor que rep text
HOST = '' # si no val res es localhost
```

```

PORT = 51000 # port per on escoltem peticions
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Constructor...
#quan diu 'STREAM' és en TCP, 'DGRAM' és UDP.
s.bind((HOST,PORT)) # lliga el servei amb una connexió(socket) (IP:port)
s.listen(1) # S'ha de passar per força a escoltar
conn, addr = s.accept() # queden clavats fins que es produeix event quan
establim connexió (s'ha connectat algú) guarda/retorna tupla amb objecte
connexió i la IP del client connectat.
while True: # Bucle infinit (mentres arribin dades)
    data = conn.recv(1024) # El client envia les dades, el servidor
    les rep i les guarda de moment
    if not data: # if not data = s'ha tancat la connexió, el servidor
    finalitzarà (SERVIDOR!)
        break # surt del while
    conn.send(data) # Enviem les mateixes dades (max 1024 bytes per bloc)
conn.close() # Client tanca la connexió (CLIENT!)
sys.exit(0) # Finalitzem programa

```

PROGRAMA 22. Ara el client no sap quantes dades ha de rebre de resposta !! I el servidor al voler **enviar el stdout d'una ordre UNIX ha d'utilitzar una PIPE !!.**

Ara el client només establint connexió rebrà X línies d'informació, ha d'escoltar infinitament fins que el servidor digui que ha acabat (com ho diu? no eviant més dades, que es tradueix amb una finalització de la connexió)

```

conn, addr = s.accept() # Accepta la connexió
command = ["date"] # Li especifiquem el command que utilitzarem
pipeData = Popen(command,stdout=PIPE) # Executem el popen, l'ordre
s'executa en el serv i s'envia via PIPE cap a client

for line in pipeData.stdout: # Retornem cada una de les línies que
retorna l'ordre
    conn.send(line) # Enviem cada línia
conn.close() # Això ES FLAG PQ EL CLIENT S'APIGA QUE JA S'HA acabat la
transmissió

```

EL CLIENT VEURÀ TOTA L'ORDRE ID de l'usuari del servidor EN 1 LINEA ja que hi cap en 1024 bytes:

```
marc@debian:~/Documents/python_ipc/ipc-2021$ python3 22-daytime-client
.py
Data: b'uid=1000(marc) gid=1000(marc) groups=1000(marc),24(cdrom),25(f
loppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),11
3(bluetooth),119(scanner),998(docker)\n'
```

Si fiquessim un missatge més llarg de 1024 passaria això, desgranaria en 2:

[illegible]

PROGRAMA 23: Port custom amb parser + Modificació del servidor
 pq accepti una petició rere una altre. **ONE_BY_ONE**
 (equival a l'ordre **nc -lk <port>**)

NOU:

```
s.listen(1)
while True: # No acaba mai d'escoltar conexions entrants, no finalitza
# encara que el client faci un s.close()
    conn, addr = s.accept()
    print("Connected by", addr)
command = ...
```

PROGRAMA 24 **PRO**: CLIENT + SERVIDOR.

```
24-calendar-server-one2one.py [-p port] [-a any]
```

Calendar server amb un popen, el client es connecta i rep el calendari. El server tanca la connexió amb el client un cop contestat però continua escoltant noves connexions.

El server ha de governar-se amb senyals que fan:

- sigusr1: llista de peers i plega.
- sigusr2: count de listpeer i plega.
- sigterm: llista de peers, count i plega.

El server és un daemon que es queda en execució després de fer un fork del seu pare (que mor) i es governa amb senyals.

Que sigui el servidor qui rep l'any com a argument és una tonteria, seria més lògic fer-ho en el client, però el diàleg client servidor queda per a una pràctica posterior. Aquí es vol practicar usar un arg en el popen.

SERVER:

```
import sys,socket,os,signal,argparse
from subprocess import Popen, PIPE

llistaPeers=[] # Definim la llista buida de les connexions entrants

def mysigusr2(signum,frame):
    # docsting Definim la funció del signal usr2 (kill -12)
    print("Signal handler called with signal:", signum)
    print(len(llistaPeers)) # Printem el len (quantitat) de la llista de connexions
    sys.exit(0) # Pleguem

pid=os.fork() # dividim

if pid != 0:      # Fem l'if en funció el PID al pare (ens dirà quin es el PID fill) AQUI NOMES ENTRA EL PARE
    print("Engegat el server CAL:", pid)
    sys.exit(0) # Programa 'pare' finalitzarà
```

```
# NOMÉS S'EXECUTARÀN AL PROGRAMA FILL JA QUE EL PROGRAMA PARE JA ES
MORT!!!
signal.signal(signal.SIGUSR1,mysigusr1)

# I A PARTIR D'AQUI IGUAL AL 23

llistaPeers.append(addr) # Afegim les adreces a la llista de
connexions (per tenir un historic acumulat)
command = "cal %d" % (ANY) # Especifiquem la commanda que s'executarà
(%d (integer), %s (string)--> any passat per argument)
pipeData = Popen(command,shell=True,stdout=PIPE) # Popen (shell=True
--> es perquè funcioni)
```

25 PROGRAMA ON EL CLIENT ES QUI ENVIA INFO AL SERVIDOR I AQUEST GUARDA EN UN FITXER

addr= guarda en un diccionari on addr[0]=IP o host addr[1]=PORT
manipula fitxer en un path concret.

També hi ha signals:

26 PROGRAMA QUE EMULA TELNET: (Client es qui tanca connexió).

Servidor es queda escoltat connexions entrants. Client connecta servidor, si servidor accepta, client envia una ordre (amb **cmd=cmd.encode** passa a binari) que ha capturat el programa via input (stdin), el servidor l'executa l'ordre i crea un pipe(popen) per on enviar linea per linea el resultat i quan caba envia **con.send(b'\x04')** i, el client que ha estat escoltant, un cop arriben dades espera fins un paquet que incorpora final de transmissió de dades **data[-1:]==b'\x04'** , imprimeix el resultat per pantalla i torna a demanar al usuari una ordre (bucle infinit), el client si envia "enter" sense comande el servidor enten que ha de tallar la connexió (penja).

CLIENT: Parser port servidor

while True

 data = s.recv(2) → Vol dir que data només guardarà 2 caracters i els imprimirà a continuació (si posem 1024 imprimiria tranquilament fins a 1024 char o fins a trobar salt de linea)

SERVIDOR: Parser port i debug(verbose)

```
if data[-1:] == b'\x04': # (quedat lo de la dreta corrent un espai a l'esquerra)
```

```
    break
```

Explicació cambrer cervesata-olives:

Cambrer = Servidor

Client = Nosaltres

Demanam unes cervesetes i després unes olives.

NO SABEM QUINA comanda arribarà abans si les olives o les cerveses, pero si sabem que l'última en arribar portarà el compte (b'\x04')

27 PROGRAMA QUE ACEPTA MULTIPLES CONEXIONS (CLIENTS) SIMULTANEAMENT

Tenim 2 connexions establertes.

A conns tenim:

conns[0] és la connexió propia del port 50007

conns[1] és la 1era connexió que hem fet amb telenet →43902

conns[2] és la següent que hem fet.

A **actius** només està guardada la connexió propia, fins que no rebem una connexió externa o dades externes. Llavors s'agregarà el socket de qui envia dades en aquest moment. Per tant mai hi haurà més de 2 elements en aquesta llista.

Cada cop que entra una connexió nova o rebem noves dades d'una connexió ja establerta saltam a la linea:

```
actius,x,y = select.select(conns,[],[])
```

I actualitza els sockets actius (actius sempre hi ha el socket propi i el que està atenent ara mateix)

```

home > marc > Documents > python_ipc > ipc-2021 > 27-echo-server-multi.py > ...
15 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
16 s.bind((HOST, PORT))
17 s.listen(1)
18 print(os.getpid()) # donem el pid per poder matar el servidor, ja que no acaba mai.
19 conns=[s] # afegim sempre el socket que escolta, el 1er (SINO NO VA, HA D'INICIALIZAR AMB UNA CONEIXO)
20 while True: # forever
21     actius,x,y = select.select(conns,[],[]) # només interessa actius (es una llista)
22     for actual in actius: # cada cop que entra una connexió aquesta val s
23         if actual == s: #
24             conn, addr = s.accept() # TIPIC D'ACCEPTAR UNA CONX ENTRANT
25             print('Connected by', addr)
26             conns.append(conn) # la connexió actual l'afegim a conns pq poguem rebre dades d'ella
27         else: # SI LA CONEX JA EXISTEIX, ESCOLTA DADES
28             data = actual.recv(1024)
29             if not data:
30                 sys.stdout.write("Client finalitzat: %s \n" % (actual))
31                 actual.close()
32                 conns.remove(actual) # treiem de la llista la connexió finalitzada
33             else:
34                 actual.sendall(data) #
35                 #actual.sendall(b'chr(4)',socket.MSG_DONTWAIT)
36
37 s.close()# TANCA SOCKET
38 sys.exit(0) #FI PROGRAMA
39
40
41

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```

env /bin/python3 /home/marc/.vscode/extensions/ms-python.python-2022.4.1/pythonFiles/lib/python/launcher 3539
5 -- /home/marc/Documents/python_ipc/ipc-2021/27-echo-server-multi.py
7592
Connected by ('127.0.0.1', 43902)
Connected by ('127.0.0.1', 44504)

```

```

t localhost 50007
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
ls
ls
ps

```

```

telnet: port 500007 out of range
marc@debian:/var/tmp/projecte_asix2/codi_terraform/codi_terraform_definitiu$ telnet localhost 50007
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

```

28 IDEM QUE 27 PERO ENVIANT ORDRES EMULANT TELNET !!

Es pot utilitzar el client 26 per fer la prova.

Pipes:

- ordre1 | ordre2 --> El contingut de la ordre 1 passa a l'ordre 2 **EX:** grep "10" /etc/passwd | wc -l

/proc --> Representació virtual dels processos del sistema

cat | sort | wc -l > cara.txt

isx48062351@i24:/tmp/m01\$ ls -la /proc/11730/fd

total 0

dr-x----- 2 isx48062351 hisx2 0 Jan 18 11:23 .

dr-xr-xr-x 9 isx48062351 hisx2 0 Jan 18 11:23 ..

lrwx----- 1 isx48062351 hisx2 64 Jan 18 11:23 0 -> /dev/pts/1 --> entrada estàndard (stdin)

l-wx----- 1 isx48062351 hisx2 64 Jan 18 11:23 1 -> 'pipe:[234590]' --> stdout

lrwx----- 1 isx48062351 hisx2 64 Jan 18 11:23 2 -> /dev/pts/1 --> sortida estàndard (stderr)

isx48062351@i24:/tmp/m01\$ ls -la /proc/11731/fd

total 0

dr-x----- 2 isx48062351 hisx2 0 Jan 18 11:24 .

dr-xr-xr-x 9 isx48062351 hisx2 0 Jan 18 11:23 ..

lr-x----- 1 isx48062351 hisx2 64 Jan 18 11:24 0 -> 'pipe:[234590]'

l-wx----- 1 isx48062351 hisx2 64 Jan 18 11:24 1 -> 'pipe:[234592]'

lrwx----- 1 isx48062351 hisx2 64 Jan 18 11:24 2 -> /dev/pts/1

isx48062351@i24:/tmp/m01\$ ls -la /proc/11732/fd

total 0

dr-x----- 2 isx48062351 hisx2 0 Jan 18 11:24 .

dr-xr-xr-x 9 isx48062351 hisx2 0 Jan 18 11:23 ..

lr-x----- 1 isx48062351 hisx2 64 Jan 18 11:24 0 -> 'pipe:[234592]'

l-wx----- 1 isx48062351 hisx2 64 Jan 18 11:24 1 -> /tmp/m01/carta.txt

lrwx----- 1 isx48062351 hisx2 64 Jan 18 11:24 2 -> /dev/pts/1

A partir del número 3, són personalitzats

mkfifo --> creem estructures de dispositius **EX:** mkfifo /tmp/dades

Per esborrar les estructures de dispositius: **EX:** rm /tmp/dades

isx48062351@i24:/tmp/m01\$ ls -la /tmp/dades

prw-r--r-- 1 isx48062351 hisx2 0 Jan 18 11:26 /tmp/dades --> Tipus 'pipe' (named pipe)

isx48062351@i24:/tmp/m01\$ tail -f /tmp/dades --> '-f' de follow (cada cop que executem una ordre dins d'aquest directori, el follow ens ho anirà mostrant)