

# T4-Entrada/Sortida



# Llicència

---

Aquest document es troba sota una llicència  
**Reconeixement - No comercial - Compartir Igual**  
sota la mateixa llicència 3.0 de Creative Commons.

Per veure un resum de les condicions de la llicència, visiteu:  
<http://creativecommons.org/licenses/by-nc-sa/3.0/deed.ca>



# Índex

---

- Conceptes bàsics d'E/S
- Dispositius: Virtual, lògic i Físic
- Gestió de l'E/S a Linux
  - Estructures de dades del kernel
  - Crides a sistema bàsiques
  - Exemples
- Sistema de Fitxers
  - Linux: Relació entre crides a sistema / estructures de dades

---

# **CONCEPTES BÀSICS D'E/S**

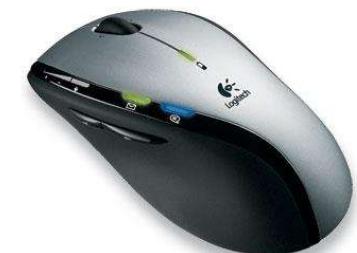
# Què és l'E/S?

---

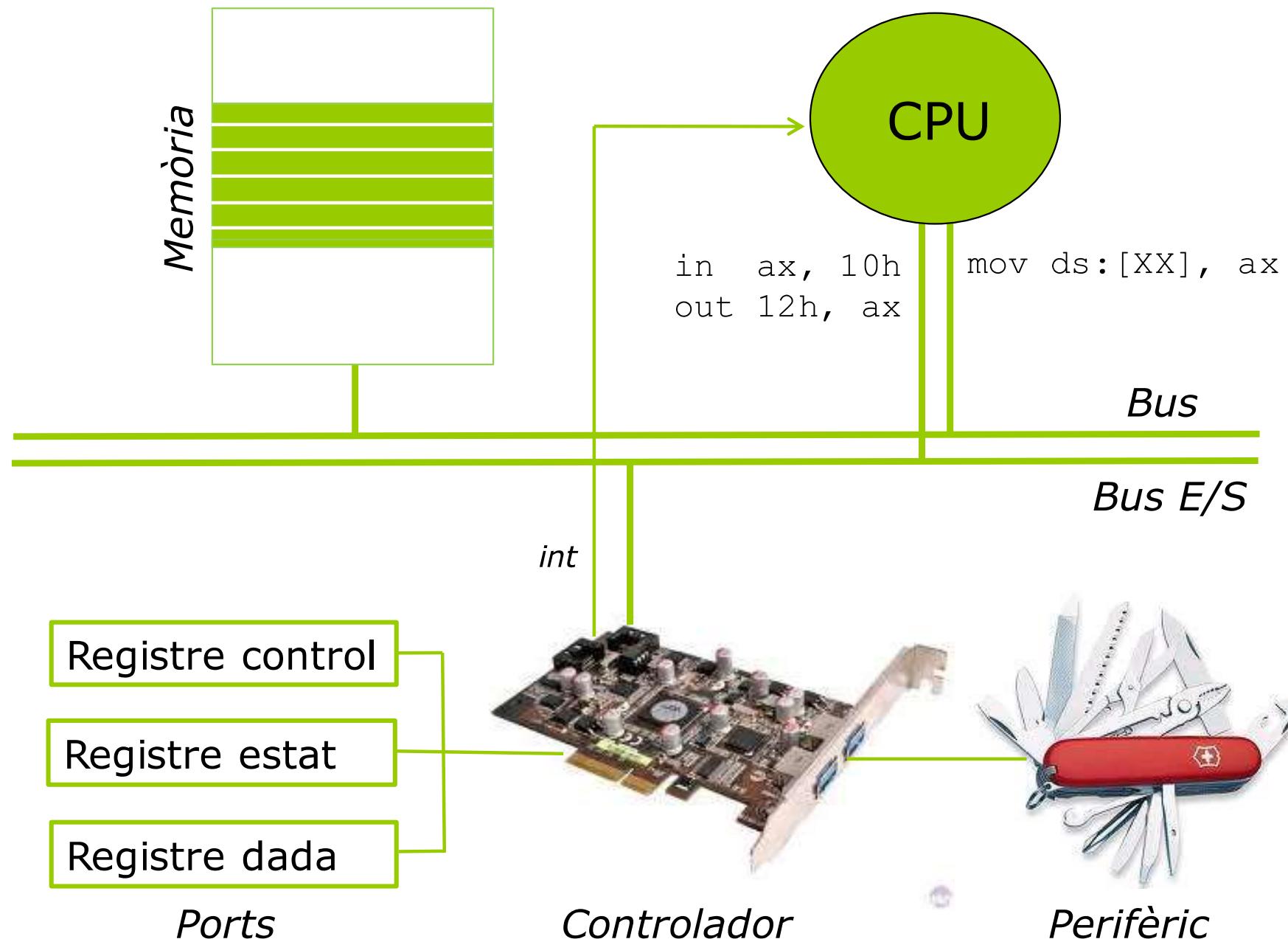
- **Definició:** transferència d'informació entre un procés i l'exterior
  - Entrada de dades: de l'exterior al procés
  - Sortida de dades: del procés a l'exterior  
(sempre des del punt de vista del procés)
- De fet, bàsicament, els processos realitzen càlcul i E/S
- Molts cops, inclús, l'E/S és la tasca principal d'un procés:  
p.ex. navegació web, intèrpret de comandes, processador de texts
- **Gestió de l'E/S:** administrar el funcionament dels dispositius d'E/S (perifèrics) per a un ús **correcte, compartit i eficient** dels recursos

# Dispositius d'E/S

---



# Visió HW: Accés als dispositius d'E/S



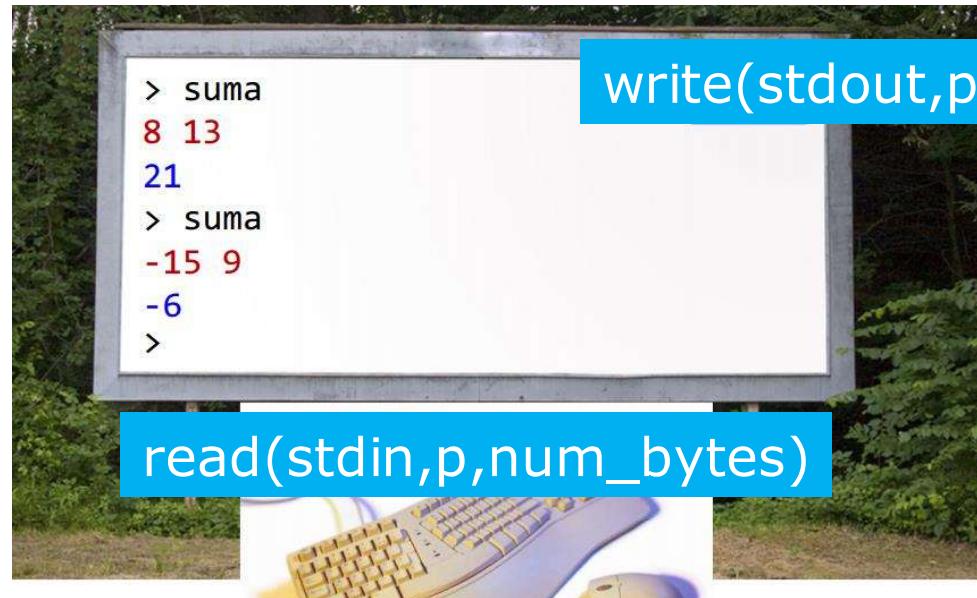
# Visió usuari (fins ara) (PRO1)



Entrada: **cin**. Llegeix les dades i les processa per adaptar-les al tipus de dades

Sortida: **cout**. Processa les dades i les escriu

# En aquest curs veurem que hi ha al mig



luction to Programming

© Dept. LSI, UPC

4

- No serà tan fàcil com a C o C++
- La conversió de tipus de dades és responsabilitat de l'usuari  
(és el que fan les llibreries de C o C++)

---

# **DISPOSITIUS: VIRTUAL, LÒGIC, FÍSIC**

# Tipus de dispositius

---

- Dispositius d'interacció amb l'usuari (pantalla, teclat, mouse), emmagatzemament (disc dur, DVD, pen drive), transmissió (mòdem, xarxa amb cable, xarxa sense cable) o inclús d'altres més especialitzats (controlador d'un avió, sensors, robots) ... moltíssima varietat!
- Classificació segons:
  - Tipus de dispositiu: lògic, físic, de xarxa
  - Velocitat d'accés: teclat vs disc dur
  - Flux d'accés: mouse (byte) vs DVD (bloc)
  - Exclusivitat d'accés: disc (compartit) vs impressora (dedicat)
  - Altres...
- Dilema: estandardització vs nous tipus de dispositius

**CONCEpte: Independència dels dispositius**

# Independència: principis de disseny

---

- L'objectiu és que els processos (codi principalment) sigui independent del dispositiu que s'està accedint
- Operacions d'E/S **uniformes**
  - Accés a tots els dispositius mitjançant les **mateixes crides al sistema**
  - Augmenta la simplicitat i la portabilitat dels processos d'usuari
- Utilitzant **dispositius virtuals**
  - El procés no especifica concretament sobre quin dispositiu treballa, sinó que utilitza uns identificadors i hi ha una traducció posterior
- Amb possibilitat de **redirecccionament**
  - El sistema operatiu permet a un procés canviar l'assignació dels seus dispositius virtuals

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



# Independència: principis de disseny

---

- Per això, habitualment, disseny en tres nivells: **virtual**, **lògic** i **físic**
- El primer nivell ens aporta independència, el **procés treballa amb dispositius virtuals** i no necessita saber que hi ha al darrere
- **El segon nivell, ens aporta compartició de dispositius.** Diferents accessos concurrents al mateix dispositiu.
- D'aquesta manera es poden escriure programes que realitzin E/S sobre dispositius (virtuals) sense especificar quins (lògic).
- En el moment d'executar el programa es determina dinàmicament sobre quins dispositius es treballa
  - Pot ser un paràmetre del programa, pot ser “heredat” del seu pare,...
- El tercer nivell separa les operacions (software) de la implementació. Aquest codi és molt baix nivell, moltes vegades en assemblador

# Dispositiu virtual

- **Nivell virtual:** Aïlla l'usuari de la complexitat de gestió dels dispositius físics
  - Estableix correspondència entre **nom simbòlic** (nom de fitxer) i l'aplicació d'usuari, a través d'un **dispositiu virtual**
    - ▶ Un nom simbòlic és la representació al sistema d'un dispositiu
      - /dev/dispX o bé .../dispX un nom de fixter
    - ▶ Un dispositiu virtual representa un dispositiu en ús d'un procés
      - Dispositiu virtual = canal = descriptor de fitxer. És un **nombre enter**
      - Els processos tenen 3 canals estàndard:
        - » Entrada estàndard canal 0 (stdin)
        - » Sortida estàndard canal 1 (stdout)
        - » Sortida error estàndard canal 2 (stderr)
  - Les crides a sistema de transferència de dades utilitzen com identificador el dispositiu virtual



# Dispositiu lògic

---

## □ Nivell lògic:

- Estableix correspondència entre dispositiu virtual i dispositiu (físic?)
- Gestiona dispositius que poden tenir, o no, representació física
  - ▶ P.ex. Disc virtual (sobre memòria), dispositiu nul
- Manipula blocs de dades de mida independent
- Proporciona una interfície uniforme al nivell físic
- Ofereix compartició (accés concurrent) als dispositius físics que representen (si hi ha)
- En aquest nivell es tenen en compte els permisos, etc
- A Linux s'identifiquen amb un nom de fitxer



# Dispositiu físic

- **Nivell físic:** Implementa a baix nivell les operacions de nivell lògic
  - Tradueix paràmetres del nivell lògic a paràmetres concrets
    - ▶ P.ex. En un disc, traduir posició L/E a cilindre, cara, pista i sector
  - Inicialitza dispositiu. Comprova si lliure; altrament posa petició a cua
  - Realitza la programació de l'operació demandada
    - ▶ Pot incloure examinar l'estat, posar motors en marxa (disc), ...
  - Espera, o no, a la finalització de l'operació
  - Retorna els resultats o informa sobre algun eventual error
  - A Linux, un dispositiu físic s'identifica amb tres paràmetres:
    - ▶ Tipus: **Block/Character**
    - ▶ Amb dos nombres: major/minor
      - **Major:** Indica el tipus de dispositiu
      - **Minor:** instància concreta respecte al major



# Device Drivers

---

- Per oferir independència, es defineix el conjunt d'operacions que ofereixen els dispositius
  - És un superconjunt de totes les operacions que es poden oferir per accedir a un dispositiu físic
  - No tots els dispositius ofereixen totes les operacions
  - Quan es tradueix de disp. Virtual a lògic i físic , d'aquesta manera està definit quines operacions corresponen

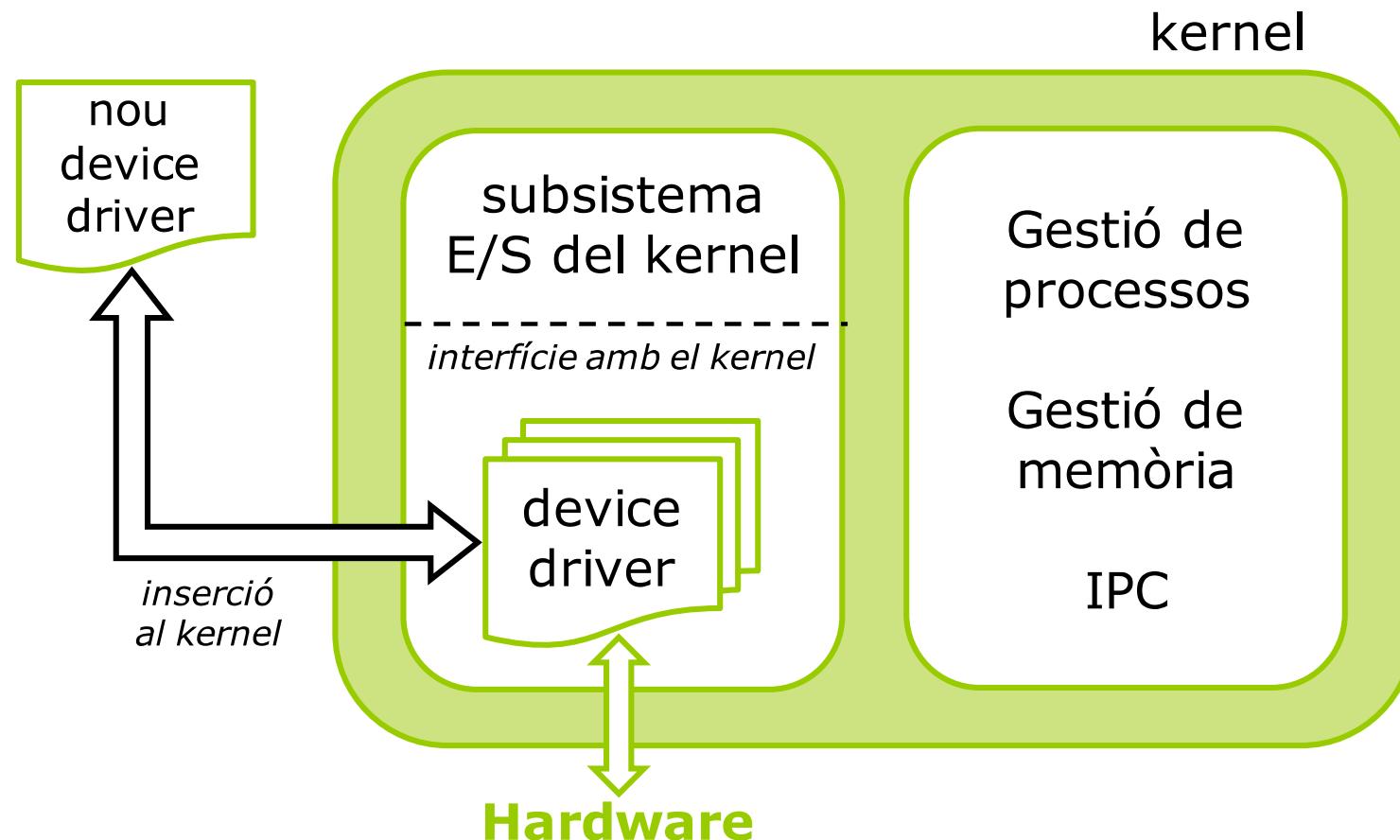
# Device drivers

---

- Els programadors del SO no poden generar codis per tots els dispositius, models, etc
- Cal que el fabricant proporcioni el conjunt de routines de baix nivell que implementen el funcionament del dispositiu
  - ▶ **Al codi+ dades per accedir a un dispositiu se'l coneix com Device driver**
  - ▶ Seguint l'especificació de la interfície d'accés a les operacions d'E/S definida pel sistema operatiu
- Per afegir un nou dispositiu podem:
  1. Opció 1: Recomilar el kernel
  2. Opció 2: Afegir nous dispositius sense recomilar el kernel
    - ▶ Cal que el SO ofereixi un mecanisme per afegir dinàmicament codi/dades al kernel
      - *Dynamic Kernel Module Support, Plug & Play*

# Device Driver

- S'identifiquen unes operacions comunes (interfície) i les diferències específiques s'encapsulen en mòduls del SO: **device driver**
  - Aïlla, la resta del kernel, de la complexitat de la gestió dels dispositius
  - A més, protegeix el kernel en front d'un codi escrit per "altres"



# Device Driver

---

- **Controlador de dispositiu (en genèric):** Conjunt de rutines que gestionen un dispositiu, i que permeten a un programa interactuar amb aquest dispositiu
  - Respecten la interfície definida pel SO (*obrir, llegir, escriure, ...*)
    - ▶ Cada SO té definida la seva pròpia interfície
  - Implementen tasques dependents del dispositiu
    - ▶ Cada dispositiu realitza tasques específiques
  - Contenen habitualment codi de baix nivell
    - ▶ Accés als ports d'E/S, gestió d'interrupcions, ...
  - Queden encapsulades en un arxiu binari

# Inserció dinàmica de codi : Mòduls de Linux

---

- Els kernels actuals ofereixen un mecanisme per afegir codi i dades al kernel,**sense necessitat de recompilar el kernel**
  - ▶ Una recompilació completa del kernel podria tardar hores...
- Actualment la inserció es fa dinàmicament (en temps d'execució)
  - *Dynamic Kernel Module Support* (linux) o *Plug&Play* (windows)
- Mecanisme de moduls a Linux
  - Fitxer(s) compilats de forma especial que contenen codi i/o dades per afegir al kernel
  - Conjunt de comandes per afegir/eliminar/veure mòduls
  - Ho veurem al laboratori

# Que tendremos en el lab...MyDriver1.c

- Un fichero con un device driver para un dispositivo “inventado”. Para no tener que recompilar el kernel, insertamos el driver usando un módulo.

```
struct file_operations fops_driver_1 = {  
    owner: THIS_MODULE,  
    read: read_driver_1,  
};
```

Esta estructura define las operaciones que ofrece el driver

```
int read_driver_1 (struct file *f, char __user *buffer, size_t s, loff_t *off) {  
    ...  
    return size;  
}
```

En este caso, solo la operación de lectura (read)

```
static int __init driver1_init(void){  
    ...  
}  
static void __exit driver1_exit(void){  
    ...  
}
```

Operaciones para cargar/eliminar el driver del kernel

```
module_init(driver1_init);  
module_exit(driver1_exit);
```

Operaciones del módulo de carga/descarga

# Device Driver (DD) a linux: +detalle

- Contingut d'un *Device Driver* (DD)
  - **Informació general sobre el DD:** nom, autor, llicència, descripció, ...
  - **Implementació de les funcions genèriques d'accés als dispositius**
    - ▶ *open, read, write, ...*
  - Implementació de les funcions específiques d'accés als dispositius
    - ▶ Programació del dispositiu, accés als ports, gestió de les ints, ...
  - Estructura de dades amb llista d'apuntadors a les funcions específiques
  - **Funció d'inicialització**
    - ▶ S'executa en instal·lar el DD
    - ▶ Registra el DD en el sistema, associant-lo a un *major*
    - ▶ Associa les funcions genèriques al DD registrat
  - **Funció de desinstal·lació**
    - ▶ Desregistra el DD del sistema i les funcions associades
- Exemple de DD: veure `myDriver1.c` i `myDriver2.c` a la documentació del laboratori



# Mòduls d'E/S a linux: +details

## □ Pasos per afegir i utilitzar un nou dispositiu:

- **Compilar** el DD, si escau, en un format determinat: .ko (kernel object)
  - ▶ El tipus (block/character) i el major/minor estan especificats al codi del DD
- **Instal·lar** (inserir) en temps d'execució les routines del driver
  - ▶ insmod fitxer\_driver
  - ▶ Recordar que el driver està associat a un major

Físic

## □ **Crear un dispositiu lògic** (nom d'arxiu) i lligar-lo amb el disp. Físic

- ▶ Nom arxiu  $\leftrightarrow$  block|character + major i minor

### ▶ **Comanda** mknod

— mknod /dev/mydisp c major minor

Lògic

## □ **Crear el dispositiu virtual (crida a sistema)**

— open("/dev/mydisp", ...);

Virtual

# Exemples de dispositius (1)

---

- Funcionament d'alguns dispositius lògics: terminal, fitxer, pipe, socket

## 1. Terminal

- Objecte a nivell lògic que representa el conjunt teclat+pantalla
- “Habitualment” els processos el tenen a com a entrada i sortida de dades

## 2. Fitxer de dades

- Objecte a nivell lògic que representa informació emmagatzemada a “disc”. S’interpreta com una seqüència de bytes i el sistema gestiona la posició on ens trobem dintre aquesta seqüència.

# Exemples de dispositius (2)

---

## □ Pipe

- Objecte a nivell lògic que implementa un buffer temporal amb funcionament FIFO. Les dades de la pipe s'esborren a mesura que es van llegint. Serveix per intercanviar informació entre processos
  - ▶ Pipe **sense nom**, connecta només processos amb parentesc ja que només es accessible via herència
  - ▶ Pipe **amb nom**, permet connectar qualsevol procés que tingui permís per accedir al dispositiu

## □ Socket

- Objecte a nivell lògic que implementa un buffer temporal amb funcionament FIFO. Serveix per intercanviar informació entre processos que es trobin en diferents computadores connectades per alguna xarxa
- Funcionament similar a les pipes, tot i que la implementació interna és molt més complexa ja que utilitza la xarxa de comunicacions.

---

# GESTIÓ DE L'E/S A LINUX



# Tipus de fitxers a Linux

---



- A Linux, tots els dispositius s'identifiquen amb un fitxer (que pot ser de diferents tipus)
  - block device
  - character device
  - Directory
  - FIFO/pipe
  - Symlink
  - **regular file → Son els fitxers de dades o “ordinary files” o fitxers “normals”**
  - Socket
- Anomenarem fitxers “especials” a qualsevol fitxer que no sigui un fitxer de dades: pipes, links, etc

# Creació de fitxers a Linux



- La majoria de tipus de fitxers es poden crear amb la crida a sistema/comanda mknod
  - Excepte directoris i soft links
  - Al laboratori utilitzarem la comanda, que **no serveix per crear fitxers de dades**
- mknod nom\_fitxer TIPUS major minor
  - TIPUS= c → Dispositiu de caràcters
  - TIPUS=b → Dispositiu de blocs
  - TIPUS=p → Pipe (no cal posar major/minor)



# Tipus de Fitxers

- Alguns exemples:

dev name	type	major	minor	description
/dev/fd0	b	2	0	floppy disk
/dev/hda	b	3	0	first IDE disk
/dev/hda2	b	3	2	second primary partition of first IDE disk
/dev/hdb	b	3	64	second IDE disk
/dev/tty0	c	3	0	terminal
/dev/null	c	1	3	null device

---

# **ESTRUCTURES DE DADES DEL KERNEL**

# Estructures de dades del kernel: Inodo

---



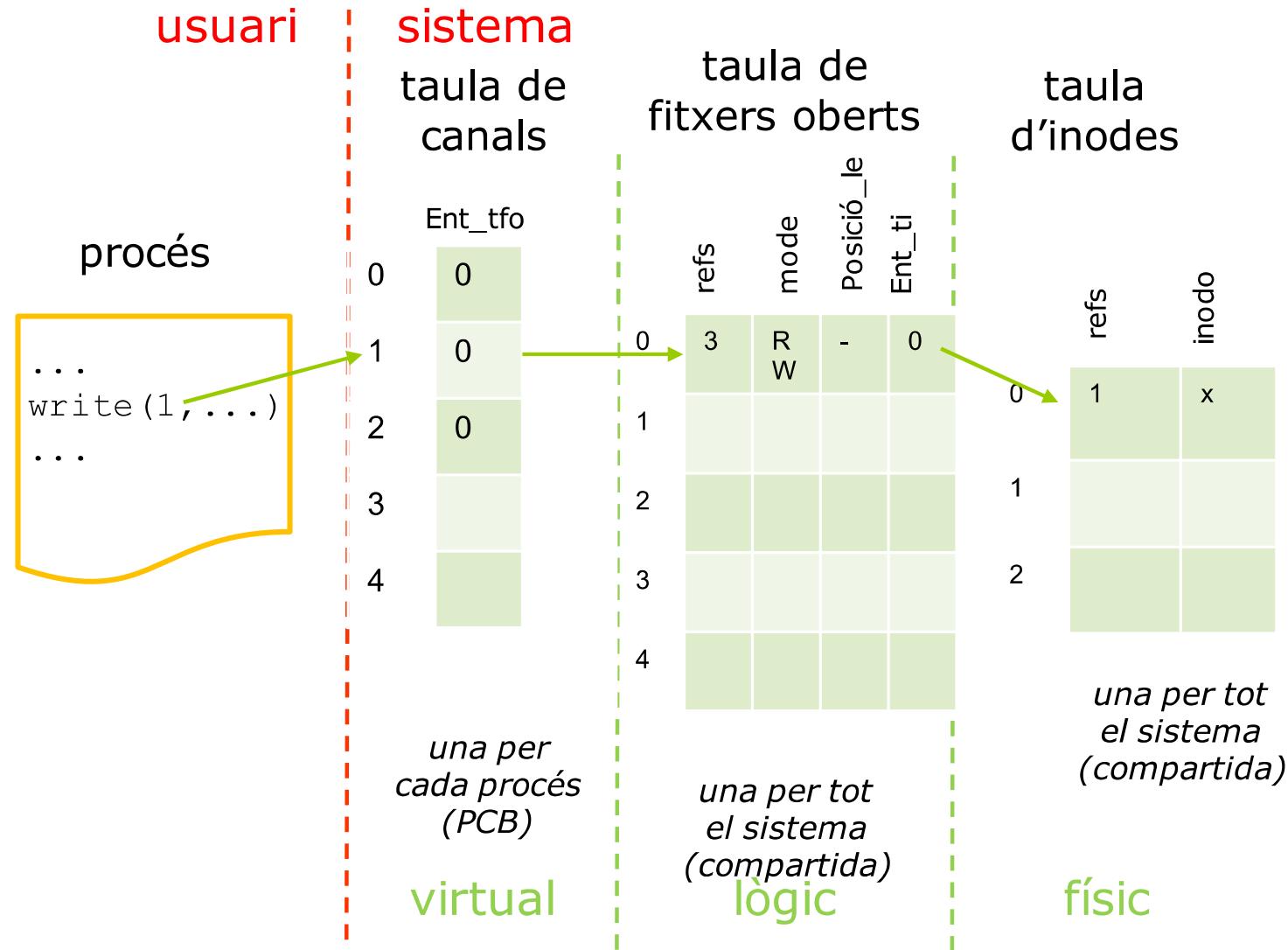
- Estructura de dades que conté tota la informació relativa a un fitxer
  - tamany
  - tipus
  - proteccions
  - propietari i grup
  - Data de l'últim accés, modificació i creació
  - Nombre d'enllaços al inodo (nombre de noms de fitxers que apunten a...)
  - índexos a les dades (indexación multinivel) → ho veurem al final del tema relacionat amb la gestió de la informació del disc
- Està tota la informació d'un fitxer excepte el nom, que està separat
- S'emmagatzema a disc, però hi ha una copia a memòria per optimitzar el temps d'accés

# Estructures de dades de kernel



- Per procés
  - Taula de canals (TC): per cada procés (es guarda al PCB)
    - ▶ Indica a quins fitxers estem accedint.
    - ▶ S'accedeix al fitxer mitjançant el canal, que és un **índex a la TC**
    - ▶ El canal és el disp. Virtual
    - ▶ Un canal referència a una entrada de la taula de fitxers oberts (TFO)
    - ▶ Camps que assumiren: num\_entrada\_TFO
- Global:
  - Taula de fitxers oberts (TFO):
    - ▶ Gestiona els fitxers en ús en tot el sistema
    - ▶ Poden haver entrades compartides entre més d'un procés i per varies entrades del mateix procés
    - ▶ Una entrada de la TFO referencia a una entrada de la TI.
    - ▶ Camps que assumirem: num\_referencies, mode , punter\_le, num\_entrada\_ti
  - Taula d'inodes (TI):
    - ▶ Conté informació sobre cada objecte físic obert, incloent les rutines del DD
    - ▶ És una copia en memòria de les dades que tenim al disc (es copia a memòria per eficiència)
    - ▶ Camps que assumirem: num\_referencies, dades\_inode

# Estructures de dades de kernel



---

# **CRIDES A SISTEMA BÀSIQUES**

# Ops bloquejants i no bloquejants (1)

---

- Algunes operacions poden ser costoses en termes de temps, el procés no pot estar consumint CPU sense fer res → El SO bloqueja el procés (RUN→BLOCKED)
- Operació d'E/S **bloquejant**: Un procés realitza una transferència d'N bytes i s'espera a que acabi. Retorna el nombre de bytes que s'han transferit
  - Si les dades es poden disposar al moment (encara que potser no totes), es realitza la transferència i el procés retorna immediatament
  - Si les dades no estan disponibles, el procés es bloqueja
    1. Canvi d'estat de RUN a BLOCKED
      1. Deixa la CPU y passa a una cua de processos que estan esperant.
      2. Es posa en execució el primer procés de la cua de preparats (Si Round Robin)
      3. Quan arriben les dades es produeix una interrupció
        - La RSI recull la dada i posa el procés a la cua de preparats (Si Round Robin)
      4. Quan li toqui el torn, el procés es posarà novament en execució

# Ops bloquejants i no bloquejants (1)

---

- Operació d'E/S **no bloquejant**
  - El procés sol·licita una transferència, disposa les dades que pugui en aquell moment, i retorna immediatament tant si hi ha dades com no
  - Retorna el nombre de dades transferides

# Operacions bàsiques d'E/S

Crida a sistema	Descripció
open	Donat un nom de fitxer i un mode d'accés, retorna el nombre de canal per poder accedir
read	<b>Llegeix N bytes d'un dispositiu</b> (identificat amb un nombre de canal) i ho guarda a memòria
write	Llegeix N bytes de memòria i els <b>escriu al dispositiu</b> indicat (amb un nombre de canal)
close	Allibera el canal indicat i el deixa lliure per ser reutilitzat.
dup/dup2	Duplica un canal. El nou canal fa referència a la mateixa entrada de la TFO que el canal duplicat.
pipe	Crea un dispositiu tipus pipe llesta per ser utilitzada per comunicar processos
lseek	Mou la posició actual de L/E (per fitxers de dades)

Les crides open, read i write poden bloquejar el procés

# Open

---

- Per tant, com s'associa un nom a un dispositiu virtual?
- `fd = open (nom, access_mode[ ,permission_flags] ) ;`
  - Vincula nom de fitxer a dispositiu virtual (descriptor de fitxer o canal)
    - ▶ A més, permet fer un sol cop les comprovacions de proteccions. Un cop verificat, ja es pot executar *read/write* múltiples cops però ja no es torna a comprovar
  - Se li passa el **nom** del dispositiu i retorna el dispositiu virtual (*fd: file descriptor* o canal) a través del qual es podran realitzar les operacions de lectura/escriptura ,etc
  - El **access\_mode** indica el tipus d'accés. Sempre ha d'anar un d'aquests tres com a mínim :
    - ▶ O\_RDONLY (lectura)
    - ▶ O\_WRONLY (escriptura)
    - ▶ O\_RDWR (lectura i escriptura)

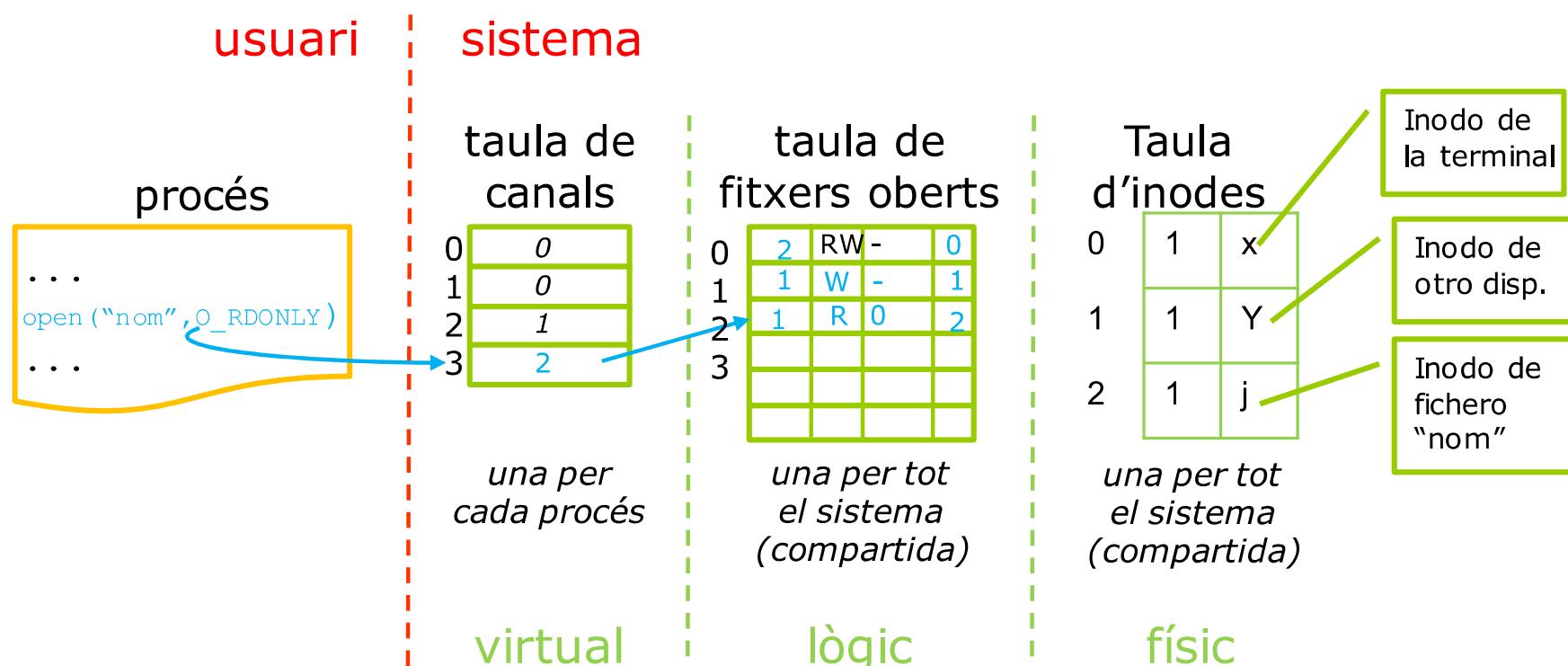
# Open: Creació

---

- Els fitxers especials han d'existir abans de poder accedir
- S'han d'especificar els **permission\_flags**
  - És una OR (|) de: S\_IRWXU, S\_IRUSR, S\_IWUSR, etc
- Els fitxers de dades es poden crear a la vegada que fem l'open:
  - En aquest cas hem d'afegir el flag O\_CREAT (amb una OR de bits) a l'accés\_mode
- No hi ha una crida a sistema per eliminar dades d'un fitxer parcialment, només les podem esborrar totes. Si volem truncar en contingut d'un fitxer afegirem el flag O\_TRUNC a l'accés\_mode
  - Ex1: open("X",O\_RDWR|O\_CREAT, S\_IRUSR|S\_IWUSR) → Si el fitxer no existia el crea, si existia no té efecte
  - Ex2: open("X",O\_RDWR|O\_CREAT|O\_TRUNC, S\_IRWXU) → Si el fitxer no existia el crea, si existia s'alliberen les seves dades y es posa el tamany a 0 bytes.

# Open: Estructures de dades

- Open (cont): Efecte sobre les estructures de dades internes del sistema
  - Ocupa un canal lliure de la TC. **Sempre serà el primer disponible**
  - **Ocupa una nova entrada de la TFO: Posició L/E=0**
  - Associa aquestes estructures al DD corresponent (major del nom simbòlic). Pot passar que diferents entrades de la TFO apuntin al mateix DD



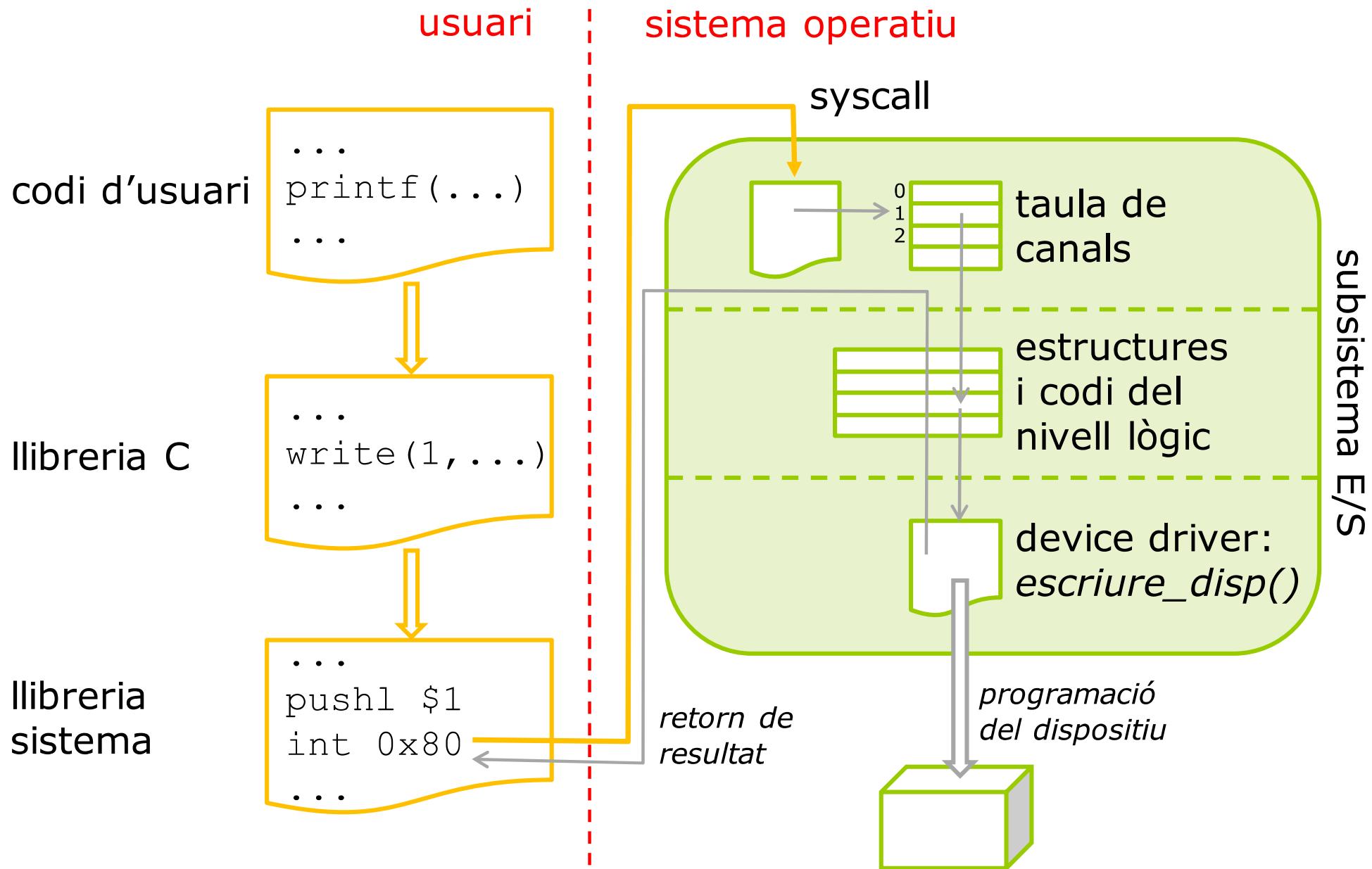
# Read

- `n = read(fd, buffer, count);` Num. Bytes a llegir
- Demana la lectura de *count* caràcters del dispositiu associat al canal *fd*
  - ▶ Si n'hi ha *count*, o més, disponibles, en llegirà *count*
  - ▶ Si n'hi ha menys, llegirà els que hi hagi
  - ▶ Si no n'hi ha cap, dependrà del funcionament del dispositiu
    - Es pot bloquejar, esperant a que n'hi hagi
    - Pot retornar amb cap caràcter llegit
  - ▶ Quan *EOF*, l'operació retorna sense cap caràcter llegit
    - El significat d'*EOF* depèn del dispositiu
- Retorna *n*, el nombre de caràcters llegits
- La posició de l/e s'avança **automàticament (ho fa el kernel a la rutina del read)** tantes posicions com bytes llegits
  - ▶ Posicio\_l/e=posicio\_l/e+num\_bytes\_llegits

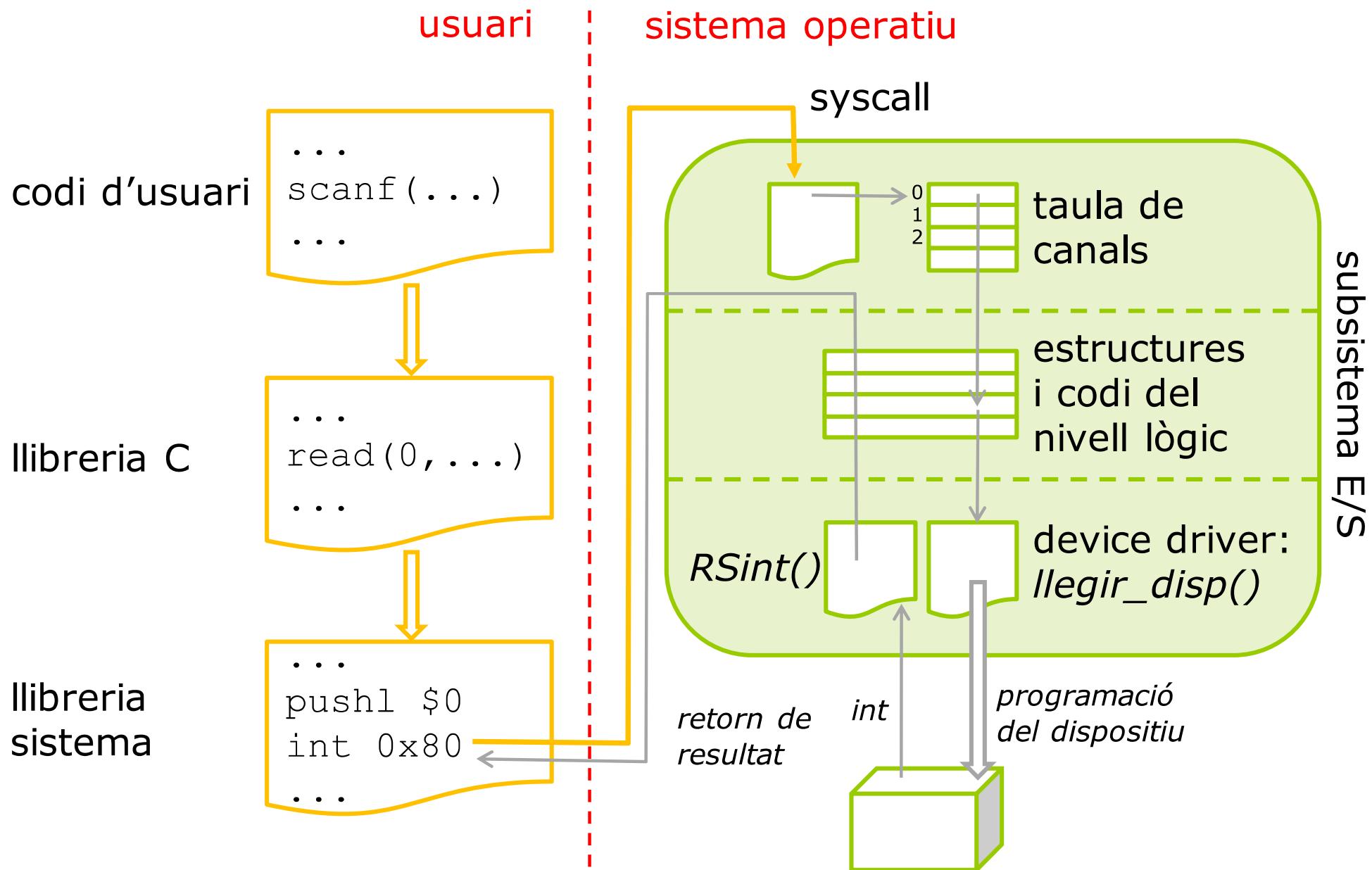
# Write

- `n = write(fd, buffer, count);`
  - Demana l'escriptura de *count* caràcters al dispositiu associat al canal *fd*
    - ▶ Si hi ha espai per *count*, o més, n'escriurà *count*
    - ▶ Si n'hi ha menys, escriurà els que hi càpiguen
    - ▶ Si no hi ha espai, dependrà del funcionament del dispositiu
      - Es pot bloquejar, esperant a que hi hagi espai
      - Pot retornar amb cap caràcter escrit
  - Retorna *n*, el nombre de caràcters escrits
  - La posició de l/e s'avança **automàticament (ho fa el kernel a la rutina del write)** tantes posicions com bytes escrits
  - Posició\_l/e=posició\_l/e + num\_bytes\_escrits

# Exemple: Escriptura a un dispositiu



# Exemple: Lectura d'un dispositiu



# Dup/dup2/close



□ `newfd = dup(fd);`

- Duplica un canal
- Ocupa el primer canal lliure, que ara contindrà una còpia de la informació del canal original *fd*
- Retorna *newfd*, l'identificador del nou canal

□ `newfd = dup2(fd, newfd);`

- Igual que *dup*, però el canal duplicat és forçosament *newfd*
- Si *newfd* era un canal vàlid, prèviament es tanca

□ `close(fd);`

- Allibera el canal *fd* i les estructures associades dels nivells inferiors
- Cal tenir en compte que diferents canals poden apuntar a la mateixa entrada de la TFO (p.ex. si *dup*), per tant, tancar un canal significaria decrementar -1 el comptador de referències a aquella entrada
- El mateix passa amb els apuntadors a la llista d'inodes



# pipe



- `pipe(fd_vector); // Dispositiu de comunicacions FIFO`
  - ▶ Crea una *pipe sense nom*. Retorna 2 canals, un de lectura i un altre d'escriptura
  - ▶ No utilitza cap nom del VFS (*pipe sense nom*) i, per tant, no executa la crida al sistema *open*
  - ▶ Només podrà ser usada per comunicar el procés que la crea i qualsevol descendent (directe o indirecte) d'aquest (pq heretaran els canals)
- Per crear una *pipe amb nom*, cal fer: *mknod + open*
- Internament: També crea 2 entrades a la taula de fitxers oberts (un de lectura i un d'escriptura) i una entrada temporal a la taula d'inodes.

# pipe

---

- Utilització
  - Dispositiu per comunicar processos
  - És bidireccional però , idealment cada procés l'utilitza en una única direcció, es aquest cas, el kernel gestiona la sincronització
- Dispositiu bloquejant:
  - ▶ Lectura: Es bloqueja el procés fins que hi hagi dades a la *pipe*, tot i que no necessàriament la quantitat de dades demanades.
    - Quan no hi ha cap procés que pugui escriure i *pipe* buida, provoca EOF (*return==0*) → *Si hi ha processos bloquejats, es desbloquejen*
  - ▶ Escriptura: El procés escriu, a no ser que la *pipe* estigui plena. En aquest cas es bloqueja fins que es buidi.
    - Quan no hi ha cap procés que pugui llegir el procés rep una excepció del tipus *SIGPIPE* → *Si hi ha processos bloquejats, es desbloquejen*
  - ▶ Cal tancar els canals que no s'hagin de fer servir! Altrament bloqueig
- Estructures de dades
  - Es creen dues entrades a la taula de canals (R/W)
  - Es creen dues entrades a la Taula FO (R/W)
  - Es crea una entrada temporal a la Taula d'Inodes

# Iseek

---

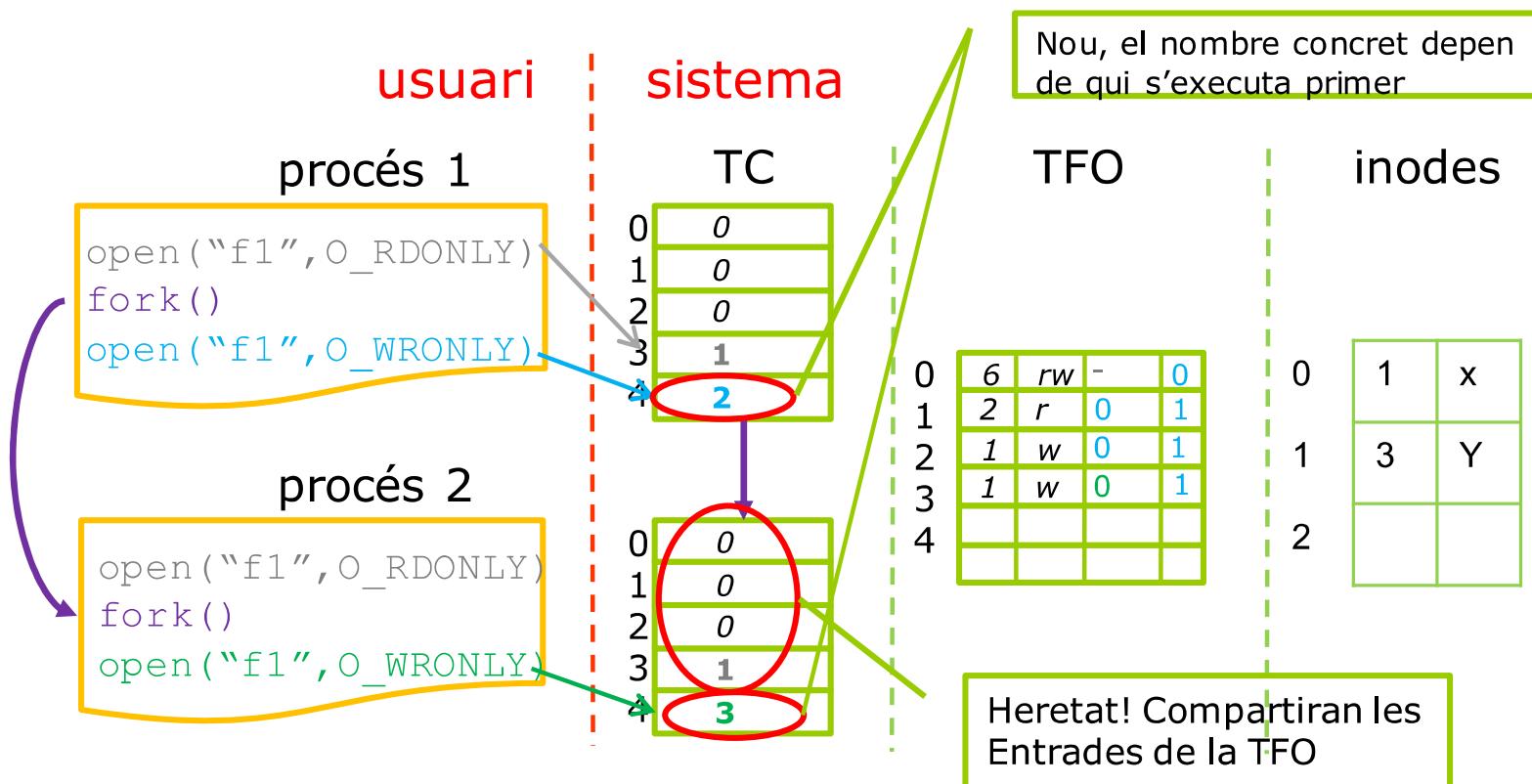
- La posició de l/e es modifica manualment per l'usuari. Permet fer accessos directes a posicions concretes de fitxers de dades (o fitxers de dispositius que ofereixin accés seqüencial)
  - S'inicialitza a 0 a l' open
  - S' incrementa automàticament al fer read/write
  - El podem moure manualment amb la crida lseek
- nova\_posicio=lseek(fd, desplaçament, relatiu\_a)
  - SEEK\_SET: posicio\_l/e=desplaçament
  - SEEK\_CUR: posicio\_l/e=posicio\_l/e+desplaçament
  - SEEK\_END: posicio\_l/e=file\_size+desplaçament
  - “desplaçament” pot ser negatiu

# E/S i execució concurrent (1)



## □ E/S i *fork*

- El procés fill hereta una **còpia** de la taula de canals del pare
  - ▶ Totes les entrades obertes apunten al mateix lloc de la TFO
- Permet **compartir** l'accés als dispositius oberts abans del *fork*
- Els següents *open* ja seran independents

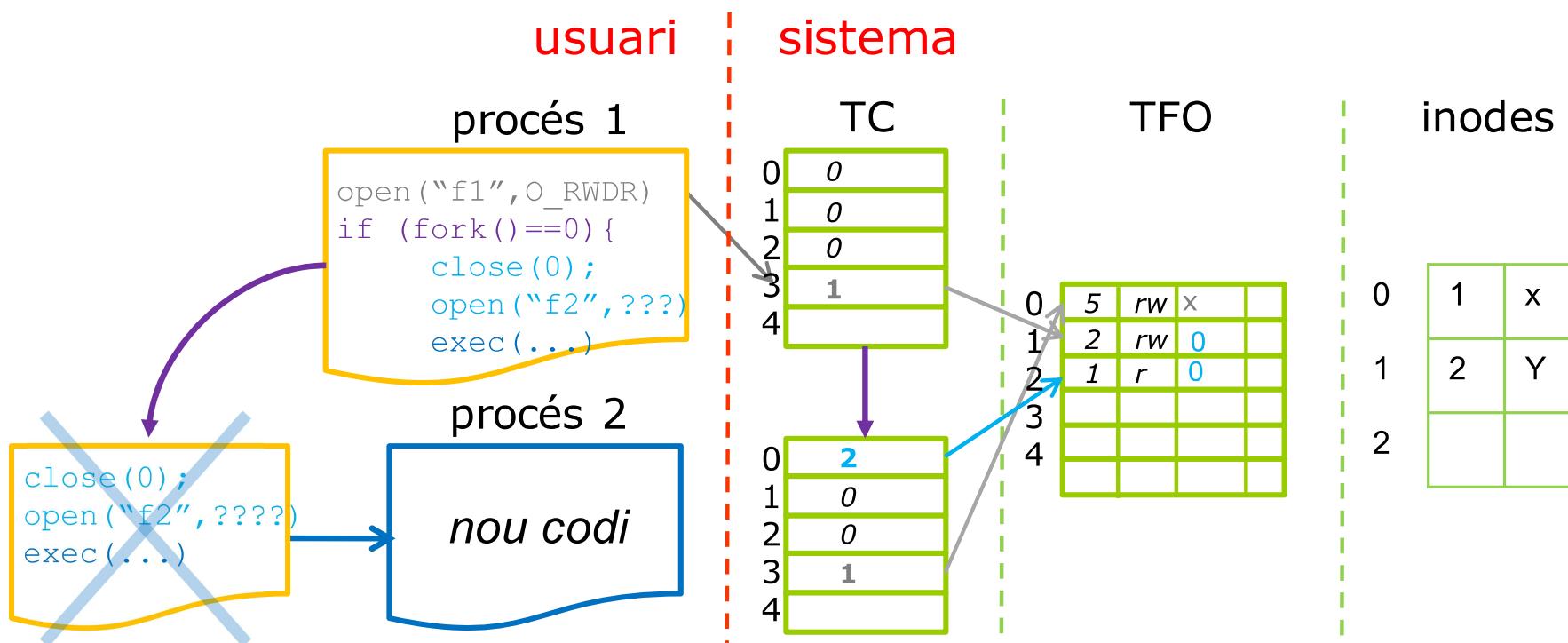


# E/S i execució concurrent (2)



## □ E/S i exec

- La nova imatge **manté** les estructures internes d'E/S del procés
- El fet d'executar *fork+exec* permet realitzar redireccions abans de canviar la imatge



# E/S i execució concurrent (3)



- Si un procés està bloquejat en una operació d'E/S i l'operació és interrompuda per un signal podem tenir dos comportaments diferents:
  - Després de tractar el signal, l'operació interrompuda es reinicia (el procés continuarà bloquejat a l'operació)
  - Després de tractar el signal, l'operació retorna error i la `errno` pren per valor `EINTR`
- El comportament depèn de:
  - La gestió associada al signal:
    - ▶ si flag `SA_RESTART` a la `sigaction` → es reinicia l'operació
    - ▶ en cas contrari → l'operació retorna error
  - L'operació que s'està fent (per exemple, mai es reinicien operacions sobre sockets, operacions per a esperar signals, etc.)
- Exemple de com protegir la crida al sistema segons el model de comportament:

```
while ( (n = read(...)) == -1 && errno == EINTR );
```

# Gestió de les característiques dels dispositius

---



- Encara que les crides a sistema son uniformes, els dispositius no ho són
- Existeixen crides a sistema per modificar característiques específiques dels dispositius lògics i virtuals.
- Dispositiu lògic
  - `ioctl(fd, cmd [, ptr]);`
- Dispositiu virtual
  - `fcntl(fd, cmd, [, args]);`
- Els paràmetres són molt genèrics per oferir flexibilitat



# Característiques terminal

- Terminal
  - El sistema manté, per cada terminal, un buffer intern per guardar els caràcters teclejats. Això permet esborrar caràcters abans de ser tractats
  - Posix defineix una funcionalitat particular a certs caràcters especials:
    - ▶ ^H: esborrar un caràcter
    - ▶ ^U: esborrar una línia
    - ▶ ^D: EOF o final de fitxer (final de l'entrada)
  - Pot tenir també un buffer d'escriptura per cada terminal i implementar certes funcions especials, del tipus “pujar N línies”, “avançar esquerra”
  - Cada controlador pot implementar el terminal tan complicat com vulgui. P.ex. que es pugui modificar text del mig de la línia
  - Canonical/No canonical: Pre-processat o no dels caràcters abans d'enviar-los al procés (afecta a la lectura)



# Característiques terminal

- Terminal: Funcionament (canonical ja que és el cas per defecte)
  - Lectura
    - ▶ **El buffer guarda caràcters fins que es pitja el CR**
    - ▶ Si hi ha procés bloquejat esperant caràcters, li dona els que pugui
    - ▶ Altrament ho guarda; quan un procés en demana li dona directament
    - ▶ ^D provoca que la lectura actual acabi amb els caràcters que hi hagi en aquell moment, encara que no n'hi hagi cap:
      - ▶ 

```
while ( (n=read(0, &car, 1)) > 0 )  
    write(1, &car, 1);
```
      - ▶ Això, per convenció, s'interpreta com a final de fitxer (EOF)
  - Escriptura
    - ▶ Escriu un bloc de caràcters
      - Pot esperar que s'escrigui el CR per ser visualitzat per pantalla
    - ▶ El procés no es bloqueja
  - Aquest comportament bloquejant pot ser modificat mitjançant crides al sistema que canvién el comportament dels dispositius



# Característiques pipes

## □ Pipe

### □ Lectura

- ▶ Si hi ha dades, agafa les que necessita (o les que hi hagi)
- ▶ Si no hi ha dades, es queda bloquejat fins que n'hi hagi alguna
- ▶ Si la *pipe* està buida i no hi ha cap possible escriptor (tots els canals d'escriptura de la *pipe* tancats), el procés lector rep *EOF*
  - Cal tancar sempre els canals que no es facin servir

### □ Escripció

- ▶ Si hi ha espai a la *pipe*, escriu les que té (o les que pugui)
- ▶ Si la *pipe* està plena, el procés es bloqueja
- ▶ Si no hi ha cap possible lector, rep *SIGPIPE*

### □ Aquest comportament bloquejant pot ser modificat mitjançant crides al sistema que canvién el comportament dels dispositius (`fcntl`)



# Dispositius de xarxa

## □ Dispositius de xarxa

- Tot i ser un dispositiu d'E/S, la xarxa té un comportament que no pot ser cobert amb les operacions genèriques d'accés als dispositius
- La gestió dels dispositius de xarxa es realitza de manera separada
  - ▶ P.ex: `/dev/eth0` no disposa ni de *inode* ni de *device driver*
- Disposen de crides al sistema específiques, i hi ha múltiples propostes
- Implementa els protocols de comunicació de xarxa (assignatura XC)



# Dispositius de xarxa

## □ **Socket:** Funcionament

- Funciona similar a una *pipe*, excepte que per comptes de dos canals (`fd[2]`) utilitza un únic canal per la lectura i l'escriptura del *socket*
- Es crea un *socket*, que es connecta al *socket* d'un altre procés en una altra màquina, connectada a la xarxa
  - ▶ Es pot demanar connexió a un *socket* remot
  - ▶ Es pot detectar si una altra es vol connectar al *socket* local
  - ▶ Un cop connectats, es poden enviar i rebre missatges
    - *read /write* o, més específicament, *send /recv*
- Es disposa de crides addicionals que permeten implementar servidors concurrents o, en general, aplicacions distribuïdes
- Habitualment, s'executen aplicacions amb un model client-servidor

# Dispositius de xarxa

## □ Socket: Exemple (*pseudo-codi*)

### □ Client

...

```
sfd = socket(...)
```

```
connect(sfd, ...)
```

```
write/read(sfd, ...)
```

### □ Server

...

```
sfd = socket(...)
```

```
bind(sfd, ...)
```

```
listen(sfd, ...)
```

```
nfd = accept(sfd, ...)
```

```
read/write(nfd, ...)
```



---

# **EXEMPLES**

# Accés byte a byte

- Lectura per l'entrada estàndard i escriptura per la sortida estàndard

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



- Observacions:

- Generalment, els canals estàndard ja estan oberts per defecte
- Es llegeix fins que no queden dades ( $n==0$ ), que depèn del tipus de dispositiu
- El total de crides a sistema depèn de quants bytes llegir
- Els canals 0 i 1 poden estar vinculats amb qualsevol dispositiu lògic que permeti llegir i escriure

```
#exemple1 →
#exemple1 <disp1 →
#exemple1 <disp1 >disp2 →
```

entrada=terminal, sortida=terminal  
entrada=disp1, sortida=terminal  
entrada=disp1, sortida=disp2

# Accés amb buffer d'usuari

- Igual, però llegim blocs de bytes (chars en aquest cas)

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

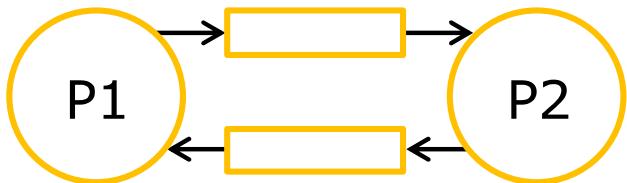


- Observacions:

- ▶ Compte! Cal escriure  $n$  bytes
  - Demanem SIZE bytes, ja que la variable ho permet, però és un màxim, pot ser que haguem llegit menys
- ▶ Quines diferències hi ha en quant a rendiment? Quantes crides al sistema farem?

# Intercanvi d'informació amb pipes

- Crear un esquema de procesos que sigui equivalent a aquest



- 2 pipes
- P1 envia per pipe1 i rep de pipe2
- P2 al revés

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

```
1. int pipe1[2], pipe2[2],pidp1,pidp2;  
2. pipe(pipe1);  
3. pipe(pipe2);  
4. pidp1=fork();  
5. if (pidp1==0) {  
6.     close(pipe1[0]);  
7.     close(pipe2[1]);  
8.     p1(pipe2[0],pipe1[1]);  
9.     exit(0);  
10. }  
11. close(pipe1[1]);  
12. close(pipe2[0]);  
13. pidp2=fork();  
14. if (pidp2==0) {  
15.     p2(pipe1[0],pipe2[1]);  
16.     exit(0);  
17. }  
18. close(pipe1[0]);close(pipe2[1]);  
19. while(waitpid(-1,null,0)>0);
```

# Accés aleatori i càlcul del tamany

- Què fa el següent fragment de codi?

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



*Trobareu aquest codi a : ejemplo1.c*

- Y aquest altre?

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



*Trobareu aquest codi a : exemplo2.c*

# pipes i bloquejos



```
int fd[2];
...
pipe(fd);
pid = fork();
if (pid == 0) {                                // Codi del fill
    while (read(0, &c, 1) > 0) {
        // Llegeix la dada, la processa i l'envia
        write(fd[1], &c, 1);
    }
}
else {                                         // Codi del pare
    while (read(fd[0], &c, 1) > 0) {
        // Rep la dada, la processa i la escriu
        write(1, &c, 1);
    }
}
...
```

Teniu disponible aquest codi a: `pipe_basic.c`

- Compte! El pare ha de tancar `fd[1]` si no es vol quedar bloquejat!

# Compartir el punter de l/e

- Què fa aquest fragment de codi?

```
...
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));
...
```



Teniu disponible aquest codi a: `exemple1.c`

# No compatir el punter de l/e

- Què fa aquest fragment de codi?

```
...
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));
...
```



Teniu disponible aquest codi a: `exemple2.c`

# Redirecció entrada i sortida std

- Què fa aquest fragment de codi?

```
...
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);
}
...
...
```



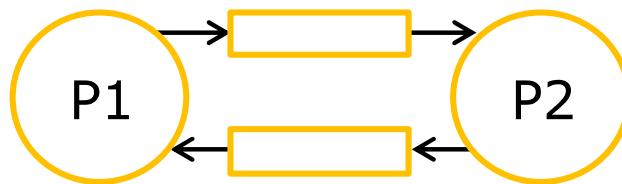
# Pipes y redirecció



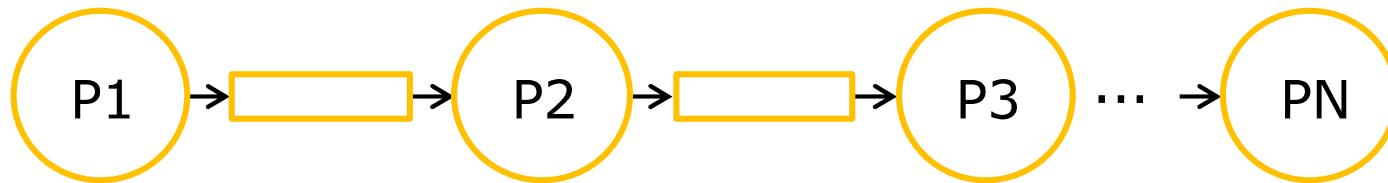
```
...
pipe(fd);
pid1 = fork();
if ( pid1 != 0 ) {                                // PARE
    pid2 = fork();
    if ( pid2 != 0 ) {                            // PARE
        close(fd[0]); close(fd[1]);
        while (1);
    }
    else {                                         // FILL 2
        close(0);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

# Exercicis a classe (1) (Col.P clase)

- Escriviu un fragment de codi que creï dos processos  $p_1$  i  $p_2$ , i els connecti mitjançant dues *pipes* pels canals d'E/S estàndard: en la primera  $p_1$  hi escriurà i  $p_2$  hi llegirà, en la segona  $p_2$  hi escriurà i  $p_1$  hi llegirà



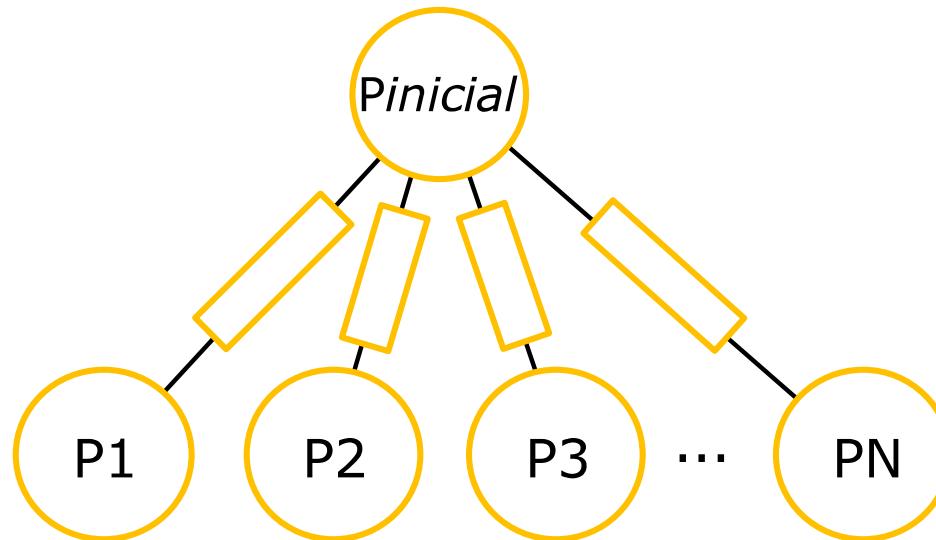
- Escriviu un fragment de codi que creï una seqüència de  $N$  processos en cadena: cada procés  $p_i$  crea un sol descendent  $p_{i+1}$ , fins arribar a  $p_N$ . Cada procés s'ha de comunicar en cadena amb l'ascendent i el descendent mitjançant una *pipe* pel canal d'E/S estàndard, de manera que el que escrigui el primer procés pugui ésser enviat per cadena fins al darrer procés



# Exercicis a classe (2)

---

- Escriviu un fragment de codi que creï  $N$  processos en seqüència: el procés inicial crea tots els descendents  $p_1$  fins arribar a  $p_N$ . Cada procés s'ha de comunicar amb el procés pare mitjançant una *pipe*; el pare ha de poder escriure a totes les *pipes* (a través dels canals  $3..N+2$ ) i els fills han de llegir de la *pipe* per la seva entrada estàndard.



---

# SISTEMA DE FITXERS

# Tasques del sistema de fitxers

---

- Responsabilitats del Sistema de fitxers respecte a gestionar fitxers i emmagatzemar dades
  - Organitzar els fitxers del sistema → espai de noms i esquema de directoris
  - Garantir l'accés correcte als fitxers (permisos d'accés)
  - Gestionar (assignar/alliberar) espai lliure/ocupat pels fitxers de dades
  - Trobar/emmagatzemar les dades dels fitxers de dades
- En qualsevol cas, tots els fitxers (de qualsevol tipus) tenen un nom que s'ha de emmagatzemar i gestionar. Els noms de fitxer s'organitzen en directoris

# Espai de noms: Directori

En Linux, tot això està a l'inode

- Directori: Estructura lògica que organitza els fitxers.
- És un fitxer especial (tipus directori) gestionat pel SO (els usuaris no el poden obrir i manipular)
- Permet enllaçar els noms dels fitxers amb els seus atributs

- atributs

- ▶ Tipus de fitxer  
directori(d)/block(b)/character(c)/pipe(p)/link(l),socket(s), de dades(-)
    - ▶ tamany
    - ▶ propietari
    - ▶ permisos
    - ▶ Dates de creació, modificació, ...
    - ▶ ...

- Ubicació al dispositiu de les seves dades (en cas de fitxers de dades)



# Linux: Visió d'usuari

---

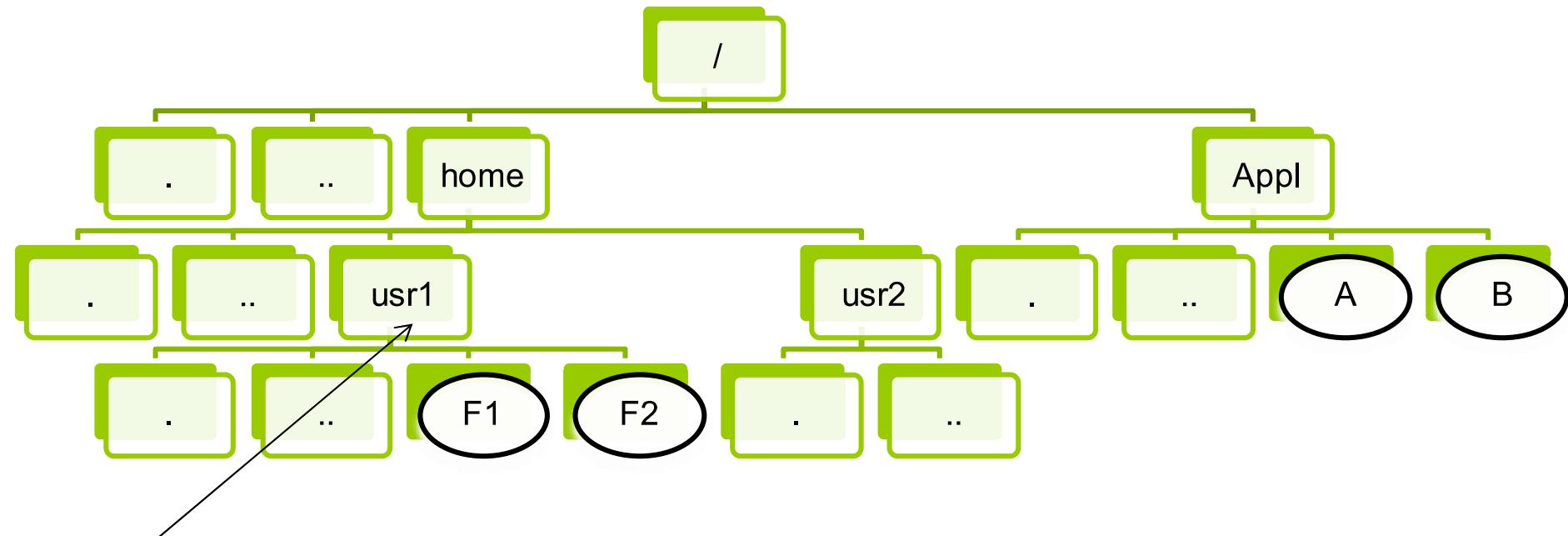


- Els directoris s'organitzen de forma jeràrquica (formen un graf)
- Permeten que l'usuari classifiqui les seves dades
- Els Sistemes de Fitxers organitza els dispositius d'emmagatzemament (cada un amb el seu esquema de directoris), en un espai de noms global amb un únic punt d'entrada
- El punt d'entrada és el directori “root” (Arrel), de nom “/”
- Un directori sempre té, com a mínim, 2 fitxers especials (de tipus directori)
  - . Referència al directori actual
  - .. Referència al directori pare

# Linux: Visió d'usuari



- Cada fitxer es pot referenciar de dues formes:
  - Nom absolut (únic): Camí des de l'arrel + nom
  - Nom relatiu: Camí des del directori de treball + nom



Si estem aquí, podem referir-nos a f2 com:

Nom relatiu: f2

Nom absolut: /home/usr1/F2

# Linux: Noms de fitxers



- Com el nom del fitxer està separat de la informació (inode), a Linux es permet que un inode sigui referenciat per més d'un nom de fitxer
- Hi han dos tipus de vincles entre nom de fitxer i inode
  - Hard-link:
    - El nom del fitxer referència directament al nombre d'inode on estan els atributs+info de dades
    - És el més habitual
    - A l'inode hi ha un nombre de referències que indica quants noms de fitxers apunten a aquest inode
      - És diferent al nombre de referències que hi ha a la taula d'inodes!!!
  - Soft-link
    - El nom del fitxer no apunta directament a la informació sinó que apunta a un inode que conté el nom del fitxer destí (path absolut o relatiu)
    - És un tipus de fitxer especial (l)
    - Es crea amb una comanda (ln) o crida a sistema (symlink)

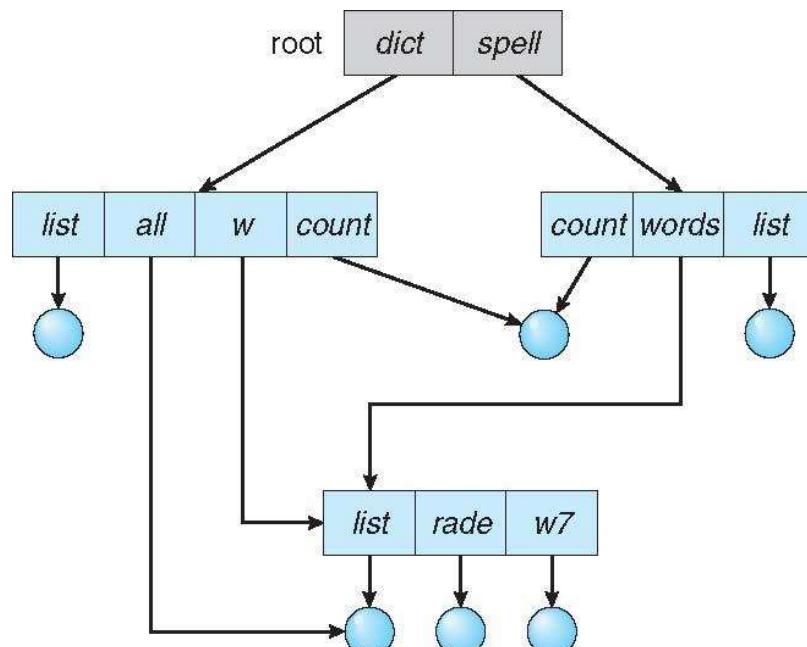
# Implementació jerarquia directoris



- La existència dels dos tipus de vincles influeix en la estructura del directori (que es un graf)
  - No es permeten cicles amb hard-links
  - Si se permeten cicles amb soft-links

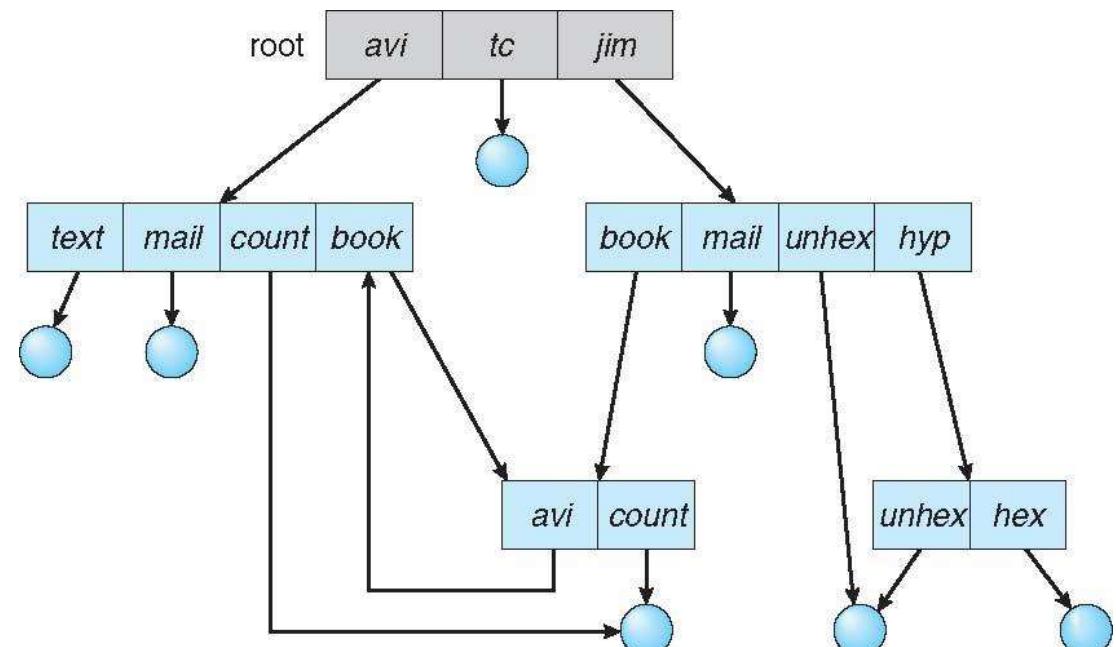
## Graf Acíclic

el SF verifica que no es creïn cicles



## Graf Cíclic

el SF ha de controlar els cicles infinitos



# Problemes dels directoris en grafs

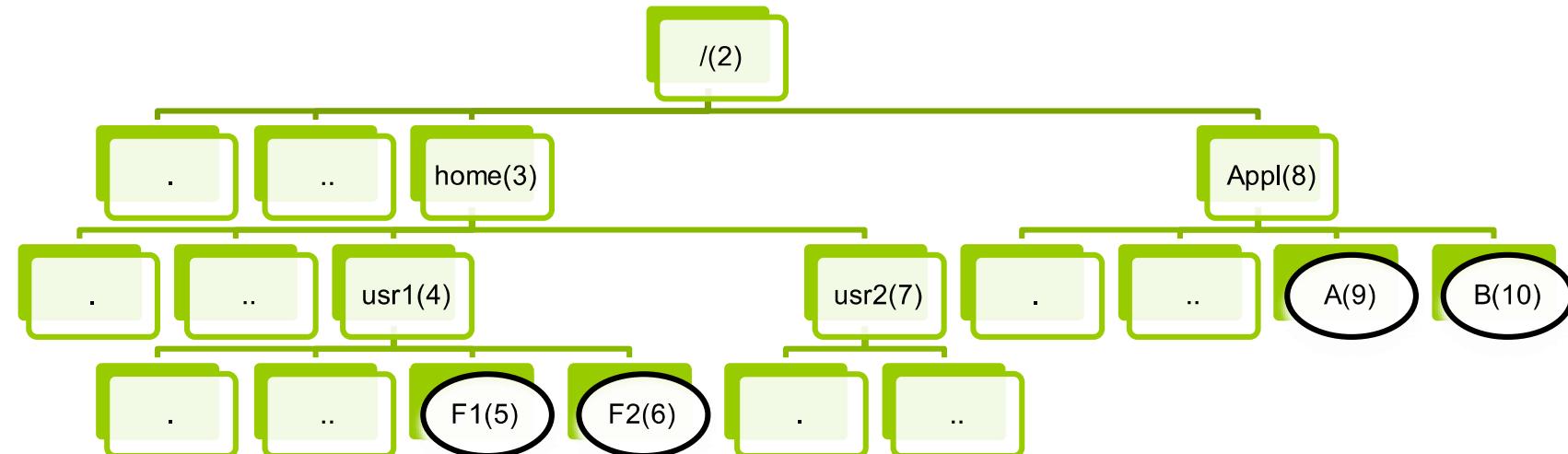
---

- Backups (copies de seguretat)
  - No fer copies del mateix fitxer dues vegades
- Eliminació d'un fitxer
  - Soft links
    - ▶ El sistema no comprova si hi han soft-links a un fitxer. A l'hora d'accendir es detectarà que no existeix.
  - Hard links
    - ▶ El fitxer només s'elimina quan el comptador de referències arriba a zero



# Visió interna de directoris

- El contingut d'un directori permet relacionar el nom del fitxer amb el seu número d'inode
- Pex: A l'exemple anterior, si assignem nombres d'inodes



Nom	inodo
.	2
..	2
home	3
Appl	8

Directori /

Cas especial

Nom	inodo
.	4
..	3
F1	5
F2	6

Directori /home/usr1

# Permisos d'accés a fitxers



- El SF permet assignar diferents permisos als fitxers
  - Es defineixen nivells d'accés i operacions que es poden fer
- Linux:
  - Nivells d'accés: Propietari, Grup d'usuaris, Resta d'usuaris
  - Operacions: Lectura (r) , Escriptura (w), Execució (x) → No confondre amb el mode d'accés!
  - Quan es defineix de forma numèrica, s'utilitza la base 8 (octal). Alguns valors típics

R	W	X	Valor numèric
1	1	1	7
1	1	0	6
1	0	0	4

- Revisar documentació laboratori

# Crides a sistema: espai de noms i permisos



Servicio	Llamada a sistema
Crear / eliminar enllaç a fitxer/soft-link	link / unlink/symlink
Cambiar permisos de un fitxer	chmod
Cambiar propietari/grup de un fitxer	chown / chgrp
Obtenir informació del Inodo	stat, Istat, fstat

- Existeixen més crides a sistema que ens permeten manipular:
  - Permisos
  - Vincles
  - Característiques
  - etc

# Unitats de treball del disc

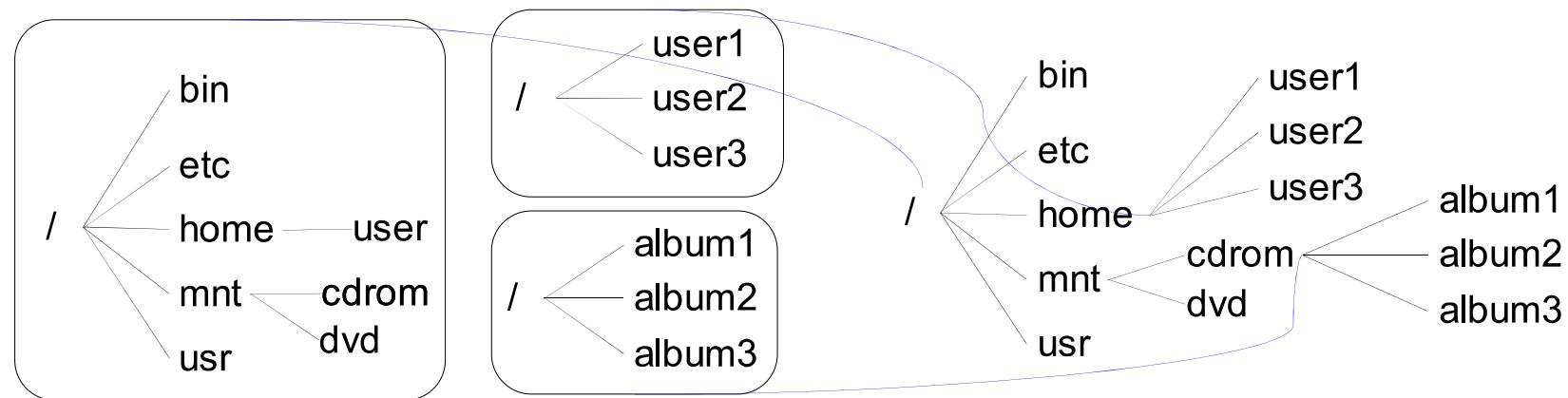
---

- Un dispositiu d'emmagatzemament està dividit en parts que anomenarem sectors
- La unitat d'assignament del SO és el bloc (1 bloc es correspon amb 1 o més sectors)
- Partició o volumen o sistema de fitxers
  - Conjunt de sectors **consecutius** amb un identificador únic (dispositiu lògic) i amb identitat propia. Són gestionats pel SO com a una entitat lògica independent
    - ▶ C:, D: (Windows); /dev/hda1, /dev/hda2 (UNIX)
  - Cada partició té la seva pròpia estructura de directoris i de fitxers independent de la resta de partitions



# Accés a particions

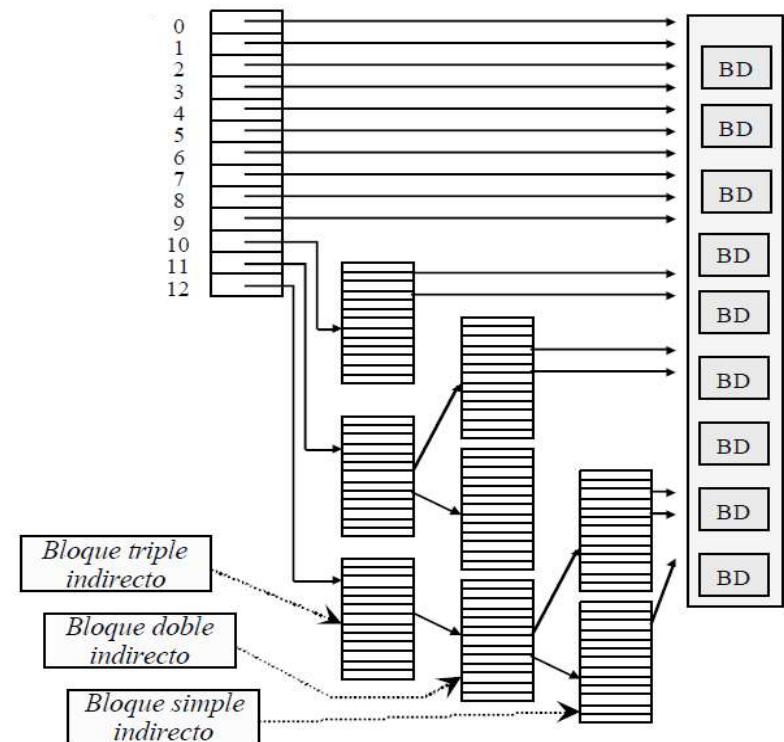
- Montar partició: acció de fer accessible una partició en el sistema de directoris global (només ho pot fer l'administrador de la màquina)
  - Linux: mount (per a montar), umount (per a desmontar)
    - ▶ # mount -t ext2 /dev/hda1 /home
    - ▶ # umount /dev/hda1



# Gestió de l'espai ocupat



- Per a cada fitxer, el SO ha de saber quins són els seus blocs.
  - Taula d'índexos: blocs asignats a un fitxer
  - Està a l'inode del fitxer: Quants indexos?
    - ▶ Esquema multinivel: índexos directes i indirectes
- Índexos a blocs de dades (1/4 Kb)
  - 10 índexos directes
    - ▶ ( 10 blocs = 10/40Kb )
  - 1 índice indirecte
    - ▶ ( 256/1024 blocs = 256Kb/4Mb )
  - 1 índice indirecte doble
    - ▶ ( 65K/1M blocs = 65Mb/4 Gb)
  - 1 índice triple indirecto
    - ▶ ( 16M/1G blocs = 16Gb/4 Tb)



# Gestió de l'espai lliure

---



- Gestió de l'espai lliure:
  - Llista de blocs lliures i llista d'inodes lliures
  - Quan fa falta un nou inode (creació d'un fitxer) o un nou bloc (per a fer créixer un fitxer) s'agafa el primer de la llista corresponent



# Metadades

- Metadades persistents: **guardades a disc**
  - Inodes i llista de blocs d'un fitxer
  - Directoris
  - Llista de blocs lliures
  - Llista d'inodes lliures
  - ... i altra informació necessària per al sistema de fitxers (quin és l'inode del directori arrel, tamany de bloc...)
- Superbloc: Bloc de la partició que conté les metadades del sistema de fitxers (redundància per a que sea tolerant a fallades)
- S'utilitza memòria per a guardar aquestes metadades mentre estan en us (i així estalviar accessos a disc)

# Metadades i memòria

---



- Zona de memòria per a guardar els últims inodes llegits
- Zona de memòria per a guardar el últims directoris llegits
- Buffer cache: zona de memòria per a guardar els últims blocs llegits
- Superbloc: es guarda també a memoria

---

# **RELACIÓ ENTRE CRIDES A SISTEMA / ESTRUCTURES DE DADES**

# Open



- Localitza l'inode del fitxer amb el que es vol treballar i el deixa a memòria
  - Quin és l'inode? Cal llegir el directori del fitxer. Dos situacions:
    - ▶ O el directori està a la cache de directoris i simplement s'accedeix
    - ▶ O el directori està a disc i s'ha de llegir (i es deixa a memòria).
      - Com sabem quins blocs llegir? → estan a l'inode
      - Quin es l'inode? → està al directori (repetim el procés)
  - Si localitza l'inode
    - Comprova si l'accés es correcte (els permisos d'accés són els adequats) → si no retorna erro
    - Si és un soft-link, llegeix el path del fitxer al que apunta i localitza l'inode corresponem (repetim el procés)
      - ▶ Si el path es curt, es troba al mateix inode
      - ▶ Si el path es llarg, es guarda en un bloc de dades separat
    - Modifica la taula de canals, la taula de fitxers oberts i la taula d'inodes
  - Si no localitza l'inode: error per accedir a un fitxer que no existeix
  - No accedeix a cap bloc de dades del fitxer objectiu



# Open (cont)

- Si utilitzem l'open per a crear un nou fitxer
  - S'ha de reservar i inicialitzar un nou inode
    - ▶ Actualizar llista d'inodes lliures al superbloc
    - ▶ Actualitzar directori a on es crea el fitxer
      - modificar el bloc de dades del directori
      - modificar l'inode amb la nova mida del directori



# Read

- Tenim a la taula d'inodes l'inode corresponent (i la informació sobre els blocs de dades)
- Tenim a la TFO el valor del punter de lectura/escritura (la posició del fitxer que toca accedir)
- Cal calcular els blocs involucrats:
  - Primer comprovem si estem al final del fitxer (valor del punter == mida del fitxer)
  - Després calculem els blocs
    - ▶ Dividint el valor del punter entre la mida de bloc obtenim l'índex del primer bloc a llegir
    - ▶ Amb el paràmetre mida de la syscall podem calcular quants blocs més cal llegir
- Si els blocs ja s'havien utilitzat abans els podem trobar a memòria, si no cal llegir-los de disc (i es deixen a memòria).



# Write

---

- La diferència amb el read es que si l'escriptura es fa al final del fitxer pot ser necessari afegir més blocs al fitxer. En aquest cas, caldrà modificar la llista de blocs lliures del superbloc i actualitzar l'inode del fitxer amb els nous blocs i la nova mida del fitxer



# Close

---

- Provoca l'actualització de l'inode amb els canvis que s'hagin fet (com a mínim la data d'últim accés)