

# Apunts SIOP a EPSEVG

Mariona Farré

## Índex:

<b>Tema 0: Resum de Coneixements Previs:</b>	<b>4</b>
0.2 Estructura de computadors:	4
0.2.2 Registres del processador:	4
0.2.3 Llenguatge màquina:	5
0.2.4 Entrada/Sortida:	5
0.2.5 Interrupcions:	5
0.3 Programació llenguatge C:	6
• Funció Main:	6
• Variables i sentències:	6
• Assignació:	6
• Crides a funcions:	6
• Salt condicional:	7
• For:	7
0.3.2 programació Modular :( ús de funcions)	7
• Paràmetres:	7
• Definició de funcions:	8
• Llibreries:	8
• Localització de variables locals i globals:	8
0.3.4 Construcció de nous tipus de dades:	9
• Vectors:	9
• Strings:	10
• Struct:	10
0.3.5 Punters i vectors:	10
• Punters:	10
• Punters i Funcions:	11
0.3.6 Entrada/Sortida amb rutines de llibreria:	11
<b>Tema 1: Introducció</b>	<b>12</b>
1.2.2 Que és un Sistema Operatiu	13
1.2.3 SO com a Gestor de Recursos	13
1.2.4 Nivells del SO	13
1.2.5. SO com a màquina virtual	13
1.3 Accés al Kernel/NUCLI del SO	14
• Les crides al sistema: (traps)	14
• Les interrupcions: (interrupcions hardware)	14
• Les excepcions: (interrupcions software)	14
1.3.3 Suport Hardware per implementar un SO fiable	14
1.3.4 Llibreries el llenguatge vs llibreries de sistema	15
• Llibreria del sistema:	15
• Llibreria del llenguatge:	15
<b>Tema 2: Gestió de Processos</b>	<b>16</b>
2.1.3 Propietats d'un procés:	17
2.1.4 Etapes en la vida d'un procés	18
2.1.5 Estats d'un procés	18
2.1.6 Concurrencia i Paral·lelisme:	18
<b>Gestió de Processos: Crides al Sistema</b>	<b>19</b>

Crida fork:	19
2.2.2 Efectes SO	19
Crida Break:	20
2.2.3 Accions del SO davant del fork	21
2.3 Esperar l'acabament de processos	21
Crida wait:	21
Crida waitpid:	22
2.3.5 Diferències entre wait i waitpid	22
2.4 Canviar el codi en execució d'un procés	22
Crida exec:	23
2.5 Finalització de processos:	24
Crida exit:	25
2.5.2 Accions del SO davant d'un exit	25
2.6 Situacions possibles:	25
2.7 Identificació de processos	26
2.8 Introducció i terminologia de Signals	27
2.8.2 Enviament de signals	27
2.8.3 Comportament dels signals:	27
2.8.4 Terminologia de signals:	28
2.9 Tipus de Signals	28
2.10 Crides a sistema per la gestió de Signals	29
2.10.1 Crides a sistema per la generació de signals:	29
Crida kil:	29
Crida alarm:	29
2.10.2 Programació de la rutina d'atenció al signal	31
Crida sigaction:	31
2.10.3 Espera de signals	32
Crida pause:	32
2.11 Interacció dels signals amb altres Crides al sistema:	33
<b>Gestió de Processos: Kernel</b>	<b>34</b>
2.12 Gestió interna de Processos:	34
2.12.1 Process Control Block:	34
2.12.2 Estructures de gestió per organitzar processos	35
2.12.3 Algorismes de planificació:	35
2.12.4 Mecanismes de planificació:	35
2.12.5 Objectius i Mètriques de la planificació:	36
<b>Tema 3: Gestió de dispositius d'entrada i sortida</b>	<b>37</b>
3.1.1 Independència de dispositius	37
3.1.2 Mapeig dels dispositius sobre el sistema de fitxers: Virtual File System	39
3.1.3 Device Driver	39
3.2 Gestió de l'Entrada/Sortida a UNIX	40
3.2.1 Estructures de dades per la gestió de L'E/S	41
1) Taula d'i-nodes:	41
2) TFO: Taula de fitxers oberts	41
3) TDVO: Taula de dispositius virtuals oberts	41
3.3 Crides a Sistema relacionades amb l'Entrada i la Sortida	42
3.3.1 Associar Canals Virtuels a dispositius	42
Crida Open	42
Crida Close	44
3.4 Lectra i escriptura sobre fitxers o altres dispositius d'E/S	44
Crida Read	44
Crida Write	45

3.4.3 Característica de les crides read/write	45
3.5 Filtres i redirecció de canals	46
3.5.1 Filtres	46
3.5.2 Redirecció	46
3.6 dup i dup 2	46
Crida dup	46
Crida dup 2	47
Crida lseek	47
3.9 Pipes	48
3.9.3 Pipes amb nom	48
3.9.3 Pipes ordinàries	48
3.9.5 pipes de shell	48
Crida Pipe	49
Crida Close sobre pipe	49
3.9.8 Lectura i escriptura sobre pipes	49
3.10 Sockets	51
3.10.2 Servidor	51
1. Inicialització del socket.	51
2. Acceptar connexions	51
3. Comunicació amb el client.	52
4. Finalitzacions.	52
3.10.3 Client	52
3.11 Dispositius específics	53
Consola:	53
Operacions ioctl i fcntl:	53
Pipe:	53
Socket:	53
<b>Tema 4: Estructura del Sistema de fitxers: VFS</b>	<b>54</b>
4.1 El Disc	54
4.1.2 Estratègies de gestió del espai de disc:	55
4.2 Sistema de Fitxers de Unix: ext2	56
4.2.1 Inode	56
4.2.2 Fitxer ordinari	56
4.2.3 Directori	57
4.2.4 Dispositius	57
4.2.5 Links	58
4.3 Permisos Fitxers: Access Control Lists	58
4.4. The virtual File System (VFS)	59
4.4.1 Mounting a File Sistem	59
4.4.3 Open i el VFS	62
4.4.4. Read i el VFS	64
4.4.5 Write i el VFS	64
4.4.6 Esborrar un fitxer en el VFS	64
4.4.7 Consideracions finals	64
<b>Tema 5: Gestió de memòria:</b>	<b>65</b>
5.1 Conceptes:	65
5.1.2 Memòria física x memòria lògica	65
5.1.3 Espai d'adreces	65
5.1.5 Assignació de memòria	65
5.1.6 Protecció d'adrecces	66
5.1.7 Mida del espai lògic del procés	66
5.2 Serveis del SO relacionats per gestió de memòria	68

5.3 Gestió interna de la Memòria	69
5.3.1 Gestió de la memòria física	69
5.3.2 Paginació	70
5.3.3 Segmentació	72
5.3.4 MMU implementar un esquema basat en segmentació	72
5.3.6 Traducció d'adreces	72
5.3.7 Protecció d'adreces	73
5.3.8 Memòria Compartida	73
Threads:	73
5.4 Optimitzacions en la Gestió de la Memòria	74
5.4.1 Copy on Write COW	74

## Tema 0: Resum de Coneixements Previs:

### 0.2 Estructura de computadors:

Ordinador pot executar programes però necessita:

- programa a executar
- entendre cada instrucció
- memoritzar el valor de les variables a l'instant
- obtenir informació del teclat i mostrar-la en pantalla
- llegir/escriure informació del disc

el programa i memoritzar el valors de les variables estan a la memòria

processador executa instruccions

teclat i disc son dispositius d'entrada i sortida

- **Memòria principal:** emmagatzema programes formats per ordres, guarda dades que es obtenen com a resultat de l'execució
- **Unitat Central de procés CPU/processador:** on s'executa el programa (càlculs)
  - **Unitat de Procés:** circuit electrònic per l'execució
  - **Unitat de control:** sistema que governa a la unitat de procés, assegurant la seqüència temporal de les accions
- **Unitat d'entrada i sortida:** connecta l'ordinador amb l'usuari

aquestes unitats estan connectades un conjunt de busos:

- **bus de control:** transporta de senyals perdi control entre processador i resta d'unitats.
- **bus de dades :** transporta dades entre diferents unitats
- **bus d'adreces:** transporta adreces/punters des del processador a altres unitats

#### 0.2.2 Registres del processador:

Registres formen part de la unitat de procés. mecanisme més ràpid d'accés a les dades.

Les variables es copien sobre registres abans de fer operacions.

Registres amb nom i concret s'utilitzen per controlar instruccions d'execució:

- Registres apuntadors:
  - IP:Instruction Pointer o PC (Program Counter): punter a la següent instrucció a executar.
  - SP: Stack Pointer: punter a la pila d'execució del programa.
- Registres de propósito general: AX,BX,CX,DX. Contenen dades o adreces (punters). En SISAF → R0 ...R7, Coma flotant→ F0 ...F7.
- Flags: indiquen l'estat de la màquina:
  - OF: Overflow flag: desbordament després d'una operació aritmètica.
  - IF: Interrupt Flag: si una interrupció externa, com l'entrada de teclat, ha de ser processada o ignorada.
  - ZF: Zero Flag: si el resultat de la darrera operació aritmètica ha estat 0.
  - CF: Carry flag: conté el bit de carry, si n'hi ha, de la última operació.

### 0.2.3 Llenguatge màquina:

El circuit està format per 0 i 1 tota informació codificada en Bits. Programa amb llenguatges de programació més properes el llenguatge natural però després ha de traduir-se en bits.

el **Llenguatge màquina** són un conjunt d'instruccions que estan codificades en bits.  
el **Llenguatge assemblador** cada instrucció correspon a una instrucció en llenguatge màquina, però és més entenedor → s'assembla a llenguatge mare

Els llenguatges màquina i assemblador s'anomenen **Llenguatges de baix nivell**, per a programar, que s'anomenen **Llenguatges d'alt nivell**

### 0.2.4 Entrada/Sortida:

En l'ordinador es necessita un bloc E/S per comunicar-se amb l'exterior

Emmagatzema dades a la memòria RAM- Volàtil les dades desapareixen al reiniciar

Es necessiten dispositius externs d'emmagatzematge.

- Dispositius de bloc:  
informació es guarden blocs, llegir i escriure independentment ex.disc
- dispositius de caràcter:  
informació es transfereix amb un flux de caràcters sense cap estructura  
ex. impressores en línia

### 0.2.5 Interrupcions:

Una interrupció es una senyal asíncrona del hardware que necessita atenció immediata pot donar en qualsevol moment, el processador para.

ex. Quan l'usuari prem una tecla el processador rep la interrupció corresponent i llegeix la informació relacionada amb els eveniments

**mètode per enquesta:** Comprova constantment si hi ha un esdeveniment extern - malbaratament de CPU

Codi de tractament de cada interrupció: vector d'Interrupcions (IQR) amb diferents nivells de prioritat. Si és interromput ha de ser capaç de tornar el mateix punt → context execució: valor de tots els registres abans de la interrupció es produeix- canvi de context

## 0.3 Programació llenguatge C:

- **Funció Main:**

Funció on comença a l'execució del programa.

Aquest programa pot tenir arguments commandes unix, També accepta arguments que recullen paràmetres.

Ex: Paràmetres de la funció main:

```
int main(int argc, char *argv[]){
    int i;
    for(i=0;i<argc;i++) {
        printf("El argument %d és %s \n",i,argv[i]); /*argc indica el nombre d'arguments*/
    }
}
```

- **Variables i sentències:**

Declaració de variables:

```
int n ; // el compilador reserva memòria guardar un valor enter (int) en variable n.
float x ; // idem. per un valor real (float) x.
```

Declaració de sentències:

```
n=3; // les sentències especificuen les accions que ha de fer el processador.
//Rebre dades, modificar el valor de les variables i mostrar els resultats.
```

- **Assignació:**

Assignar un valor per exemple a una variable.

```
void main(){
    int n,m; /* Variables */
    float x,y; // Inicialment, les variables n, m, x y tenen un valor arbitrari.
    n=3; // Després d'aquesta sentència, n val 3, un valor enter.
    m=n+2; // El processador calcula expressió (n+2) i assigna el resultat (5) a m.
    x=2.5; // Un valor real.
    y=m*1.5; // Després d'aquesta sentència y valdrà 7.5.
}
```

- **Crides a funcions:**

Permet programació més estructurada, funcions poden ser definides per nosaltres o una libreria importada. cadascuna realitza una acció es pot fer servir més d'una vegada.

```
DemanaCarta(); // jugada DemanaCarta();
```

Una funció molt utilitzada és la funció printf: escriure missatges a la pantalla de text.

```
#include <stdio.h>
void main{
    int i;
    float x;
    i=3;
    x=3.5;
    printf("Hola. Vaig a escriure el valor de les variables. \n"); //\n indica salt de línia.
    printf("i=%d\n",i); //%d indica un valor enter. En aquest cas el valor d'i.
    printf("x=%f\n",x); //%f indica un valor real (float). En aquest cas el valor d'x.
}
```

- **Salt condicional:**

if Executen la primera part dins del if sinó s'executa la part de else

```
void main(){
    int decisió;
    decisió=decidirContinuar(); // Un valor a l'atzar entre 0 i 1.
    if (decisió==0){ // La condició es posa entre parèntesis
        printf("El jugador s'ha plantat \n"); // Si es compleix la condició s'executa això.
    }else{
        printf("Facin joc... \n"); // En cas contrari s'executa això.
    }
}
```

- **For:**

Repetició amb for/while, dins d'un bucle tantes vegades com la condició indiqui

```
for (i=12; i>0; i--){ // i=12 és la inicialització, aquí posem instruccions que
    //s'executaran una sola vegada al inici i>0 és la condició d'acabament del
    // bucle i-- instruccions que s'executaran a cada iteració del bucle
    printf("3 x %d = %d \n",i,i*3);
}
```

### 0.3.2 programació Modular :( ús de funcions)

- **Paràmetres:**

- **Paràmetres per passats per valor:** una funció depen del valor.

Són paràmetres només d'entrada a la funció i no poden ser modificats.

quadrat(10); // Retorna 10 elevat al quadrat

- **Paràmetres constants:** poden tenir un valor constant

```
int num,num2,m;
num = 5;
num2 = 3;
m=min(num,num2); // El mínim entre 5 i 3
```

- **Paràmetres passats per referència:** modificar el contingut d'una variable des de dins d'una funció, com a variable E/S. Passant l'adreça de la variable== un **punter a la variable**

```
int num,num2,m;
num = 5;
num2 = 3;
min(&m,num,num2); // El mínim entre 5 i 3 es guarda a m
Operador & permet indicar l'adreça de la variable m
```

- **Definició de funcions:**

Una funció es un tipus de dades que retorna un resultat seguit pel nom d'aquesta i una llista d'arguments entre (). Ex: declaració de la funció i la crida des del main:

```
int fact( int v ) {           // capçalera de la funció
    int r = 1, i = 0;         // inicialització de variables
    while ( v >= i ) {       // bucle
        r = r * i;
        i = i + 1;
    }
    return r;                // retorn de la funció
}
int main() {
    int r, valor = 50;      // inicialització
    r = fact (valor);       // crida a la funció
    printf("El factorial de %d és %d \n",valor,r);
}
```

Ex: una funció modifiqui dues variables? maxmin retorna el màxim i el mínim de tres enters que passen com a paràmetre, max i min han de ser modificades a dins la funció → passades per referencia **utilitzar punters**

```
maxmin (&maxim, &minim, a, b, c);
```

Passant l'adreça de maxim i minim i valors de a,b,c s'haurà de guardar els valor, calculats de maxim i mínim a les adreces que hem passat com a punters.

- **Llibreries:**

Llibreria de C una molt important a utilitzar serà:

```
# include <stdio.h>
```

- **Localització de variables locals i globals:**

La visibilitat d'una variable indica quins llocs del programa aquesta està activa.

- **Variable local:** defineixen al principi de la funció, memòria a l'ordinador i es destrueixen al sortir de la funció.
- **Variable global:** algunes vegades variable que sigui accessible des de totes les funcions d'un mateix programa.

Si s'utilitza altres fitxers conjunts dona error de compilació→ indicar que es una variable ja definida a un altre lloc (modificador extern)

Definir un fitxer .h amb totes les variables globals.

```
#include <elmeu.h> (abans de cada fitxer que les utilitza)
extern int variable_global;
```

#### 0.3.4 Construcció de nous tipus de dades:

Interessa crear tipus més complexes que (int,char,double..)

- **Vectors:**

Arrays/vectors de qualsevol tipus, emmagatzemar un grup d'elements del mateix tipus

```
#include <vector.h>
tipus nom [mida]      int v[50] // declara un vector anomenat v de 50 enters
Accedim mitjançant un índex va 0 → N -1 per un vector N elements
v[0];                  // per accedir al primer element del vector
a = v[2];              // assigna el tercer element del vector a la variable a
printf("el ultim element es %d \n",v[N-1]);
```

Poden ser inicialitzats en la mateixa declaració:

```
char vocal[5] = { 'a', 'e', 'i', 'o', 'u' };  float n_Bode[5] = { 0.4, 0.7, 1, 1.6, 2.8 };
```

Poden ser multidimensionals:

```
int matrix[25][2]
```

La mida de la matriu depèn del tipus de dades del vector (int,char,dou..) i la mida d'aquest => files x columnes.

Saber la mida del vector podem crear función **sizeof (nom\_vector)**

- **Strings:**

Cadena de caràcters. Utilitza un caràcter especial com a marca de final de la cadena **/0** o caràcter **NULL**. Vector de mida N pot guardar una cadena de N-1 caract, i l'últim caràcter estara posicio N-2 i **/0** a la casella N-1

**#include <string.h>**

```
char nom[60]; char nom[7] = "Montse";
```

Saber la mida de string cridar funció **strlen(nom\_string)**

```
printf("la mida del string nom és %d \n",strlen(nom));
```

- **Struct:**

Struct/tupla agrupa diferents tipus de dades dins d'una mateixa estructura/nom

```
struct nom { llista de declaracions };
struct planeta {
    struct co r, v, a; //altra tupla
    double masa; //enter
    char nom[10]; //string
    struct co {
        double x,y,z; //enter
    };
};
```

Accessible mitjançant el nom de registre seguit de punt i el nom del camp:

**struct planeta plato; // declaració de la tupla**

**plato.r.x = 1.0; // inicializo el camp x de l'estructura co de dins l'estructura planeta**

Convenient renombrar nous tipus de dades comprensió més senzilla funció **typedef**  
**typedef struct planeta PLANETA; // donem nom al nou tipus creat**

**PLANETA mercuri, venus, terra, mart; // variables utilitzant el nostre tipus**

### 0.3.5 Punters i vectors:

Cada variable es troba en una determinada posició de la memòria (la seva adreça)

Permet passar una variable a una funció per referència i modificar-la des de dins, mitjançant un punter a la variable la podem modificar indirectament des de qualsevol funció.

- **Punters:**

Declaració de punters:

```
tipus *nom; float *ptr_a_float; //un punter que apunta a una variable de tipus float
```

Manipular un punter utilitzar el seu nom (la seva propia adreça),

Accedir a la variable es fa amb el caràcter ( \* ) (valor que apunta el punter)

Existeix caràcter ( & ) adreça de la variable que apunta

```
void prova_punter( void ) {
    long edat;
    long *p;
    p=&edat; // fem que el punter p apunti a edat
    edat=50; // modifiquem el valor d'edat
```

```

printf("La edat es %d \n", edat );
*p=*p/2; //modificar el contingut de la variable p apunta estem modificant edat
printf("La edat es %d \n" ,edat);
}

```

El resultat serà: La edat es 50 La edat es 25

- Punters i Funcions:**

Com podem passar paràmetres que son modificats:

```

void incr (int *p){
    *p=*p+1;    //volem modificar la variable a la que el punter apunta per tant hem
}                      accedir al contingut del punter amb el *

int main(){
    int a=0;
    incr (&a);    //passem &a que és del tipus punter ja l'adreça de la variable
    printf("a val %d \n",a);
}

```

$*p \Rightarrow$  accedir al contingut de la variable la qual apunta al punter (que estem passant amb  $\&a$ ) i accedim al contingut de l'adreça de a (0), i la modifiquem a la funció fent  
 $*p(\&a) = [ *p(\&a)=0 ] +1 == 1$

### 0.3.6 Entrada/Sortida amb rutines de llibreria:

Comunicació amb l'usuari i el sistema operatiu a través de funcions de llibreria. stdio.h

- **printf:** donar format i enviar dades a la sortida

```
printf ("Aquest és un missatge que mostra un argument decimal %d \n",10);
```

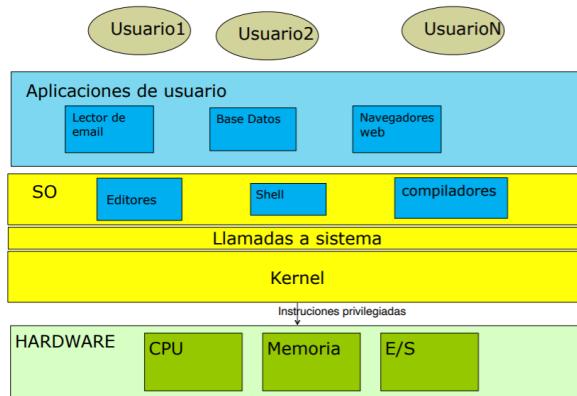
d	I d	f	e	c	s	p
enter decimal amb signe	enter decimal llarg	coma flotant en la forma (-)dddd.dddd	coma flotant en la forma (-)d.dddd e(+/-)ddd	un caràcter	cadena de caràcters acabada en ” (string)	punter

- **scanf :** llegir dades de l'entrada. arguments (i) on es guarden els valors  
`scanf("%d",&i);`

## Tema 1: Introducció

Sistema operatiu: tots els programes necessiten tasques iguals (teclat,memòria,compilar...) si cada programa necessites codificar cada detall del codi passaria:

- replicació del mateix codi
- difícil compartir recursos
- codi molt complicat i molt baix nivell (codi binari)
- poca portabilitat → codi molt dependent de la màquina, s'hauria d'anar escrivint
- poca estabilitat → errors en el codi usuari que podria destabilitzar el sistema



SO ho resol fent d'intermediari entre programadors i hardware, interfereix a l'entrada si es vol programar - llamadas a sistema

Programadors no accedeixin directament al hardware sinó indirectament mitjançant serveis (crides al sistema) oferts per SO.

Hardware: principals elements CPU,memòria,dispositius, opera amb llenguatge màquina, alguns necessiten accés privat/privilegiat

- Processador: executar instruccions de llenguatge màquina que componen el programa
- Memòria: emmagatzema les dades/codi i contingut al utilitzar el processador
- Dispositius E/S: permet interacció de la màquina amb l'usuari i altres màquines

Kernel: és l'únic accés directe al hardware, el nucli del SO i ofereix la llamadas a sistema

Aplicacions de Sistema: formem part de la distribució de SO i utilitzen les llamadas a sistema de SO, implementen serveis necessaris pel sistema (shell,login,compiladors..)

Aplicacions de l'usuari: pot utilitzar les llamadas a sistema, llibreries,programes i implementa serveis orientats a l'usuari (word, mail. Mozilla)

#### 1.2.2 Que és un Sistema Operatiu

- Gestor de recursos: controla /asigna, decideix qui els té i durant quant temps tots els recursos (CPU, mem, dispositius) per ser eficient, segur i just.
- programa de control: controla l'execució dels programes per evitar errors i la utilització incorrecte per part de l'usuari actura entre intermediari entre el programa de l'usuari i el hardware
- Actuar com a intermediari entre el hardware i els programes de l'usuari
- Ha de poder: executar programes de l'usuari i donar accés al sistema de forma fàcil oferir un ús eficient i segur
- com màquina virtual ofereix una abstracció més simple del HW i ofereix portabilitat

#### 1.2.3 SO com a Gestor de Recursos

- Gestió de processador: processador es un recurs compartit → encarregar gestionar la compartició i garantir que tots els processos rebin el processador periòdicament
- gestió memòria: memòria es un recurs compartit → garantir cada usuari només pugui accedir a la seva porció de memòria
- gestió dispositius E/S: gestió correcta i diferents usuaris no interfereixin
- gestió sistema de fitxers: donat un nom de fitxer localitzar on exactament es troben les dades associades aquest.
- protecció: protegir cada usuari dels accésos incorrectes (in/voluntaris) per altres usuaris

#### 1.2.4 Nivells del SO

Dividir el **software instal·lat** en:

- Nucli del sistema: SO: implementació dels serveis del SO i l'estructura de dades necessària. Accés directe al hardware. Ofereix una interfície → **crides a sistema**

(Programadors no accedeixin directament al hardware sinó indirectament mitjançant serveis (crides al sistema) oferts per SO.

- Aplicacions del sistema: faciliten el treball dels usuaris. Fan servir el SO (crides del sistema) per accedir al hardware

Interacció dels usuaris amb el sistema:

- Visió externa (usuari)  
SO ofereix programes de suport, les aplicacions del sistema/comandes. mode usuari
- Visió externa (programador)
  - accés a través de la interfaz de programació, serveis oferts pel SO/crides al sistema. executa el codi en mode usuari
- Visió interna (sistema/kernel)
  - codi i estructures de dades i codi del SO → codi del Kernel mode sistema

#### 1.2.5. SO com a màquina virtual

Crides al sistema → programador entra hardware indirectament => visió independent de la màquina (abstracció d'aquesta)

La visió de les crides al sistema → també com **màquina virtual** pot ser diferent **màquina real**, ha de ser molt més senzilla pe l'enteniment.

### 1.3 Accés al Kernel/NUCLI del SO

Entrar al Sistema Operatiu => traspassar la interfície d'accés, les crides del sistema i accedir al codi per sota d'aquestes crides => **Codi del Kernel i hardware de la màquina**

Esdeveniments que provoquen l'execució del codi propi del SO:

- Les crides al sistema: (traps)  
Provocades amb l'execució instrucció del llenguatge màquina.  
Un programa necesita el SO a través de la interfície que ofereix el Kernel amb les crides al Sistema. Síncrones i programades provocades per una instrucció del programa al ejecutarse.
- Les interrupcions: (interrupcions hardware)  
Provocades per dispositius per notificar algun esdeveniment.  
Tracta d'actuar en consecuencia a la notificació (teclat pitja tecla → dada a disposició que necessiti). Les rutines d'atenció que atenen a les interrupcions gestionen els dispositius hardware. Asíncrones i externes (a qualsevol moment, quan el dispositiu necessita atenció)
- Les excepcions: (interrupcions software)  
Provocades per una situació anòmala, detectada quan s'executa una instrucció del llenguatge màquina.  
(Divisió per zero, violació de segment → intentar accedir a una zona de la memòria que no li pertany) La rutina de tractament intenta posar remei a la situació re-executant la instrucció, sinó abortar l'execució del programa. Asíncrones i programàtiques  
(provocades per una instrucció i normalment involuntàries)

El tractament pels tres tipus es el mateix i es gestiona a través del **Vector d'interrupcions**:

- Salvar l'estat del programa que estava executant-se, no modificar el context/dades
- Pas de paràmetres a través de registres/pila (salvar-los)
- S'identifica el servei -> mitjançant una consulta al vector determinar quina es la **rutina d'atenció**
- Invocar i executar el servei: es passa el control a la rutina d'atenció que és pròpia del SO, no forma part del programa d'execució
- Quan la rutina finalitza retorna el resultat (pila o registres)
- Restaura l'estat del programa

#### 1.3.3 Suport Hardware per implementar un SO fiable

A un programa pot aparèixer qualsevol instrucció en llenguatge màquina → un usuari pot programar directament instruccions amb accés als dispositius, gestió de memòria..

Si es permet l'execució SO no controla l'accés dels dispositius +no garanteix confidencialitat ni l'estabilitat del sistema

Proporcionar un mínim suport de:

- modes d'execució del processador
- mecanismes de canvi de mode d'execució
- partició del repertori d'instruccions del llenguatge màquina
- memòria protegida

SO en mode privilegiat (kernel mode) i la resta del codi mode NO privilegiat (mode usuari)

## Mecanismes de canvi de mode d'execució del processador:

Mecanisme que canvi el mode d'execució en els esdeveniments com:

- Execució d'un codi extern - execució en mode usuari
- Un programa invoca una instrucció (de llenguatge màquina) a una crida al sistema s'executa la rutina indicada en el vector d'interrupcions i torna a mode usuari
- Rep una interrupció o una excepció, guarda el mode actual (usuari/sistema) i passa a ser mode sistema i consulta el vector d'interrupcions quina es la rutina. Al acabar es restaura el mode prèviament guardat.

## Partició del repertori d'instruccions del llenguatge màquina:

- **Instruccions privilegiades:** instruccions amb tasques pròpies del SO o poden posar en compromís la fiabilitat. Només poden ser executades si processador està en mode sistema.  
Ex: d'accés al registres dels dispositius, modificació del vector d'interrupcions, aturada del processador
- **Instruccions ordinàries:** instruccions aritmètiques/salt/accés a memòria/de tram  
Poden ser executades en qualsevol mode

Sempre que el processador executi una instrucció verificarà el mode actual si ho permet sinó generarà una excepció del tipus **mode d'execució invàlid** (avortament del procés)

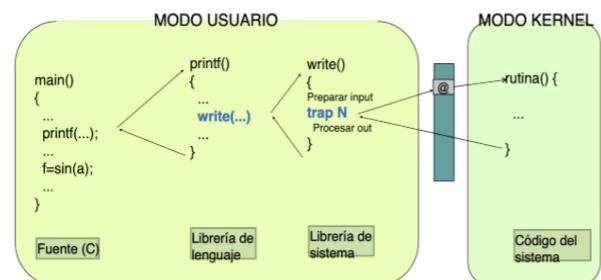
**Memòria protegida:** controlar que només es pugui accedir al tros de memòria assignat i només accedir al codi del kernel si esta mode privilegiat. Garantir que el vector d'interrupcions no pugui ser modificat, implementant un mecanisme de protecció de memòria.

**Interrupció de rellotge:** SO ha de garantir que tots els processos disposen periòdicament del processador per poder executar-se. SO aprofita les interrupcions del rellotge per garantir la presa de control i forçar els canvis i que ningú procés s'apropa indefinidament del processador

### 1.3.4 Llibreries en llenguatge vs llibreries de sistema

Crides al sistema ← execució instrucció en llenguatge màquina => problema al programar amb C no garanteix la potabilitat del codi

Solució **llibreria** (conjunt de routines d'ús comú ja implementades a disposició usuari)



#### • **Llibreria del sistema:**

rutines necessàries invocar crides al sistema programa en llenguatge C(/alt nivell). Funcions per accedir als serveis del kernel. Cada màquina disposa de la seva pròpia versió de llibreria de sistema, però la interfície(capçalera) de les funcions es la mateixa en totes les màquines.

programador - crides sistemes en C - llibreries sistema crea executable on es troba la interrupció software ← crea la crida al sistema

rutina **write** crida al sistema que s'encarrega de l'escriptura (dada port dispositiu)

#### • **Llibreria en llenguatge:**

rutines que proporcionen el llenguatge de programació.

crides Implementades mitjançant crides al sistema i amb nivell d'abstracció major.

La rutina **fprintf** rutina llibreria llenguatge C utilitzat la crida al sistema **write**

## Tema 2: Gestió de Processos

Programa: seqüència d'instruccions i dades emmagatzemades en un fitxer ordinari q es pot executar pel propietari, el grup i resta d'usuaris.

Fitxer executable conté codi màquina d'un programa però és estàtic.

Creació de fitxers executables:

- Shell script : comandes q entén el shell i pot executar-se (fàcil escriure/només comandes shell)
- Programa amb llenguatge d'alt nivell (creació d'un fitxer amb el codi font i després compilar-lo/tradueix codi font a codi objecte que enten la màquina) obté executable)

Procés: unitat d'assignació de recursos que proporciona el SO (memòria, dispositius, processador), variable durant la seva vida.

Aquest codi es carregat en la memòria i s'executa la primera instrucció passa a ser un programa en execució procés gestionat pel SO que li assigna els recursos:

- Un espai a la memòria, imatge, per contenir el codi i les dades
- Assignar-li un identificador de procés Pid(Process identifier)
- Assignar-li un estat del procés
- Donar-li accés als dispositius per poder interactuar amb l'entorn
- Donar-li accés als fitxers per guardar els resultats de forma permanent
- Llegir dades
- **CPU per executar-se**

Assignació a la cpu es un procés anomenat **thread d'execució o flux**. guarda info associada a un thread, punters a la pila SP (stack pointer), punter instrucció actual PC o els valors dels registres. La imatge en memòria tota informació necessària per l'execució d'un procés.

Conjunt informació associada dispositius assignats pel procés → **Taula de dispositius**

**Virtuels Oberts (TDVO)** Canals per realitzar operacions d'entrada.

Els processos no comparteixen recursos entre ells,memòria, descriptors de fitxers ni threads

### Threads:

Procés que s'executa amb un sol flux d'execució/thread i pot crear més.

Cada procés té una imatge del procés en memòria, el SO assigna els recursos.

Cada part del programa (instruccions) que es poden executar de manera independent se li pot associar un thread com a representació.

Threads d'un procés comparteixen els recursos assignats:

- Taula de Dispositius Virtuels (TDVO)
- Espai d'adreses. cada Thread té la seva pila del programa

### 2.1.3 Propietats d'un procés:

Un procés inclou no només el programa que executa sinó tota la informació necessària per diferenciar una execució del programa d'una altre. Les propietats:

- **La identitat:** procés defineix qui es i que pot fer. Consta de:
  - Process ID (PID): identificador únic pel procés. Identificar un procés dins del sistema. En crides al sistema identifica el procés que es vulgui.
  - Credencials: procés associat a un usuari tindrà accés a recursos en funció d'aquest. UNIX sistema multiusuari, admin assignen USER ID o GROUP ID per controlar l'accés. **ROOT** usuari privilegiat permesa qualsevol petició.
- **Entorn:** d'un procés que hereda del procés que l'ha creat(el procés pare).Dues llists:
  - Llista de paràmetres/arguments: paràmetres passats comandes quan s'executa el programa. El primer de la llista argv[0] - nom de l'executable
  - Llista de variables d'entorn: cada element es una parella format NOM=VALOR. existència d'aquestes variables fa que s'hereten pares/fills  
**PATH** llista directoris on buscar executables/**HOME** directori base del procés
- **Context:** informació defineix l'estat del procés en cada moment. foto de la seva execució. Guarda:
  - Informació de planificació: per poder suspendre/resumir el procés. contingut dels registres, mapeig de la memòria
  - Informació Contabilitat de recursos: recursos que utilitza i que consumeix durant l'execució. SO dona límit d'ús dels recursos.
  - Informació E/S: vector de punters relacionat amb la gestió E/S
  - Informació relacionada amb el sistema de fitxers: directoris per defecte i root per buscar fitxer quan rep petició obrir fitxer.
  - Taula de gestió de signals: processos envien events i reben del SO. Cada procés té una taula que defineix l'acció a prendre després de cada event.
  - Informació sobre la memòria virtual: contingut del seu espai d'adreses

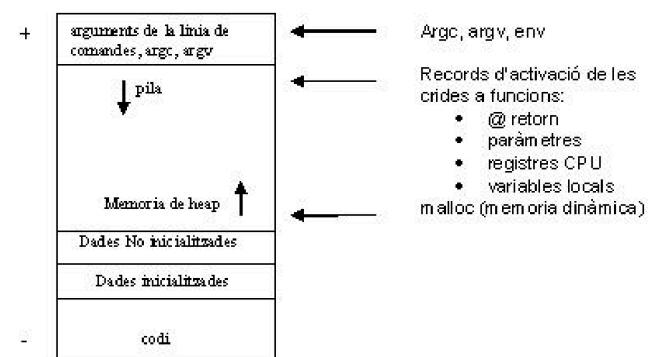
Després de ser carregat en memòria el programa és un bloc contingut de memòria a l'espai de les adreces del procés. La imatge del procés es forma per:

- Codi: instructions entén CPU de la màquina (fa copia del fitxer executable)
- Dades: conjunt dades que han de ser inicialitzades a l'inici. globals com locals
- Pila: serie de blocs lògics quan es crida una funció i son eliminats quan retornen (blocs d'activació) Regula l'execució del programa- qui punt s'executa

Les dades i pila poden créixer dinàmicament.

Cada nova funció crea un nou bloc d'activació informació de:

- retornar al punt on era (desp crida a la funció) guarda context de l'execució, l'adreça de retorn i els valors de tots els registres del processador
- informació necessària per executar la funció: paràmetres i variables de funció/locals
  - Variables global: estan en la memòria de heap igual memòria dinàmica
  - Variables locals: en la pila dins del bloc de la seva funció



El **context d'un procés** consisteix en el valor de tots els registres del processador (punter al codi, punter a la pila, registres d'ús general, paraules d'estat. . . ) en un moment donat. Guardat en **Process Control Block**

#### 2.1.4 Etapes en la vida d'un procés

- **Creació:**

Crea un nou procés al SO pot ser: procés buit/ determinats recursos assignats/còpia altre Cas de UNIX → crea un processos a partir de la còpia d'altres processos(pares) amb la crida al sistema `fork()`, la copia sera el (fill) → serà exactament igual però es podrà modificar (no crear BUCLES)

- **Carrega:**

Programa carrega l'espai lògic del procés. Dues maneres d'assignar els recursos:

- Estàtica: abans de l'execució: fitxer executable conté informació per defecte
- Dinàmica: realitza a petició i alliberament de recursos durant l'execució.  
Crida al sistema `exec()`

- **Execució:**

A partir d'ara se li dóna el control al flux d'execució.

- Assignació del PC (Program Counter): línia actual d'execució.
- Assignació del SP (Stack Pointer): punter a la pila d'execució.

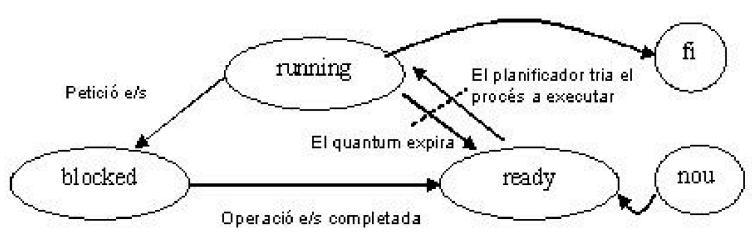
- **Destrucció:**

Finalització del procés. Per una finalització correcte- allibreri tots els recursos assignats al procés- el procés pare ha d'esperar l'acabament del seu fill

#### 2.1.5 Estats d'un procés

Un cop carregat el procés en memòria i preparat per l'execució → tenir en compte UNIX sistema multiprocés. No és l'únic que s'ha d'executar (esperar torn)

- running El procés s'està executant. Està fent ús de la CPU.
- ready està preparat per a ser executat però la CPU està ocupada.
- blocked passa a l'estat de blocked quan realitza una operació d'entrada sortida
- zombie ha acabat la seva execució però pare no ha fet wait i ocupa espai PCB



El planificador(scheduler) de processos és l'encarregat de gestionar la CPU

El canvi de context produeix quan el planificador treu un procés de l'estat de running i passa a executar un altre procés.

Un procés passa a l'estat blocked quan realitza una operació E/S son costosas de temps i el planificador no espera que acabi sinó passa a estat blocked i executa un altre.

#### 2.1.6 Concurrència i Paral·lelisme:

Concurrència- dos codis s'executen de forma simultània quan un processador és compartit alhora per varis fitxers// si hi ha més d'una unitat de procés- paral·lelisme.

## Gestió de Processos: Crides al Sistema

Utilitat d'executar diversos processos alhora → pot ser util per tasques que es necessiten fer mentre hi ha una E/S de dades alhora (navegador entrar en una web/buscar una altre cosa)

### Crida fork:

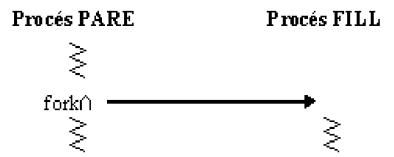
Unix crea un procés de rèplica d'un altre existent → utilitzar la crida **fork**

Un procés s'executa i amb la crida fork son dos processos

Crear un nou procés que serà **fill** del procés que ha realitzat la crida -**pare**.

El retorn de la crida serà diferent pel pare que pel fill (distinció del codi)

**Sintaxis:** `pid_t fork(void)`



Pare	> 0	PID II (important en cas contrari no el podrem saber)
Error	- 1	(ex: si s'ha superat el nombre màxim de processos que pot executar o executant-se en el sist.)
Fill	0	

### 2.2.2 Efectes SO

Fill és una còpia exacte del pare en: Dades /Pila /TDVO

El fil es diferencia del pare en: PID/Valor de retorn del fork

-TDVO: (taula de Dispositius Virtuals Oberts): ofereix independència de dispositius. Cada entrada de la taula es un canal associat que té un dispositiu d'entrada associat.

A cada procés és únic. Conté identificadors (punters) a la TFO

- TFO (Taula de Fitxers Oberts) gestionar dispositius E/S conté tota informació dels fitxers oberts en el sistema- mirar per la comanda `lsof`

Funcionament a alt nivell (codi)	Possible exercici: resultat variara ordre de scheduler	
<pre> main(){ int pid; pid = fork(); switch (pid){     case -1: /*tractament error*/     break;     case 0: /*codi del fill*/     break;     default: /*codi del pare*/ } } </pre>	<pre> main() { int ret; int a = 0; ret = fork(); if (ret == 0) {     a = 1;     printf("Soc el fill (a val %d)\n", a); } else {     printf("Soc el pare, a val %d\n", a); } </pre>	
Cas exemple pare//fill break necessari per evitar que el fill executi el seu codi més el del pare	Soc el fill (a val 1) Soc el pare (a val 0)	o en ordre invers Soc el pare (a val 0) Soc el fill (a val 1)

### Crida Break:

procés surti del actual bloc d'execució (codi entre {})S'utilitza per bucles i per la sentència switch

<pre>loop {     break;     //code... }  switch( variable ) {     case value:         /* code */         break;     case value:         /* code */         break;     default:         /* code */ } }</pre>	<p>Aquest cas el brak farà el codi dins //code no s'executi i saltar la primera instrucció després del bucle- acaba</p> <p>La sentència switch - break fa que sortir del bloc Un cop entrat dins d'un case no entrarem a l'altre- ni tan sols s'ha comprovat la condició.</p>
--	---

Exemple 1:

<pre>main(){ int pid; printf("Hola soc el pare\n"); pid = fork(); printf("Hola, soc el pare o el fill\n"); switch (pid){     case -1: /*tractament error*/ break;     case 0: /*codi del fill*/ break;     default: /*codi del pare*/ } }</pre>	<p>Instruccions abans del fork executades pel procés pare. (fill no creat) Instruccions després del fork seran executades pels dos processos (pare i fill)</p>
	<p>Resultat: Hola soc el pare → procés pare Hola soc el pare o el fill → resultat pare. o fill Hola soc el pare o el fill → resultat pare. o fill</p>

Exemple 2:

<pre>main(){ int pid; printf("Hola soc el pare\n"); pid = fork(); printf("Hola, soc el pare o el fill\n"); switch (pid){     case -1: /*tractament error*/ break;     case 0: /*codi del fill*/ break;     default: /*codi del pare*/ } printf("Ja he acabat"); }</pre>	<p>Instruccions després del fork hi ha una nova instrucció que els dos processos pare/fill hauran de treure per pantalla com el printf anterior</p>
	<p>Resultat: Hola soc el pare → procés pare Hola soc el pare o el fill → resultat pare. o fill Hola soc el pare o el fill → resultat pare. o fill Ja he acabat → resultat pare. o fill Ja he acabat → resultat pare. o fill</p>

Amb estructura If si es vol:

```
main(){
int pid;
pid = fork();
if (pid == -1) { /*tractament error*/ }
else if (pid == 0) { /*codi fill*/ }
else { /*codi pare*/ }
```

### 2.2.3 Accions del SO davant del fork

- Buscar una entrada lliure en el PCB i assignar-la al procés fill
- Assigna un PID únic al fill
- Copia codi/dades/pila privades del pare al procés fill
- Duplica la TDVO-Després fork, la TFO (unica SO) unic comparteixen->resta duplicat.
- Retorna al procés pare el PID del fill i al procés fill un 0

### 2.3 Esperar l'acabament de processos

Quan un procés finalitza la seva execució allibera gran part dels recursos utilitzats-no tots  
Alliberació de recursos completa procés esperi al seu acabament- crida [wait](#) o [waitpid](#)

Procés pare que no espera l'acabament- **procés zombie**- procés que mort però el seu pare no ha recollit el seu estat encara. Continua ocupant espai PCB (ProcessControlBlock) igualment que no usa la seva imatge- la entrada s'allibera quan el pare fa [wait](#)  
Important alliberar espai- PCB estructura de mida límitada sinó SO no permetrà crear nous processos- excedit nombre màxim de processadors

#### Crida wait:

Espera l'acabament del primer fill que acabi- es una crida bloquejant. es bloqueja l'execució del procés pare fins que un dels seus fills crea acabi- recull el seu estat o codi d'acabament dins de la variable status que passem e/s i retorna el PID del fill esperat

#### Sintaxi: [pid\\_t wait \(int\\* status\);](#)

- Status: punter variable entera té l'estat de finalització del fill.  
Per consultar-ho macro [wexitstatus \(status\)](#) que aplica la mascara  
[\(status >> 8\) & 0xFF](#)

15	8 7	0	8 bits més pes inicialitzats pel fill crida exit (codi)
+per		-pes	8 bits de menor pes inicialitzats pel SO (motiu)

Comunicar l'estat d'acabament(fill ha acabat l'execució correctament=0/codi error=programa)

Es pot passar qualsevol informació que es vulgui

Per recuperar la informació inicialitzada pel fill cal macro [WEXITSTATUS](#)

Retorna	-1 si hi ha un error >0 PID del procés que ha acabat	Exemple: <pre>int st ; pid = wait (&amp;s t ) ; printf (" El codi d ' acabament de l meu fill /* */ amb pid %d e s %d" , pid ,WEXITSATUS(status ) );</pre>
---------	---	---

- Si existeix algun zombie: retorna pid-status i el fill mor definitivament
- Si no existeixen zombies, es poden donar dues situacions
  - Si existeixen fills vius: bloqueig ns que algun d'ells acabi-retorna pid-status fill i aquest mor definitivament
  - Si no existeixen fills vius-retorna -1 (error)

Exemple wait:

Programa crea dos fills i mostra per pantalla el pid del fill que ha acabat primer (1 o 2)

<pre>main(){\nint pid1,pid2;\npid1=fork();\nif (pid1==0) /*codi fill 1*/ exit(0);\npid2=fork();\nif (pid2==0) /*codi fill 2*/ exit(0);}\nif (pid1 == wait(NULL)) printf( "El fill 1 ha acabat primer pid%d",pid1);\nelse printf("El fill 2 ha acabat primer pid %d",pid2);\nwait(NULL);\n}</pre>	<pre>main(){\nint pid;\nif (fork()==0) /*codi fill 1*/ exit(0);}\nif (fork()==0) /*codi fill 2*/ exit(0);}\npid = wait(NULL);\nprintf(\"El fill amb pid %d ha acabat primer\",pid);\nwait(NULL); /* espero l'altre fill */\n}</pre>
--	---

### Crida waitpid:

Crida waitpid permet esperar a un fill en concret o un grup o a qualsevol

**Sintaxi:pid\_t waitpid (pid\_t pid, int\*status, int options);**

- pid: pid del fill/s a esperar: (que pot tornar amb el fork d'abans)
  - -1 qualsevol fill
  - >0 un fill en concret
  - =0 qualsevol fill que pertany el mateix grup del propietari del procés
- status: punter a variable entera que tindrà l'estat de finalització del fill
- options: modificar el comportament de la comanda
  - WNOHANG: permet la crida waitpid sigui no bloquejant  
→ retorna 0 si no hi ha fills zombies i retorna immediatament

Exemple:

<code>waitpid(-1,NULL,WNOHANG);</code>	Esperar qualsevol fill però sense bloquejar el procés pare Pot ser que el fill estigui zombie- tornar a fer el wait en algun punt per evitar deixar fills zombies
--	--

### 2.3.5 Diferències entre wait i waitpid

- Wait bloqueja fins que un fill mori // waitpid opció que no passi
- wait espera al primer fill que es mor//waitpid depèn dels paràmetres que posem (fill concret)- té una crida que no es bloquejant

## 2.4 Canviar el codi en execució d'un procés

S'haurà de canviar el codi d'execució dels fills pq sinó tots els processos faran el mateix  
→ controlar el fil d'execució amb el retorn crida fork

Per poder canviar el codi d'execució es fa amb la crida `exec`

Permet: aprofitar programes existents + codi més fàcil de tractar amb estructura modular

### Crida exec:

És una família de crides que canvia el codi en execució del procés que fa la crida per l'executable (pare) que li passa com a paràmetre

**Sintaxi: exec{v |l,e| p}** //el procés passa a executar un altre programa

```
int execl (const char *path, const char *arg, ...)
int execlp (const char *path, const char *arg, ...)
int execle (const char *path, const char *arg , ..., char * const envp[])
int execv (const char *path, char *const argv[])
int execvp (const char *path, char *const argv[])
int execve (const char *path, char *const argv[], char *const envp[])
```

- l : list: paràmetre llista dels arguments del programa. últim argument NULL(indicllista)
- v : vector: paràmetre vector d'arguments del programa. Paral·lelisme amb arguments del main que rep la línia de comandes en vector de punters a caràcter modalitat **execv**
- e: environment/entorn: afegeix paràmetre les variables de l'entorn en comptes copiar de l'entorn del pare.
- p : path: nom del fitxer a executar, busca en tots els directoris de la variable d'entorn path- no cal posar path absolut i relatiu en el primer argument  
Convenient fer un executar d'una comanda **ls o ps**
- arg: llista d'arguments inclòs nom del programa- paràmetres que rebrà el main
- argv: vector strings conté arguments inclòs nom del programa- paràm. rebrà el main

Retorn:

-1	Cas d'error
X	ok no retorna res- sobreescriu la imatge del procés (no tornar codi on l'hem cridat)

Efectes SO: no creem un nou procés sinó produeix un canvi d'imatge del procés en memòria.

Programa vell és substituït pel nou i mai es retorna a ell- passa executar el nou

Nou codi, dades i pila/ mateixa TDVO i mai retorna (sinó existeix un error)

Paral·lelisme amb el main C:

void main(int argc, char *argv[]) argv[0] : nom del programa argv[1] : primer argument, argv[2] ... argv[N-1], argv[N] : NULL	execlp ("ls", "ls", "-l", "-a", null); char arg[4]; strcpy(arg[0],"ls"); strcpy(arg[1],"-l"); strcpy(arg[2],"-a"); arg[3]=null; execvp("ls",arg);
--	---

```
bash-3.2$ make exec_basic
gcc -o exec_basic exec_basic.c
bash-3.2$ ./exec_basic
Abans exec (pid=89525)
Ex1: Abans fork (pid=89525)
Ex1: Despres fork (pid=89525) soc el pare
Ex1: Despres fork (pid=89526) soc el fill
bash-3.2$ ls
!/bin/bash [running]
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main()
6 {
7 printf("Abans exec (pid=%d)\n", getpid());
8 execl("./ex1","ex1", (char *)NULL);
9 printf("Despres exec (pid=%d)\n", getpid());
10 }
```

```
exec_basic.c
"exec_basic.c" 10L, 200C
```

## Exemples:

<pre>main(){ printf( "Hola %d \n",getpid()); execl("./progvs","progvs",null); printf("Adeu %d \n",getpid()); }</pre>	<p>Exemple de execl Programa substitueix el codi fill pel programa progvs</p>
<pre>main(){ printf("Hola %d \n",getpid()); while(1){ if (fork() == 0) execlp("ls","ls",null); printf( "Volta %d \n",getpid()); } printf("Adeu %d \n",getpid()); }</pre>	<p>Exemple de execlp, un programa que crea fills, tots ells executem la comanda ls</p>
<pre>main(int argc, char *argv[]){ int i; if (argc != 2) panic("Parametres incorrectes"); for(i=1;i&lt;argc;i++){ if (fork()==0) { execlp(argv[i], argv[i], null); panic("Error en execlp"); }}}</pre>	<p>Exemple de execlp, un programa que crea un fill per executar cadascuna de les comandes passades com a arguments per la línia de comandes  prog ls ps La sortida sera l'execució d'un ls seguida per un ps</p>
<pre>main(int argc, char *argv[]){ int i; if (argc != 2) panic("Parametres incorrectes \n"); if (fork()==0) { execlp(argv[argc-1], argv[argc-1], null); panic("Error en execlp"); }else{ argv[argc-1]=NULL; execv(argv[0],&amp;argv[0]);}}</pre>	<p>Exemple de execv i execlp. El programa es crida passant-li com a arguments una llista de comandes (com al programa anterior) en aquest cas, el programa es recursiu, el programa crea un fill que executa la ultima comanda i es crida a ell mateix amb un argument de menys.</p>

## 2.5 Finalització de processos:

Un programa pot acabar de diverses maneres en condicions normals:

- Ja no hi han més sentències a executar en el main
- Executem la sentència return en el main
- Crida exit
- Aborta la seva execució
- Rep un signal que provoca la seva mort

Últims dos casos- programa perd el control i el SO s'encarrega(trap exit amb el codi acabat)

### Crida exit:

Crida al sistema exit finalitza l'execució del procés que la crida.

Sintaxi: `void exit (int status)`

Paràmetres:

- Status: codi de finalització del procés
  - 0 : tot ok
  - diferent 0: cas d'error

(SO passa aquesta informació al pare que ho recull en la crida `wait()`)

Retorn: no retorna

Exemple:	<code>exit(0); /* si tot ha anat bé */</code>
----------	---

### 2.5.2 Accions del SO davant d'un exit

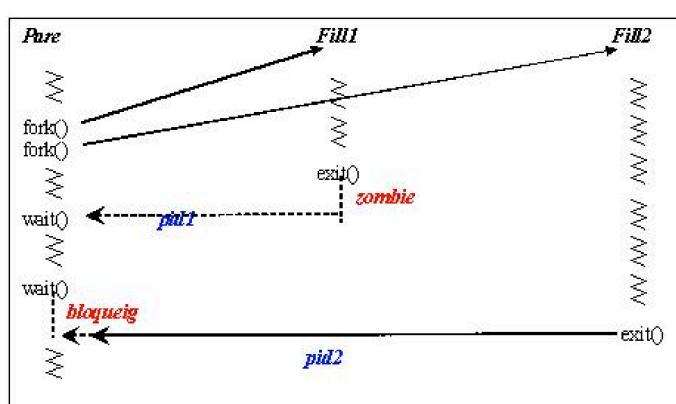
- Alliberar la memòria associada (codi/dades/pila)
- Tancar/Destruir la TDVO del procés
- Respecte la TFO, decrementar el nombre de links de cada entrada
  - destruir l'entrada si el `#referencies=0`
  - si `#referències =1`(un programa) es decrementa (-1) i es borra la ref del procés executat
- El SO li envia un signal al pare, `SIGCHLD` notificar fills ha mort.  
Per defecte aquest signal s'ignora, però es pot reprogramar.

### 2.6 Situacions possibles:

- Si mor el pare

Si el procés té fills vius, PPID=1, aquests seran adoptats per init(PID=1). Aquest s'encarrega de recollir el seu estat, així s'evita que el sistema es saturi de zombies (memòria ocupada innecessariament, compten com a processos en el sistema. . . )

- Si mor el fill
- Si pare esta en wait, obté l'status del II (els 8 bits menys significatius) i el II mor definitivament. Així el pare pot saber com ha acabat el seu fill.
- Si pare no esta en wait, el fill passa a ser zombie (defunc) fins que o bé el pare recull el seu estat o bé el pare mori i sigui adoptat per l'init.



## 2.7 Identificació de processos

Crear un procés en el SO que li assigna un identificador únic (de forma creixent) és el PID (Process identifier) i guarda el procés a la taula de PCB (Process Control Block)

Tot procés té un pare- si mor el procés serà adoptat pel sistema procés 1 o **INIT** (procés init va alliberant processos zombies que va agafant )

Crida que permet conèixer el pid del procés en EXECUCIÓ:      `int getpid();`

Crida que permet conèixer el pid del procés del PARE:      `int getppid();`

Unix un sistema multiusuari- administradors assignen user ID i grup ID per controlar accés.

Els usuaris que tenen permisos sobre fitxers i aquests son fitxers executables que passen a ser processos per accedir aquesta informació hi han les crides:

- `getuid()`      retorna el usuari real que está executant el proces
- `getgid()`      retorna el grup real
- `geteuid()`      retorna el usuari efectiu que esta executant el procés
- `geteguid`      retorna el grup efectiu

Usuari efectiu casi igual usuari real en determinats casos deixar un usuari pugui executar un fitxer executable i accedeixi a recursos com si fos un altre usuari.

Cas del fitxer tingui contrasenya, hi ha fitxer que guarda tots els usuaris/contrasenya

`/etc/passwd r w _ r _ _`

Permisos només de lectura i escriptura del propietari (root) i només lectura als altres usuaris

Per canviar una contrasenya donar permisos provisionals

Comanda canviar password : `passwd`      modificar el fitxer `/etc/passwd`

Permisos del executables `passwd` son:      `/bin/passwd r w s _ _ x _ _ x`

Amb la s: li diem que un usuari executri aquest fitxer el seu usuari efectiu serà el de root

Hi ha la possibilitat de crackear programes i per poder entrar al sistema com a root.

## 2.8 Introducció i terminologia de Signals

Un signal és un event/aconteixement que es produeix és una interrupció software que el SO envia al procés per informar-lo d'algun event asíncron o una situació especial, que ofereix una determinada informació (significat del propi signal).

Asíncron=Interrompen immediatament el procés el precís moment que succeeix

Síncron= senyal que passa cada cop que s'executa una determinada instrucció

### 2.8.2 Enviament de signals

- Crida la funció kill: podem enviar qualsevol signal existent en el nostre SO, a qualsevol procés que s'estigui executant (llistat de signals del nostre SO posar terminal kill -l)
- A través del terminal: ex: pulsant ctrl +C es mata el procés. Tots els signals tenen una forma abreviada de cridar-se pel teclat
- Automàticament: temporitzador o alarma que s'esgota el temps i es llençat o excepció de hardware que SO llença un signal- procés s'ha produït aquella excepció
- Una excepció hardware (el proces efectúa una divisió per 0)
- Senyal generat pel terminal (pulsació d'una tecla)
- el propi procés (directa o indirectament). Per exemple, quan una proces escriu sobre una pipe sense lectors es genera un SIGPIPE.

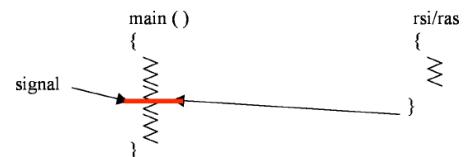
El procés pot testejar la variable `errno` per veure si s'ha produït un signal mentre s'estava executant una sentència del codi que ha retornat l'error.

Exemples de signals:

- Es rep un signal pel venciment d'un temporitzador
- El pare rep un signal que noticia la mort d'un fill
- El SO vol matar al procés li dona un avís però li demana la última voluntat
- Directament el sistema operatiu envia un signal que provoca la mort del procés
- Un altre signal provoca una pausa en l'execució del procés

### 2.8.3 Comportament dels signals:

Accions del SO quan arriba un signal a un procés. Passa a executar el codi de la rutina d'atenció al signal:



- 1) Salvar l'estat del procés
- 2) Executa la rutina de tractament del signal
- 3) Recuperar l'estat original del procés i seguir executant des del punt on es captura el signal – a no se que mori durant la rutina d'atenció al signal (RAS)

Com respon un procés en rebre un signal:

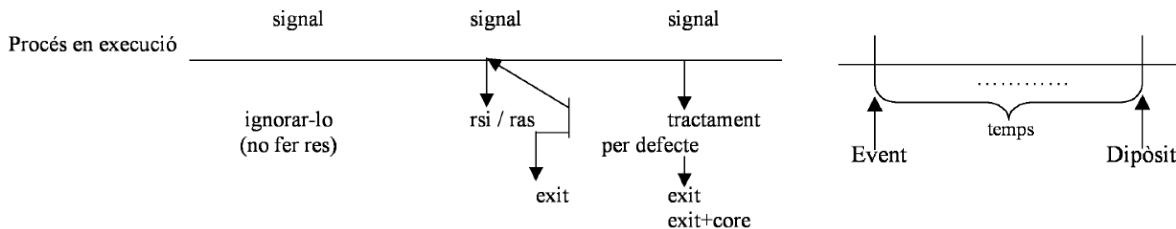
- 1) Actuació per defecte del signal (SIG\_DFL) s'executarà el codi perfecte corresponent a la RAS d'aquell signal concret. És codi pertanyent al sistema operatiu (normalment acabar l'execució del procés) Ex d'actuacions per defecte:
  - a) SIGCONT: Continua l'execució o del procés, si aquest estat aturat.
  - b) SIGSTOP: Atura l'execució del procés.
  - c) SIGKILL: Finalitza l'execució del procés

- 2) Canviar el comportament per defecte d'un signal reconfigurant la RAS. Mitjançant la crida al sistema `sigaction` o `signal` es pot programar o redefinir la pròpia rutina d'atenció al signal
- 3) Pot ser ignorat (`SIG_ING`) l'event de signal es produirà però l'event morirà sense haver fet res- com si no s'hagués produït. Si `SIGCHLD` és ignorant- un fill no queda mai zombie- mor directament i es descarta el seu estat de finalització.  
En la crida `wait` o `waitpid` si el pare espera el fill i no té altres fills vius, la crida retorna error i errno prendrà el valor de `ECHILD`

A l'inici del programa els signals o es tracten per defecte o s'ignoren

#### 2.8.4 Terminologia de signals:

- Generació d'un signal: es produeix l'event que genera el signal
- Dipòsit; s'inicia el tractament associat
- Signal ignorat: al dipositar-lo aquest signora
- Signal bloquejat: el signal no pot dipositar fins que sigui desbloquejat o es canviï el seu tractament perquè no sigui ignorat



#### 2.9 Tipus de Signals

Llistat del conjunt de signals més comuns en diferents SO. Un signal s'identifica per un enter positiu- POSIX defineix també un nom per cada signal:

- **SIGABRT**: abort del procés. L'acció depèn del que l'usuari hagi indicat, sinó no fa res.
- **SIGALRM**: alarma de rellotge. L'acció serà acabar l'execució del procés.
- **SIGBUS**: accés a una part no definida de memòria.. L'acció depèn del que l'usuari hagi indicat, sinó no fa res.
- **SIGCHLD**: indica quan un fill del procés a qui se li envia ha acabat la seva execució o bé si aquesta s'ha pausat o reiniciat. L'acció per defecte es ser ignorada.
- **SIGCONT**: L'execució del procés continuara si estava parada.
- **SIGFPE**: Error en operació aritmètica, com podria ser una divisió per zero. L'acció per defecte dependrà del que l'usuari li hagi indicat, sinó no farà res.
- **SIGILL**: Quan hi ha una instrucció del hardware incorrecte. L'acció per defecte dependrà del que l'usuari li hagi indicat, sinó no farà res.
- **SIGINT**: Atenció del signal interactiu (l'usuari ha generat el signal a través de teclat), per exemple Ctrl-C. L'acció per defecte sera acabar l'execució del procés.
- **SIGKILL**: Matem el procés (aquest senyal no pot ser ni bloquejat ni ignorat). L'acció per defecte sera acabar l'execució del procés.
- **SIGPIPE**: Es produeix quan s'escriu en un pipe que no té processos a punt per llegir-la. L'acció per defecte sera acabar l'execució del procés.
- **SIGSEGV**: Referència invàlida de memòria. L'acció per defecte dependrà del que l'usuari li hagi indicat, sinó no farà res.
- **SIGSTOP**: Para l'execució (d'aquest senyal no pot ser ni bloquejat ni ignorat).

- SIGTERM: Acabament d'algún procés. L'acció sera acabar l'execució del procés.
- SIGTSTP: Terminal parat. Para l'execució (stop)
- SIGUSR1: signal per definir per l'usuari 1. L'acció sera acabar l'execució del procés.
- SIGUSR2: signal per definir per l'usuari 2. L'acció sera acabar l'execució del procés.

## 2.10 Crides a sistema per la gestió de Signals

El SO ofereix rutines per gestionar els signals com:

- Rutines per gestionar un conjunt de signals
- Crides a sistema per bloquejar un conjunt de signals
- Crides a sistema per la generació de signals
- Crides per reprogramar la rutina d'atenció als signals
- Crides per a l'espera de signals

En aquest curs mirarem un subconjunt d'aquestes:

- crides per a la generació de signals: kill alarm
- crida per a reprogramar la rutina d'atenció al signal: signal
- una crida per a l'espera de signals pause

### 2.10.1 Crides a sistema per la generació de signals:

#### **Crida kill:**

Genera un signal de tipus signo a un procés de la màquina especificat pid:

**Sintaxi:** `int kill(int pid, int signo)`

- pid: pid del procés que rep el signal
  - res. > 0 PID d'un procés
  - = 0 a tots els processos del mateix grup del proceso que envia
  - <0 A tots els processos al mateix grup del valor abs. posat
  - -1 POSIX no ho especifica- enviat a qualsevol procés mateix propietari
- signo: identifica el tipus de signal

Retorn	0	ok
	-1	error

Només permès l'enviament de signals sobre processos del mateix usuari.

Pot ser que un procés es generi un signal sobre si mateix:

**kill(getpid(),SIGUSR1)**

#### **Crida alarm:**

Permet l'enviament de signals només del tipus SIGALRM i no ho envia immediatament→ programa l'enviament amb un temporitzador:

Envia el signal SIGALRM al cap de **seconds** segons al mateix procés que realitza crida

**Sintaxi:** `int alarm (int seconds)`

- seconds: temps inicial del temp. en segons (màxim 31 dies en segons)

Retorn	número de segons que queden fins a exhaustir la petició anterior
--------	--

Només existeix un temporitzador actiu al mateix moment	alarm(0) desactiva la petició en curs.	Per a generar una alarma ara mateix
alarm(7); ... (< 7 segons) alarm(3);	alarm(7); ... (< 7 segons) alarm(0);	<b>kill(getpid(),SIGALRM);</b>
Retorna quants segons quedaven.	Retorna quants segons quedaven.	

Exemple crida alarm:

Programa rebi in string. pot ser izquierda o derecha. Programa crear dos processos fills  
 Si per ha rebut derecha el primer fill mata el pare (hauria de matar germà de la dreta però no sap el seu pid (encara no creat llavors mata el pare))  
 Si ha rebut izquierda el segon fill matarà el primer (sabrà pid germa 1 per fork- hereda variables del pare)

```
int main(int argc, char *argv[])
{
int pid1, pid2, fd[2];
char s1[] = "izquierda";           //inicialitzar s1
char s2[] = "derecha";            //inicialitzar s2
if (argc!=2){                     //si crida incorrecte no pot entrar res
    printf("Paràmetres incorrectes\n");
    exit(-1);
}
if ((pid1=fork()) > 0) pid2 =fork(); //creació dels fills pid1= fill 1 pid2=fill 2
if (pid1 == 0){                  //si es rep abans dreta
    if (strcmp(argv[1],s1) == 0) kill(getppid(), SIGKILL); //matar a pare
    else while(1);                //esperar que acabi d'executar se
    printf("He matat al meu pare\n"); //sortida que ha matat el seu pare
    exit(-1);                    //sortir
}
if (pid2 == 0){                  //si rep el esquerra abans
{
    if (strcmp(argv[1],s2)==0)
        kill(pid1, SIGKILL);      //mata el pid1=el germà de pid2
    else while(1);              //mentre ho hagi fet
    printf("He matat al meu germa1\n"); //sortida que ha matat el germà
    exit(-1);                  //sortir
}
wait(NULL);                      //no esperar a cap fill
wait(NULL);                      //no esperar a cap fill
exit(0);                         //acabar programa
```

### 2.10.2 Programació de la rutina d'atenció al signal

Cada senyal té associada una acció per defecte en el procés que el rep. des de ser ignorat a finalizar, aturar execució o continuar....

Pot modificar-se en cada signal a excepció de SIGKILL i SIGSTOP -> si es comporta diferent a l'acció q té associada es diu **captura el signal**

#### Crida sigaction:

La crida al sistema que permet fer la captura el signal- comportament diferent a l'acció associada que te per defecte. Modifica/ consultar rutina tractament d'un signal

**Sintaxi:** `typedef void (*sighandler_t) (int);  
sighandler_t signal(int signo, sighandler_t func);`

Paràmetres:

- Signo: identifica el tipus de signal associat
- func: punter a la nova rutina d'atenció al signal- seguir la sintaxi de typedef
  - !NULL: nou tractament associat al signal
  - NULL: operació de consulta

Retorn	SIG_ERR	si error i erno conté l'error
	punter a la rutina d'atenció al signal anterior	ok

Exemple sigaction:

Reprograma la rutina d'atenció al SIGALRM mostri un missatge per pantalla

```
#include <stdio.h>
#include <signal.h>
void f_alarma(); // Implementada mas abajo...
void main()
{
    long i;
// Programo la rutina de atencion al signal: f_alarma signal(SIGALRM, f_alarma);
// Y programo una alarma para dentro de 1 segundo alarm(1);
// Ahora se supone que el proceso se dedica a hacer cualquier tarea que no tiene nada que ver con el tiempo.Por ejemplo, aqui hay un bucle infinito que escribe por pantalla.
    while(1) {
        // Le ponemos un retardo, para que de tiempo para ver visualmente los mensajes
        for( i = 0; i < 20000000; i++);
        printf("Estoy haciendo cierta tarea...\n");
    }
    // Para acabar la ejecucion, pulsa ^C
}void f_alarma(){
// Escribo mensaje de tiempo y reprogramo la alarma de reloj
printf("\n*****\n");
printf("*** TIEMPO ***\n");
printf("*****\n\n");
alarm(1);
}
```

### 2.10.3 Espera de signals

Podem utilitzar els signals punt de sincronisme entre dos processos- procés envia un signal en un moment donat per avisar a un segon procés que no executaria fins l'espera d'aquest signal.

#### Crida pause:

Permet bloquejar el procés de la crida a l'espera d'un subconjunt de signals -> suspèn l'execució a l'espera de qualsevol signal ← no podem controlar quins signals ho desbloqueen serà el primer que arribi.

**Sintaxi:** int pause()

Es pot cridar fent una espera activa (while(1)) esperar l'arribada d'un signal- consumeix cpu en espera activa

El tractament SIG\_IGN aquest signal no desbloquejarà pause ni provocarà l'error EINTR

Retorn	-1 sempre amb errno	EINTR
--------	---------------------	-------

Exemple: Com capturar el SIGALRM i reprogramar l'alarma cada segon mostri els processos que s'estan executant (executi ps)

```
#include <errno.h> #include <unistd.h> #include <stdlib.h> #include <signal.h>#include <sys/types.h> #include <sys/wait.h>
int alarma=0;
void error (char *msg) {
    perror (msg);
    exit (0);
}
void crea_ps () {
    int pid;
    pid = fork ();
    if (pid == 0){
        execvp ("ps", "ps", (char *) 0);
        error ("execvp");
    } else if (pid < 0) error ("fork");
}
void f_alarma (int s) {
    alarma=1;
}
void fin_hijo (int s) {
    while (waitpid(-1,NULL,WNOHANG)>0);
}
void main (int argc, char *argv[]) {
    int x = 0;
    signal (SIGALRM, f_alarma);
    signal(SIGCHLD,fin_hijo);
    while (x <10) {
        alarm (2);
        while (alarm==0) pause ();
        alarma=0;
        crea_ps ();
        x++;
    }
}
```

## 2.11 Interacció dels signals amb altres Crides al sistema:

- Crides al sistema bloquejants i signals:

Crida bloquejant està en procés d'espera- un signal pot provocar (desp d'executar la RAS) que avorti retornant -1 (variable errno= EINTR)

→ per evitar-ho posar la crida bloquejant dins d'un bucle - reiniciar si ha estat interrompuda per un signal (errno es EINTR)

→ tmb posant el flag SA\_RESTART s'activa per defecte quan capturem un signal amb rutina ([signal](#)) però si la crida ja incialt la seva execució- en rebre un signal pot retornar abans d'haver executat

Exemples de crides afectades:

- read sobre pipes, terminal, sockets.
- write sobre pipes, terminal, sockets si les dades no son acceptades immediatament.
- wait / waitpid . . .

Fork / Exec i signals: Tot procés té:

- una taula d'accions associada als signals
- una llista de signals pends
- una mascara de signals bloquejats  
(s'hereta tant en el fork com en el exec)

Fork: proceso nou → reset temporitzador d'alarma (alarm(0)) en el fill.

Si existeixen signals pends no es tenen en compte els fill, la llista de signals pends d'esborra en el procés del fill- fill és un procés nou.

Els signals capturats passen del pare a fill - duplica tot el codi + taula d'accions de signals

Exec: mateix proceso → no resetja el temporizador d'alarma.

signals a SIG\_DFL taula d'accions assiciada a signals es posa per defecte el codi es perd menys so SIG\_ING - info pertany a PCB

Els events son enviats a processos concrets (PID) i el procés no canvia i llista sig. conserva

## Gestió de Processos: Kernel

### 2.12 Gestió interna de Processos:

Per gestionar processos el SO necessita:

- Estructures de dades
- Estructures de gestió
- Algorismes de planificació
- Mecanismes

#### 2.12.1 Process Control Block:

Tota la info referent a un procés s'emmagatzema a estructura de dades interna de SO anomenada PCB que descriu l'estat del procés en cada moment.

La info que conté depèn del sistema però pot contenir també:

- PID ( Process Identifier )

Identificador únic del procés dins del sistema. S'assigna al crear el procés i invariable en execució. PID=1 procés creat pel SO al final del procediment de boot -inicialitzar el sistema

- UID ( User Identifier )

Propietari del procés (pot canviar). Límita els recursos al quals el procés pot tenir accés

- Process State

- Program Counter , registres

Conté informació sobre l'estat d'execució del procés. PC i tota informació necessària pel planificador per suspendre/resumir l'execució: contingut reg. mapeig de memòria ...

- Address Space , memory límits

Espai de memòria al qual el procés té accés

- File Descriptors ( I /O ) Dispositius virtuals E/S:

Conjunt de descriptors de dispositius els quals tenen accés per E/S

- Priority/Prioritat d'execució:

Prioritat assignada a cada procés per determinar qui serà el següent en executar-se

- Signal and exception management policies

Conjunt de normes que segueix un procés quan es produeix una excepció/signal.

Taula de rutina d'atenció als signals/Mascara signals bloquejats/Taula signes pendants

- Process Statistics Accounting

Informació de contabilitat de recursos que s'estan utilitzant i els que ha consumit executant

- Temps de CPU

- Memòria ocupada

### 2.12.2 Estructures de gestió per organitzar processos

Organitzin els PCB en funció de estat/necessitats del sistema Normalment son llistes/cues o taules de hash o arbres - Han de ser eficients (rapida inserció,eliminació,búsqueda id o usuari..) i escalables (dependrà del sistema, processos actius, memòria necessària...)

Estructures que fa servir SO per organitzar processos **cues de planificació** tmb inclou:

- Cua de processos: que inclou totes els processos creats pel sistema
- Cua de processos ready. Processos que estan a punt per executar-se, esperant la CPU. En molts sistemes, aquí més d'una cua- agrupades per classes prioritats etc ..
- Cues de dispositius: processos que estan esperant dades algun dispositiu E/S.

El sistema mou els processos d'una cua a l'altre segons correspongui

### 2.12.3 Algorismes de planificació:

Quan tenim més processos (threads) que CPUs → compartir el recurs CPU

El SO decideix quin procés/thread toca executar, temps...

Depèn del sistema nivells de planificació: dos nivells:

- Long-term scheduler or job scheduler.

Selecciona els processos que passen a estar a la cua de ready i son candidats a executar.

- Short-term scheduler or CPU scheduler.

Selecciona quins processos (o threads) s'executen en cada moment dels disponibles a la cua de ready i se'ls hi assigna un temps de CPU. El temps pot ser límitat o no.

Normalment ho és i el SO determina el temps màxim. L'algoritme que decideix quan un procés ha de deixar la CPU, qui entra i quanta estona-**Política de planificació o scheduler**. La política pot ser apropiativa o no.

- Els no apropiatius el procés ha d'abandonar la CPU voluntàriament
- Els apropiatius el sistema operatiu li pot prendre, es defineix el quantum.

### 2.12.4 Mecanismes de planificació:

**Canvi de context (context switch)** SO treu un thread CPU i posa un altre.

El sistema ha de salvar l'estat del thread/procés i restaurar l'estat del thread/procés que passa a executar-se.

El context del procés es vol guardar en les dades del kernel -representa procés (PCB)

El canvi de context no es temps útil de l'aplicació- cal ser ràpid

### 2.12.5 Objectius i Mètriques de la planificació:

Polítiques de planificació diferents objectius depenen del sistema:

- Temps d'ús de CPU: Objectiu es mantenir la CPU tan ocupada com sigui possible.
- Throughput: quantitat de processos que acaben la seva execució/unitat de temps.
- Turnaround time: temps total d'execució d'un procés, arriba al sistema fins que acaba.
- Temps d'espera: d'un procés es el temps que passa a la cua de ready.
- Temps de resposta: desde que el procés arriba al sistema fins a executar-se.

Pot passar degut a la política un procés mai toqui la CPU mai executi- **starvation** (evitar)

Exemple de política:

- **First-Come First-Served FCFS scheduling.** En FCFS els processos s'executen en ordre d'arribada. No es treuen de CP ni si estan bloquejats. algoritme no apropiat
- **Round Robin.** Es apropiat, els processos s'encén per ordre d'arribada i cada procés rep la CPU durant un període de temps límit (quantum) típicament del ordre de 10-100 ms.  
El planificador utilitza la interrupció de rellotge per assegurar-se que cap procés monopoliza la CPU- Acaba el quantum (o també si el proceso es bloqueja per E/S o acaba) el proces deixa la CPU i es selecciona el següent de la cua de ready per a executar-se. El procés primer s'afegeix al final de la cua (si es que no havia acabat).

Calcular rendiment: N processos a la cua de ready/ quantum es de Q ms (temps donat)

Cada procés rep  $1/N$  parts de temps de la CPU en blocs de Q ms màxim.

Assegurar cap procés esperarà més de  $(N - 1) Q$  ms (de temps)

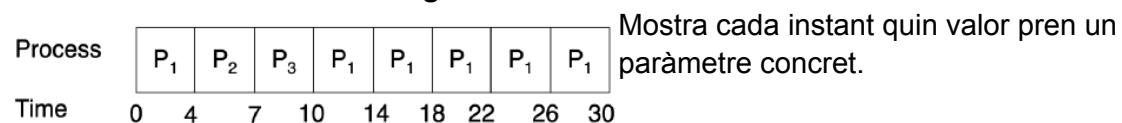
La política en funció del quantum:

- Q molt gran → comporti com una FIFO. Efectes pràctics en processos/ threads que reben la CPU fins que es bloquegen
- Q molt petita → tenir en compte el cost del canvi de context- sinó **overhead**

Exemple:

Process	Burst time	Següent taula temps d'execució dels processos amb Quantum=4 aplicant Round Robin
P1	24	
P2	3	
P3	3	

Planificació concreta utilitza **Diagrama de Gantt**



$P1=6$   $P2=4$  i  $P3=7 \rightarrow$  temps mig d'espera  $t = 5,6$

### Tema 3: Gestió de dispositius d'entrada i sortida

Entrada i sortida com la transferència d'informació des de o cap a un proceso

El dispositiu és l'element que permet fer l'operació de E/S (xarxa, disc, fitxers, cd, impressora)

Els processos realitzen el càlcul i E/S, que es la tasca principal del procés (navegació web)

SO administrar el funcionament dels dispositius E/S per un ús correcte, compartit i eficient dels recursos. Impedir al l'usuari accés directe als dispositius comporta:

- El SO aïlla l'usuari de les característiques físiques dels dispositius, facilita la feina programadors d'applicacions amb codis més portables i comprensibles.
- Compartició de recursos de manera robusta i segura

#### Tipus de dispositius:

- dispositius d'interacció amb l'usuari

(pantalla, teclat, mouse, d'emmagatzemament (disc dur, DVD, pendrive)).

- de transmissió

(modem, xarxa amb cable, xarxa sense cable)

- tipus més especialitzats

(controlador d'un avió, sensors, robots).

Per poder gestionar aquesta diversitat, conceptes clau en gestió d'E/S a Unix són:

- Independència de dispositius

- Mapeig dels dispositius sobre el sistema de fitxers.

#### 3.1.1 Independència de dispositius

- **Uniformitat de les operacions d'E/S:** El SO mateixes crides a sistema per tots els dispositius. Disposem de cinc crides (**open, close, read, write, ioctl**) permeten accedir a qualsevol dispositiu de la mateixa manera i el SO ja s'encarregarà de diferenciar els dispositius internament. Augmenta la simplicitat i la portabilitat dels programes d'usuari.

- **Dispositius virtuals** El SO ofereix als programadors una abstracció dels dispositius en forma de canals o dispositius virtuals, així el programa no especifica sobre quin dispositiu treballa sinó interacciona sempre amb els dispositius virtuals (descriptor de fitxer)

- **Redirecció** El SO permet canviar l'assignació dels dispositius virtuals abans o durant l'execució d'un procés.

Permet que el mateix programa s'executi sobre diferents dispositius sense canviar el codi. Per exemple, desde l'intèrpret de comandes podem redireccionar l'E/S d'una comanda: **% comanda < disp1 > disp2**

mateix servei per a escriure dades per pantalla que per a escriure dades per la impressora o sobre un fitxer: serà feina interna del SO diferenciar a quin tipus de dispositiu s'està accedint

1. **Nivell Físic:** dispositius físics a nivell hardware (disc). Implementa a baix nivell les operacions abstractes sol·licitades Tradueix paràmetres del nivell abstracte a paràmetres concrets. Per exemple en un disc, tradueix el punter de lectura/escriptura a cilindre, cara, pista i sector del disc.

El SO implementa la operació per accedir als dispositius físics, això pot incloure examinar l'estat, posar motors en marxa (disc) ...

El SO no pot programar el codi per a cadascun dels diferents dispositius, marques i models existents. La solució és el **device driver** → fabricant del dispositiu proporciona rutines d'accés a baix nivell dels dispositius.

2. **Nivell Lògic:** Implementa taques independents dels dispositius. Estableix correspondència entre nom simbòlic (path al **VFS Sistema de Fitxers**) → UNIX tots els dispositius lògics són representats al sistema de fitxers ) i el controlador adequat. Aquest nivell proporciona una interfície uniforme al nivell físic

Existeix un fitxer especial dins del directori /dev cada dispositiu lògic ,poden:  
tenir un dispositiu físic associat, com és el cas del

- mouse: un dispositiu lògic correspon a un dispositiu físic.
- terminal: pantalla més teclat /dev/tty:dispositiu lògic correspon a dos de físics.
- dispositiu físic, la pantalla, pot tenir més d'un dispositiu lògic associat, més d'un terminal(/dev/tty1, /dev/tty2) També el disc amb particions /dev/hda1 /dev/hda2...

pot ser que no existeix dispositiu físic associat:

- /dev/null no té cap dispositiu físic associat

3. **Nivell virtual:** l'abstracció que ofereix el SO de dispositius lògics (nom simbòlic) als processos. Procés sempre interactua amb dispositius E/S través dispositius virtuals.

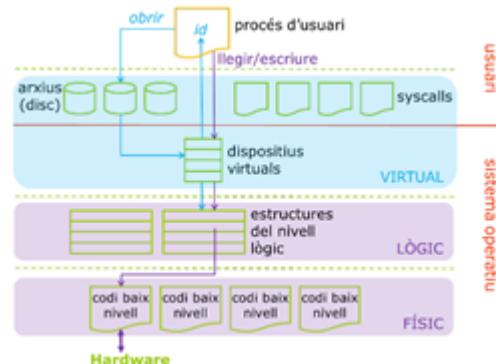
- A UNIX s'anomena canal o descriptor de fitxer (file descriptor) i és un enter.
- Els processos tenen tres canals estàndards oberts per defecte:

**stdin (0) stdout(1) stderr(2)** .

- Les estructures de dades que guarden informació dels dispositius i permeten al sistema operatiu gestionar aquesta abstracció són la Taula de Dispositius **Virtuels Oberts (TDVO)** o Taula de Canals, i la Taula de Fitxers Oberts (TFO). També anomenades file descriptor tables.
- o El SO proporciona les funcions genèriques d'accés als dispositius, a Unix son les crides al sistema d'E/S.
- o La funció específica per establir l'associació dispositiu virtual amb nom simbòlic és: open (obrir el dispositiu)

Seguint aquest disseny a tres nivells es poden fabricar nous dispositius sense modificar el kernel el problema és que aquesta implementació a nivell físic és molt dependent del dispositiu concret. Així, depèn de:

- o quin tipus de dispositiu és: impressora, disc...
- o les característiques tècniques: velocitat de transmissió, unitat de transferència...
- o protocol de comunicació entre el processador i el controlador...



### 3.1.2 Mapeig dels dispositius sobre el sistema de fitxers: **Virtual File System**

Tots els dispositius tenen un nom al Virtual File System, permet qualsevol procés pugui accedir al dispositiu i proporciona portabilitat al codi

Normalment els dispositius es troben al directori/dev

Dispositius com a fitxers especials que:

- major: identificador de tipus de dispositiu
- minor: instancia de dispositivo del tipus "major"
- tipus: orientat a blocs o caràcters ( char or block )

Creacio d'un fitxer especial: **mknod nom c|b major minor**

Crida al sistema per la creació d'un fitxer especial: **mknod(pathname, mode, dev)**

- mode: tipus de caràcter o block i proteccions del fitxer
- dev: codifica el major i el minor

### 3.1.3 Device Driver

E/S aïlla la resta del kernel la complexitat de la gestió de dispositius, tmb d'un codi extern

S'identifiquen unes operacions comuns que formen la interfície que el fabricant ha de programar i les diferències específiques de cada dispositiu (marca, model ...) encapsulen dins del device driver.

**Device Driver(DD)** software especifica una forma un conjunt de rutines encapsulades que proporcionen funcions d'accés/control d'un determinat dispositiu.

Ha de respectar la interfície definida pel SO associada al major (que indica el tipus de disp.) i la minor(indica la instancia del dispositivo del tipus major) corresponents.

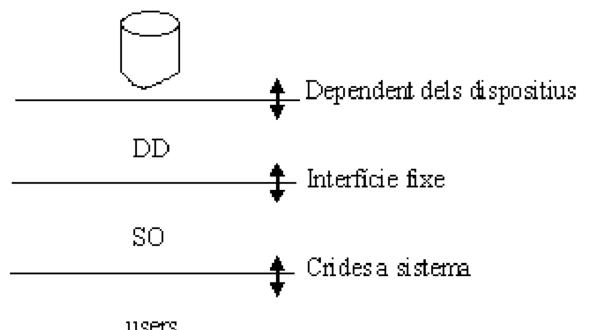
Les crides al sistema usen una interfície fixa per a tots els dispositius del mateix tipus

Cada fabricant proporciona el driver adequat per un SO determinat.

Pasos de creació al ús del dispositiu son:

- **Instal·lació** (inserció= en temps d'execució de les rutines del driver. Als dispositius s'identifiquen amb 2 noms: major i minor:  
**insmod fitxer\_amb\_codi\_acces\_dispositius**
- **Creació del dispositiu lògic** o nom simbòlic (nom d'arxiu en el FS) i lligar-lo amb el driver. Ho fem a través del major i minor. Utilitzant la comanda mknod  
**mknod /dev/mydisp c major minor**
- **Enllaç** del nom simbòlic amb el file descriptor utilitzant el nom (path) del arxiu:  
**open("/dev/mydisp", ... )**

El sistema de fitxers es comunica amb els dispositius d'emmagatzemament secundaris (discs) mitjançant els device drivers proporcionan protocol de comunicació entre SO i els perifèrics



### 3.2 Gestió de l'Entrada/Sortida a UNIX

Dos tipus de dispositius diferents segons la forma d'accés:

- **Block devices** (dispositius de block): l'accés es porta a terme mitjançant la intervenció de buffers que milloren molt la velocitat de transferència, un exemple és el disc.
- **Row devices** (orientats a caràcters): L'accés és directe, sense intervenció de buffers. Un exemple és el teclat.

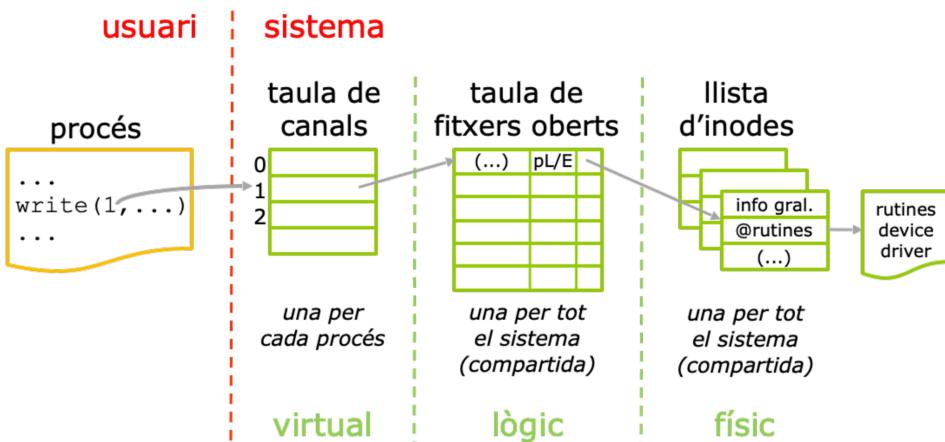
Un mateix dispositiu físic pot ser manejat en els dos modes, depen del handler que utilitzem per accedir a ell

Operacions de E/S poden ser bloquejants o no bloquejants

- OP. Bloquejant: executa l'operació i retorna quan ha acabat  
Si pot completar-ho amb les dades demanades ho fa, altrament retorna les dades disponibles i el numero de dades.  
Si no hi ha dades disponibles es bloqueja → mou el procés a una cua d'espera (els bloquejats) i es posa en execució el primer procés de la cua.  
Quan arriba una dada es produeix una INTERRUPCIÓ → RAI (Rutina d'Atenció a la Interrupció) recull la dada i la torna a la cua. Quan el procés de la cua es posi en execució se li donaran les dades (op. de E/S estan bloquejades per defecte)
- OP. NO Bloquejant executa l'operació i retira tant si hi ha dades com si no, retorna el nombre de dades llegides o escrites
- En una OP E/S NO bloquejant asíncrona realitza una petició de dades → segueix executant el codi i quan arriben les dades: pot fixar algun valor en alguna variable o es produeix una interrupció software per avisar el procés

### 3.2.1 Estructures de dades per la gestió de L'E/S

El SO utilitzar tres estructures de dades que corresponen a 3 nivells de disseny:



#### 1) Taula d'i-nodes:

Correspon al nivell físic Conté infosobre objecte físic obert, incloent Device Driver.

És única per a tot el SO. Conté una entrada per cada fitxer (inode) del sistema. Conte informació com ara:

- mida del fitxer: en bytes
- propietari del fitxer
- grup al que pertany
- proteccions
- tipus de fitxer: ordinari, directori, pipes, dispositius especials
- temps sobre l'accés al fitxer: data últim accés, modificació...
- nombre de links: nombre total de noms que el fitxer té en la jerarquia del sistema de fitxers. Un fitxer pot tenir associats diferents noms que corresponguin a diferents rutes però mateix i-node→mateixes dades.
- llista de blocs: que apunten a les dades.

#### 2) TFO: Taula de fitxers oberts

Única per a tot el SO, compartida per tots els processos. Implementa el nivell lògic Conté informació sobre tots els fitxers oberts en el SO per algun procés.

Nova entrada cada cop que obrim un fitxer, que conté:

- número de links : número de dispositius virtuals que apunten al fitxer
- punter lectura/escriptura : punter a la posició actual del fitxer des de l'inici del fitxer, sobre quin byte del fitxer tindran efecte les operacions→ actualitza
- Indicadors de mode d'apertura: o\_rdwr, o\_append ...
- numero d' i-node : número d'i-node associat al fitxer

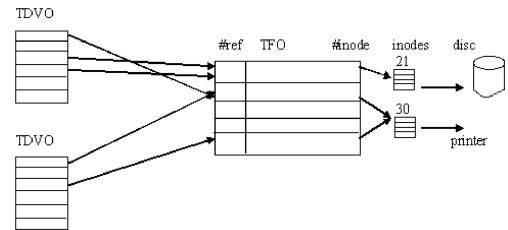
#### 3) TDVO: Taula de dispositius virtuals oberts

Una per a cada procés. Proporciona el nivell virtual. Conté punters a la TFO, identificant tots els fitxers oberts per un procés. Poden existir diferents punters, en la mateixa o diferents TDVO que apunten a la mateixa entrada de la TFO.

Al crear un procés nou la TDVO té 3 entrades per defecte: stdin (0), stdout (1), stderr (2) que poden ser modificades mitjançant les redireccions.

Ex. `ls > hola.txt; ls | more`

Treballar sobre un fitxer, l'identificar pel file descriptor, retornat per la system call **open** i utilitzat per les altres system calls: **read**, **write**, **Iseek** processos accediran a un fitxer mitjançant el seu file descriptor associat (número de canal, dispositiu virtual, descriptor de fitxer o entrada de la TDVO).



### 3.3 Crides a Sistema relacionades amb l'Entrada i la Sortida

#### 3.3.1 Associar Canals Virtuels a dispositius

El SO proporciona crides al sistema per associar Canals Virtuels a disp. físics → poder operar (ope, creat, pipe, socket)

Tenint un canal → porta d'entrada al dispositiu i realizar op de lectura i escritura (**read/write**)

Per finalitzar l'associació del canal virtual al físic utilitzar la crida **close**

Si afegim bytes sobre el final del fitxer → incrementa la mida

Trucar la mida a 0 → esborrar tot el seu contingut

Dos processos poden llegir/escriure concurrentament sobre el mateix fitxer ← resultat depèndrà l'ordre d'execució entre els dos processos depenen del scheduler/planificador

#### Crida Open

Associa un descriptor de fitxer (file descriptor) o canal virtual a → fitxer o disp. físic

El SO habilita les estructures de dades per poder treballar amb el fitxer

**Sintaxis:** `int open (const char *path, int oflags)`  
`int open (const char *path, int oflags, int mode)`

- path: nom absolut o relatiu del fitxer o dispositiu a obrir
- oflags: mode d'obertura del fitxer

obligatori	<code>O_RDONLY</code>	obre el fitxer només de lectura
	<code>O_WRONLY</code>	només escriptura
	<code>O_RDWR</code>	lectura i escriptura
opcional	<code>O_CREAT</code>	si el fitxer no existeix, el crea, si ja existeix, l'obra
	<code>O_EXCL + O_CREAT</code>	si no existeix, el crea, si ja existeix, retorna error
	<code>O_TRUNC</code>	borra el contingut inicial del fitxer, sobreescrivides del principi, requereix permisos d'escriptura
	<code>O_APPEND</code>	escriu sempre a partir del EOF (final de fitxer), i requereix permisos d'escriptura

- mode: permisos del fitxer

<code>S_IRWXU</code>	700	permisos de lectura i escriptura i execució pel propietari només
<code>S_IWGRP</code>	020	el usuaris del grup tenen permís d'escriptura

#### Retorna

> 0	descriptor del fitxer assignat (index a la TDVO)
- 1	cas d'error

Cada procés té una màscara **unmask** que és utilitzada a la crida open alhora de donar permisos a un nou fitxer creat. Paràmetre mode indica a la cria se li aplica una **and logica** amb el **negat de la mascara** del procés **unmask (mode & ~umask)**

La màscara depèn de l'usuari (norm. 022) es pot consultar/modificar amb comanda **umask**

### Exemples: Open

```
fd = open("hola.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU | S_IRWXG);
```

Obrim el fitxer per escriure sobre ell, esborrem el contingut i comencen a escriure des de l'inici. Si el fitxer no existeix es crearà amb els permisos 077, amb tots els permisos pel propietari i el seu grup.

```
fd = open("hola.txt", O_RDWR | O_APPEND);
```

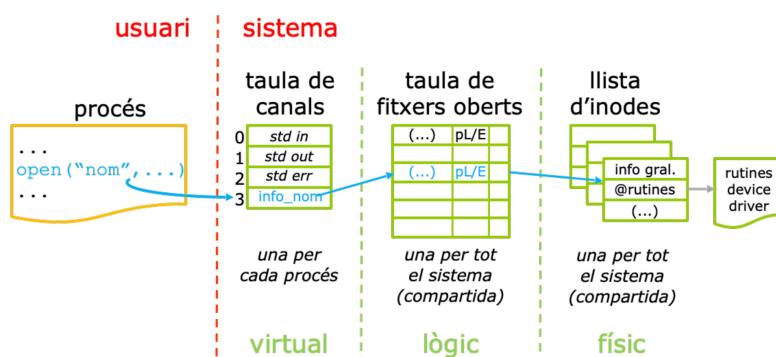
Obrim amb permís per llegir/escriure sobre el fitxer, escrivim sobre EOF, és a dir, afegir continguts al fitxer.

```
fd = open("hola.txt", O_CREAT | O_EXCL | O_RDWR, S_IRWXU,S_IRWXO);
```

Si existeix el fitxer, retorna error si no existeix el fitxer, el crea amb permisos de lectura, escriptura i execució pel propietari i els altres (S\_IRWXU |S\_IRWXO), i l'obre per lectura i escriptura (O\_RDWR).

### Accions del SO davant d'un **open**:

- Busca la primera entrada lliure de la TDVO (la menor).
- Crea (ocupa) una nova entrada en la TFO: per defecte el punter de lectura/escriptura estarà a l'inici del fitxer, el mode d'obertura serà aquell indicat a la crida open (flags) i el número de links estarà a 1.
- L'entrada ocupada de la TDVO apunta a la nova entrada de la TFO.
- Associa aquestes estructures al DD corresponent (major del nom simbòlic) a traves de la taula d'i-nodes. Pot passar que diferents entrades de la TFO apunten al mateix i-node.



### Crida Close

Allibera un dispositiu virtual

**Sintaxis:** `int close (int fd)`

- fd: descriptor del fitxer que volem alliberar → entrada a la TDVO

**Retorna**

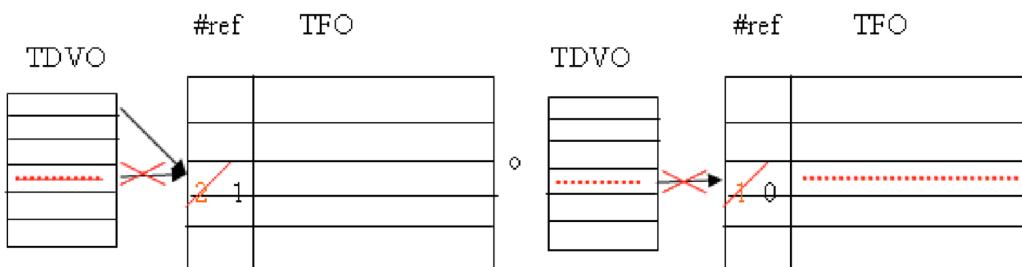
0	OK
- 1	cas d'error

**Exemples: Close**

```
close(0); fd=open("fitxer",O_RDWR); close(fd);
```

Accions del SO davant d'un **close**:

- Allibera l'entrada a la TDVO.
- Decrementa el número de referències (número links) a la TFO, i si és 0 allibera l'entrada.
- El mateix passa amb els apuntadors a la llista d'inodes.
- És necessari per estalviar recursos al sistema. Si no fem el open al acabar el procés el SO tanca tots els canals oberts.



### 3.4 Lectra i escriptura sobre fitxers o altres dispositius d'E/S

Un cop tenim un dispositiu físic associat a un disp. o canal virtual es poden realitzar les operacions de lectura i escriptura sobre aquest.

#### Crida Read

Llegir un dispositiu virtual → transmetre un màxim de n bytes des del disp. virtual indicat fins a la memòria intermitja apuntada per buffer

**Sintaxis:** `int read (int fd, void *buffer, int nbytes);`

- fd: dispositiu virtual o descriptor del fitxer → entrada a la TDVO
- buffer: adreça on es deixen els bytes llegits, (ha d'estar inicialitzada!)
- nbytes: nombre de bytes que volem llegir (si és possible)

**Retorna**

0	EOF (si intentem llegir més enllà del final de fitxer)
>0	número de bytes llegits realment
- 1	cas d'error (ex. dispositiu virtual no obert)

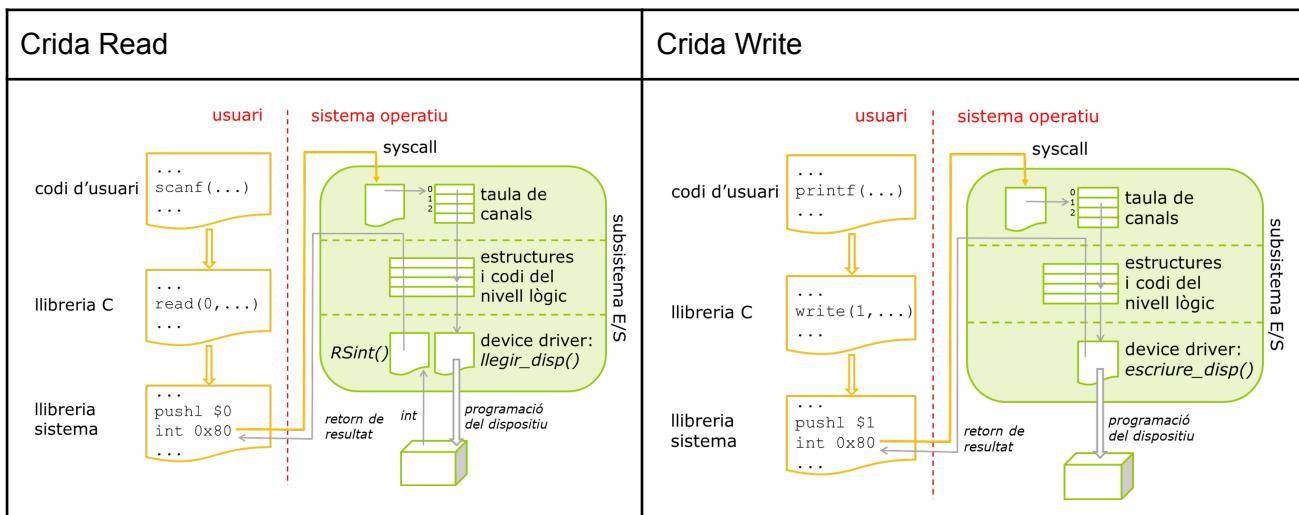
## Crida Write

Escriure un dispositiu virtual → transmetre un màxim de n bytes de la memòria intermitja apuntada per buffer (espai lògic del procés) fins al dispositiu virtual indicat (fd)  
disp. virtual indicat fins a **Sintaxis:** `int write(int fd, void *buffer, int nbytes);`

- fd: dispositiu virtual o descriptor del fitxer → entrada a la TDVO
- buffer: buffer que conté els bytes (dades) que volem escriure
- nbytes: numero de bytes a escriure (si és possible)

## Retorna

>0	número de bytes escrits realment
- 1	cas d'error (ex. dispositiu virtual no obert)



### 3.4.3 Característica de les crides read/write

- Actualitzen implícitament el punter de lectura/escriptura, avançant-lo tantes posicions com bytes llegits/escrits.
- En la crida write, si el fitxer ha estat obert amb O\_APPEND, primer el SO posa el \*r/w sobre EOF i després fa el write, és a dir, afegeix informació.
- Cas de write. Si hi ha espai per nbytes o més, n'escriura nbytes menys → escriurà els que hi càpiguen, comportament dependrà del dispositiu: es pot bloquejar esperant que hi hagi espai o bé retornar sense escriure res.
- Cas de read: Si hi ha menys caràcters disponibles dels sol·licitats, llegira els que hi hagi. no n'hi ha cap, dependrà del funcionament del dispositiu: es pot bloquejar esperant que n'hi hagi o bé pot retornar amb cap caràcter lleigit.  
Quan final del fitxer (EOF), l'operació retorna sense cap caràcter llegit.
- Les crides read o write, transfereixen bytes, és responsabilitat del programador tenir en compte els bytes que ocupen els tipus de dades:  

```
// codi 1    int i = 8;      write(1,&i,4);    // codi 2    char c = '8';    write(1,&c,1);
```
- Tenim funcions C de suport com **sizeof** o **strlen**.

### 3.5 Filtres i redirecció de canals

#### 3.5.1 Filtres

Filtre → programa que llegeix dades de l'entrada estàndar (stdin 0), fa una transformació a les dades i treu el resultat per la sortida estàndard (stdout 1) --> si error indica canal estàndar d'error (stderr 2)

#### 3.5.2 Redirecció

Redirecció → modifica l'entrada de la TDVO de manera que apunta a una entrada diferent de la TFO

>	ls > fitxer.txt	redirecció de la stdout
<	sort -n 4 < fitxer.txt	redirecció de la stdin
2 >	./configure 2> error.log	redirecciona la stderr
	ls   more	redirecciona la stdout de ls a la stdin de more (mitjançant una pipe)
& >		redirecciona stdout i stderr

### 3.6 dup i dup 2

Crides al sistema que permeten fer la redirecció dels canals

#### Crida dup

La crida dup duplica l'entrada de la TDVO que passa com a paràmetre i es copia a la primera entrada lliure de la TDVO

**Sintaxis:** `int dup (int fd);`

- fd: l'entrada de la TDVO o descriptor de fitxer a duplicar
- nbytes: nombre de bytes a escriure (si és possible)

**Retorna**

>0	nou descriptor de fitxer (primera entrada lliure de la TDVO)
- 1	error: el dispositiu virtual no és obert o TDVO és plena

Accions del SO davant d'un dup:

- Busca la primera entrada lliure de la TDVO (la menor).
- Fa que apunti a la mateixa entrada de la TFO que el paràmetre fd:
  - Incrementa el nombre de links de l'entrada de la TFO.
  - Comparteixen l'entrada TFO: mateix punter de lectura i escriptura.

```
void main (int argc, char *argv[]){
    int fd1,fd2;
    if (argc!=2) panic ("Paràmetres incorrectes");
    fd1=open(argv[1],O_RDONLY);
    close(0); dup(fd1); close(fd1);
}
```

## Crida dup 2

La crida dup2 mateixa que la dup, però especificant el canal destí, duplica fd\_origen sobre fd\_desti

**Sintaxis:** `int dup2 (int fd_origen, int fd_desti);`

- fd\_origen: el descriptor de fitxer que volem duplicar
- fd\_desti: la posició on el volem duplicar, si el canal està ocupat (obert) el tanca.

## Retorna

>0	fd_desti
- 1	error: fd_origen no obert, TDVO ple

## Crida lseek

La crida lseek mou el punter de lectura/escriptura del fitxer amb descriptor de fitxer fd, les posicions indicades a offset. offset pot ser un número negatiu.

Cap a on va el moviment depèn del paràmetre whence.

**Sintaxis:** `int lseek (int fd, int offset, int whence)`

- paràmetre whence: pot ser els valors;

SEEK\_SET mou el punter a la posició offset

SEEK\_CUR desplaça el punter offset bytes desde la posició actual

SEEK\_END desplaça el punter offset bytes desde fi de fitxer

## Retorna

>0	nova direcció del punter des de l'inici del fitxer
- 1	cas d'error

Permet moure el punter més enllà del fi del fitxer- crear forats que es podenaprofitar

No tots els dispositius suporten aquesta crida- com terminal o pipe

<code>fd = open('abc.txt', O_RDONLY); while (read(fd, &amp;c, 1) &gt; 0) {     write(1, &amp;c, 1);     lseek(fd, 4, SEEK_CUR); }</code>	fd es el fitxer descriptor de fitxer abc.txt mentre es llegeixi de fd i sigui més gran a 0 escriu per pantalla el valor que es trobi en la posició del fitxer. Després amb lseek mou el punter a la posició actual
<code>fd = open('abc.txt', O_RDONLY); size = lseek(fd, 0, SEEK_END); printf("%d\n", size);</code>	fd es el fitxer descriptor de abc.txt variable size= a la funció lseek de fd posició inicial 0 fins al final del fitxer-donara el size i treu la variable size

## Comunicació entre processos

### 3.9 Pipes

Dispositiu lògic per comunicar dades entre processos locals amb el parentesc- FIFO  
comunicació és bidireccional no simultànea-primer un i després espera resposta de l'altre

#### 3.9.3 Pipes amb nom

Les pipes amb nom (fifo) tenen associat un dispositiu lògic present al Sistema de Fitxer amb la comanda

##### **mknod p pipename**

→ crea un fitxer que cal obrir dos opens (un proceso lector i l'altre proceso escriptor)

Al ser present al sistema de fitxers permet comunicar dos processos sense parentesc

#### 3.9.3 Pipes ordinàries

- No té nom en el SF
- **Comunicació de processos amb parentesc**
- Dos canals a la TDVO un de lectura i un d'escriptura
  - entrada/lectura
  - sortida/escriptura
- Reservem dues entrades de la TDVO i la TFO, com si fos un fitxer.

Ei SO li assigna igualment un i-node de tipus pipe, però que no apunta a blocs de dades, sinó que es gestiona en un buffer cache, sense arribar a disc. Per tant, per a transmetre informació és un mecanisme més ràpid que fitxers.

- Contingut volàtil: el seu contingut va desapareixent a mesura que anem llegint.
- Mida màxima depèn del SO (constant PIPE\_BUF), però mai menor a 4096 bytes.
- No té punter de lectura/escriptura(accedir al quart element llegir els tres davant seu)
- Comunicació unidireccional.

Si necessitem tenir una comunicació bidireccional, és millor utilitzar 2 pipes. Amb una també és possible, però podem tenir problemes de sincronisme i per tant, hauríem d'utilitzar signals o algun altre mecanisme per a controlar-ne l'accés. En cas contrari, pot ser que llegim allò que hem enviat nosaltres mateixos.

- No utilitza cap nom al sistema de fitxers (VFS) i no executa la crida al sistema open.
- Només podrà comunicar el procés que la crea i qualsevol descendent d'aquest (perquè necessita heretar els canals)

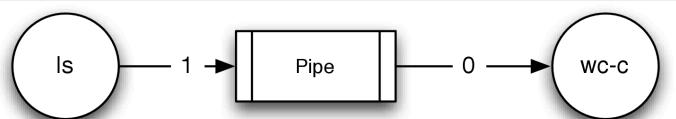
#### 3.9.5 pipes de shell

comanda: ls | wc -c

- Crea la pipe
- Crea dos fills
- efectua les redireccions necessàries

(canal 0 del wc a la sortida de la pipe i el 1 de ls a l'entrada de la pipe, altres continen tancats).

- executa la comanda en cadascun dels fills.



## Crida Pipe

Crea una pipe ordinària, dos canals nous (lectura i escriptura a la TDVO i TFO)

**Sintaxis:** `int pipe(int fd[2]);`

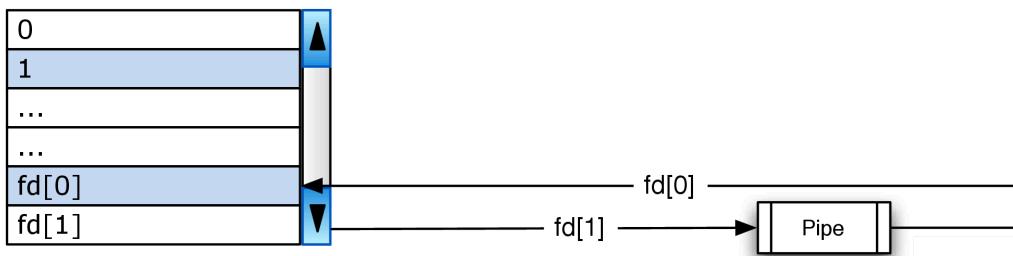
- fd: vector de dos canals (int fd[2];)
  - lectura fd[0] //escriptura fd[1]

**Retorna**

0	ok (s'ha creat la pipe)
- 1	cas d'error

Exemple:

```
int fd[2]; if (pipe(fd) < 0) panic("Error pipe");
```



## Crida Close sobre pipe

S'han de tancar tots dos canals el de lectura i escriptura (tancar canals que no es fan servir)

**Sintaxis:** `int close(int fd);`

- tancar els dos
- close (fd[0]); //close( fd[1]);

Exemple:

```
int fd[2];
pipe(fd); if (fork()==0) { close(fd[0]); }
else { close(fd[1]); }
```

## 3.9.8 Lectura i escriptura sobre pipes

La crides read i write és la mateixa, però s'ha descriure sobre el descriptor correcte

**read fd[0];**

- si hi ha algun descriptor d'escriptura sobre una pipe buia es bloquejant
- retorna 0 si tots els descriptors d'escriptura son tancats (EOF)

**write fd[1];**

- Sobre pipe sense lectors envia SIGPIPE i retorna -1 (erro=EPIPE)
- pipe té mida màxima PIPE\_BUF → write sobre pipe plena bloqueja

Exemples:

Pipe pare a fill

```
main(){
    int fd[2];
    char c;
    char s[5] = "abcd";
    pipe(fd);
    switch(fork()){
        case 0: close(fd[1]);
            while(read(fd[0], &c, sizeof(char)) > 0)
                write(1, &c, sizeof(char));
            close(fd[0]);
            break;
        default: write(fd[1], s, strlen(s));
            close(fd[0]); close(fd[1]);
    }
}
```

IMPORTANT TANCAR ELS CANALS

Pipe fill a pare

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

usuari

procés 1

pipe(fdp)

fork()

procés 2

pipe(fdp)

fork()

sistema

TC

0 std in

1 std out

2 std err

3 fdp[0]

4 fdp[1]

TFO

inodes

En aquest exemple no es tanquen els canals de pipe →  
es genera un bloqueig encara que hagi acabat

usuari

procés 1

pipe(fdp)

fork()

close(fdp[1])

procés 2

pipe(fdp)

fork()

close(fdp[0])

sistema

TC

0 std in

1 std out

2 std err

3 fdp[0]

4 fdp[1]

TFO

inodes

Es crea una pipe entre pare i fill o cada proceso tanca  
el canal de la pipe que no fa servir

Exemple Pipe entre fills i redirecció: p36

```
int fd[2] ....
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]);
        execv(programa2, \programa2, NULL);
    } } else { // FILL 1
        close(1); dup(fd[1]);
        close(fd[0]); close(fd[1]);
        execv(programa1, \programa1, NULL);
    }
```

Crea pipe entre fills i redirecciona les seves

entrades i sortida estàndard.

després substitueix la memòria dels fills per  
dos altres programes

El resultat programa1 escriu la sortida a la  
pipe que llegeix programa2 per la seva  
entrada estàndar

### 3.10 Sockets

La comunicació entre processos per pipe només amb processos emparentats entre si

Quan els processos no estan emparentats → utilitzar pipes amb nom o sockets

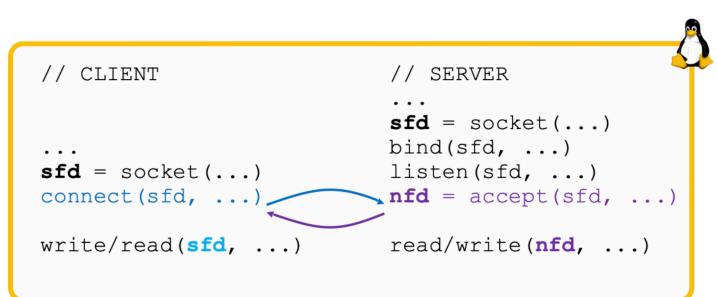
Sockets permeten comunicar processos en màquines diferents → comunicació a través de la xarxa → només establir la connexió i fer transferència de dades (comunicació)

Socket és un dispositiu de xarxa que implementa el protocol IP → implementació similar a la pipe però en operacions específiques i té un sol canal bidireccional de lectura/escriptura

arquitectura de tipus client-servidor:

- servidor: proceso escolta la xarxa fins detectar la trucada d'un client
- client: demana la connexió

el servidor l'accepta i queda establert un canal bidireccional al qual es comunicaran (mateixes crides del read i write)



El servidor s'identifica pel nom o adreça IP de la màquina i número de port

- número del port: distingir entre diferents serveis que ofereix la mateixa màquina (1024, i serveis definits per l'usuari (qualsevol número de port > 1024))

#### 3.10.2 Servidor

Té 4 fases:

##### 1. Inicialització del socket.

indica el tipus de socket (i.e. Internet: AF\_INET per comunicació a través de la xarxa) protocol de comunicació (i.e. TCP: SOCK\_STREAM) que volem fer servir (crida socket).

El protocol TCP estableix un circuit virtual entre client i servidor que facilita la comunicació posterior. L'alternativa, que no es descriu aquí, és utilitzar el protocol UDP amb "datagrames" (SOCK\_DGRAM) en que cada paquet de dades ha d'especificar el destinatari.

La crida socket retorna el descriptor de fitxer o canal (sfd al exemple 3.21) pel qual el servidor escolta per rebre peticions de connexió. Amb la crida bind li donem un "nom" al socket (adreça IP i port que escolta). I finalment el proces servidor expressa que vol rebre/"escoltar" (crida listen) peticions de connexió (provinents de la part/procés client). La crida listen defineix paràmetres com ara quantes peticions pendents puc tenir (backlog queue).

##### 2. Acceptar connexions

Aquesta crida bloqueja el procés servidor fins que algun client es connecta i, un cop establerta la connexió, retorna un nou descriptor de canal (nfd) a través del qual es pot fer la comunicació. A continuació, si el servidor és concurrent, cada vegada que es connecta un client (després de cada execució d'accept) el codi del servidor crea un fill (fork) per atendre al nou client.

Fixeu-vos que es convenient que el pare tanqui nfd cada vegada, i el fill tanca sfd. El procés pare pot acceptar més clients a través de sfd i cada procés fill pot comunicar-se amb un client diferent a través de la seva còpia de nfd.

### 3. Comunicació amb el client.

El servidor es comunica amb el client a través de nfd, un canal bidireccional en el que escrivim (write) o llegim (read) de la manera habitual. Perquè la comunicació flueixi, el codi del client i el servidor han de ser recíprocs de manera que quan un escriu l'altre llegeixi, i viceversa. Dit en altres paraules, la comunicació entre client i servidor ha de seguir un protocol (en aquest cas, un protocol que haurem definit nosaltres en dissenyar la nostra aplicació).

### 4. Finalitzacions.

Dependrà del protocol establert qui (client o servidor) finalitza la comunicació. Sigui com sigui s'indicara tancant el socket sfd.

#### 3.10.3 Client

Té 3 fases:

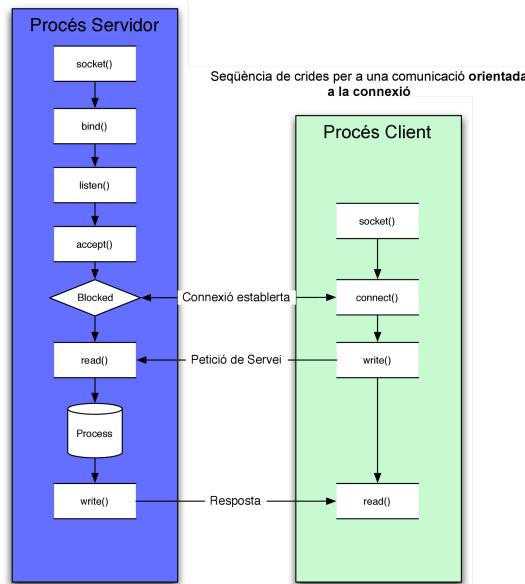
1. Inicialització del socket i connexió al servidor.

Inicialitzem el socket indicant el tipus de comunicació (i.e. Internet/TCP) tal i com hem fet al codi del servidor amb la crida socket. La crida socket ens retorna el descriptor de fitxer que farem servir per a la comunicació amb el servidor sfd. Amb la crida connect indico l'adreça del servidor al que em vull connectar (IP, port). Quan retorno de la crida connect ja està establerta la connexió amb el servidor i podem iniciar la comunicació

2. Comunicació amb el servidor.

Funciona igual que a la part del servidor crides read i write sobre el canal sfd seguint el protocol establert.

3. Finalitzacions. Igual que al servidor



### 3.11 Dispositius específics

#### **Consola:**

Consola és la combinació de teclat i pantalla

Per indicar el fi de fitxer des del teclat (EOF) POSIX estableix per conveni que s'indica amb la combinació de tecles Ctrl+D.

El sistema manté per cada consola, un buffer intern on guarda bytes/ caràcters teclejats fins al Return (CR), això permet esborrar caràcters abans que siguin tractats..

POSIX defineix tecles especials que ens permeten desplaçament: Ctrl+H (esborrar un caràcter), Ctrl+U (esborrar una línia)... depenen del controlador (device driver)

Pot tenir també un buffer d'escriptura per cada consola i implementar funcions especials del tipus pujar N línies etc., de fet cada controlador pot implementar la consola tan complicada com vulgui.

La consola constitueix per defecte l'entrada i sortida estàndard d'un procés (si no s'han redireccionat)

**`while ((n = read(0, buff, N)) > 0) write(1, buff, n);`**

Lectura de teclat (o consola) el buffer guarda caràcters fins que es pulsa CR.

Read es bloquejant (N bytes o EOF) Ctrl+D acaba la lectura amb el que hi hagi

L'escriptura del write - no bloqueja mai- escriu bloc de caràcters i potser esperar a CR

#### **Operacions ioctl i fcntl:**

Crida d'escapament per fer ajustos dependents d'un dispositiu ioctl (Linux) permet que una aplicació accedeixi qualsevol funcionalitat pugui ser implementada per un **device driver**

**`int ioctl (int fd, cmd [,ptr])`**

- fd: descriptor del dispositiu
- cmd: comanda a executar (depèn dispositiu)
- ptr: apuntador a una estructura de dades que depèn del dispositiu

La crida fcntl permet modificar el comportament d'un canal (file descriptor)

**`int fcntl(int fd, cmd [, args])`**

- fd: descriptor del dispositiu
- cmd: comanda a executar (depèn dispositiu)
- args: arguments de la comanda

Modificar el fet que el canal es mantingui obert en executar una crida al sistema tipus exec

#### **Pipe:**

S'implementa com un buffer temporal gestionat pel SO

- En la lectura: bloqueig a l'espera de dades, si hi ha un possible escriptor. Si no hi ha cap escriptor genera EOF.
- En la lectura: Bloqueig si pipe plena. Excepte si no hi ha possible lector que genera un SIGPIPE

#### **Socket:**

La comunicació és bidireccional i cal establir i implementar un protocol de comunicació

Qualsevol de les parts (servidor/client) pot tancar el seu socket i si no hi ha més info a llegir la crida read retorna erro amb codi d'error (erro) que indica la situació (man socket, man read)

## Tema 4: Estructura del Sistema de fitxers: VFS

### 4.1 El Disc

El disc dispositiu d'E/S permet emmagatzemament de dades de forma permanent.

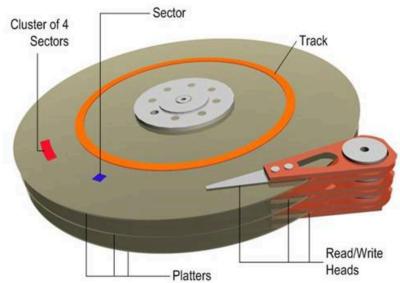
Pot ser un o més plats/discs rígids units per units eix

Cada plat té dues cares-> cada cara =>pista que es divideixen en porcions de mida fitxa =>sectors

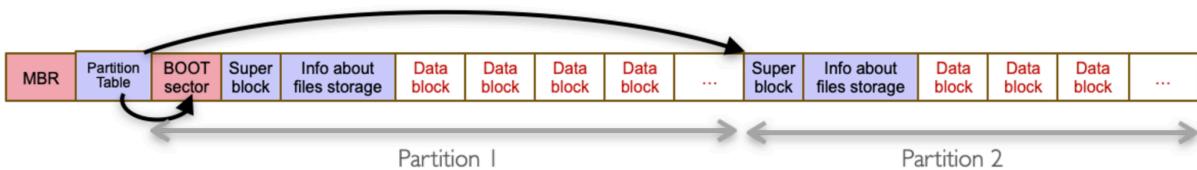
Hi ha un cap+al que s'ha de situar on es vol realitzar la Lectura/Escriptura

Els sectors son de la mateixa mida

UNIX pot gestionar un o més discs físics<- poden contenir un o més sistemes de fitxers



Un disc es particiona-> i cada partió es formateja en un sistema de fitxers determinat que divideix el disc en blocs lògics <- porcions de la mateixa mida (Blocs amb N sectors físics)



- Master Boot Record (MBR)

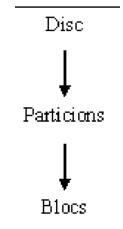
espai que sempre es troba situat al primer sector (512 bytes) del disc reservat per

- bootloaders: busca per la taula de particions quina és la partió activa que ha de bootar i copiar el seu sector a memòria
- taula de particions

- Bloc de boot Boot sector: Indica on es troba el codi a executar quan s'inicia la màquina. En particions no de boot, aquesta part no conté cap informació útil.
- Superbloc: conté informació sobre el Sistema de Fitxers aquesta partió, n'hi pot haver més d'una per disc, en cal una per partió Conte
  - o informació sobre l'estructura del disc: mida total / mida del bloc
  - o informació sobre fitxers (les dades): nombre de blocs, nombre de blocs lliures
- Informació sobre com s'emmagatzemen les dades dels fitxers.  
I.e. en el cas de ext2 es una llista d'inodes. El procés accedeix a un fitxer pel nom i aquestes estructures s'encarreguen de organitzar les dades d'aquest fitxer pel disc.
- Blocs de dades: Guarden el contingut propi d'un fitxer, les dades. Si un fitxer ocupa més d'un bloc, pot ser que no siguin contigus (en cas contrari, estaríem desaprofitant l'espai de disc). Un bloc només pot ser assignat a un fitxer, tant si l'ocupa totalment com si no. 1 bloc (Sistema Operatiu) correspon a N sectors (disc).

Diferents tipus de particions:

- Partició primària, s'utilitza per arrancar un sistema operatiu, poden haver-hi fins a 4 particions primàries per disc (aquest límit l'estableix el MBR).
- Partició extesa: es un contenidor de particions lògiques, i pot ser subdividida en varies particions lògiques
- Espai de Swap, pot ser una part d'una partició, Però va bé tenir una partició apart. En l'àrea de Swap es col·loquen dades que no caben a la memòria



#### 4.1.2 Estratègies de gestió del espai de disc:

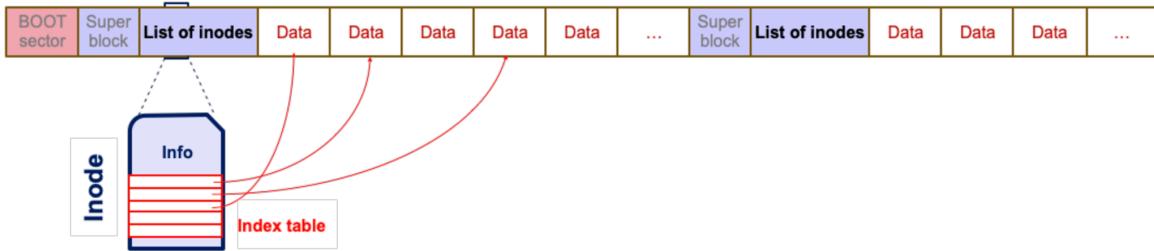
FS utilitza estrategia de gestió de l'espai de disc per aprofitar el màxim espai disponible per minimitzar el temps d'accés

- Assignació contigua: es fa servir normalment per write-once discs (CDs, DVDs...). Tots els blocs del arxiu s'assignen de manera consecutiva.  
Cal una única entrada per arxiu que indiqui el bloc inicial i la mida. Ens dona un accés eficient al disc però es necessita saber la mida dels fitxers a priori i produceix fragmentació externa.
- Assignació Enllaçada o enllacada en taula: En l'assignació encadenada cada bloc de dades reserva d'espai per un punter que indica quin és el següent bloc del fitxer.  
Cal una entrada per fitxer amb el punter al bloc inicial. Les avantages es que es suporta bé assignació dinàmica i no hi ha fragmentació externa. Els inconvenients es que per accedir al bloc i-essim cal recórrer tots els anteriors, per tant és terrible el rendiment per accessos directes (a una posició concreta del fitxer). La assignació encadenada per taula Linked List, un exemple és la FAT, que durant temps va fer servir Windows. Els punters enllacats de guardar-se en els blocs de dades es guarden tots junts en una taula que es diu File Allocation Table (FAT). És millor que l'anterior perquè no cal recórrer tota la taula, i també es pot replicar la taula per fer el sistema de fitxers més robust.  
L'inconvenient es que per discs grans la taula creix molt.
- Assignació indexada o multinivell. A través d'una taula d'indexació: inodes. Existeix un bloc que conté aquesta taula amb una entrada per cada fitxer amb punters als blocs. Un exemple concret es l'assignació indexada multinivell que permet fitxers més grans perquè crea una estructura jeràrquica de indexos. Es la que fa servir ext2 i es la que veurem.

Es comú utilitzar un bitmap de blocs lliures-> conte un bit => 0=bloc lliure 1=bloc ocupat

## 4.2 Sistema de Fitxers de Unix: ext2

Segueix una estructura que conté les metadades ⇒ inode  
conte les dades que s'organitzen per blocs. Es troba:



- Boot sector
- Superblock que conté informació sobre l'estrucció del disc: mida total / mida del bloc, nombre d' i-nodes, nombre d'i-nodes lliures i informació sobre els fitxers (les dades): nombre de blocs, nombre de blocs lliures
- Llista d'i-nodes: El FS s'encarrega de fer la traducció de nom de fitxer a a i-node. La llista d'i-nodes conté una entrada per cada fitxer del disc. Conte tota la informació sobre el fitxer, que inclou la taula d'indexes a blocs de dades.
- Blocs de dades

### 4.2.1 Inode

Estructura de dades utilitzada per sistema de fitxers per descriure un fitxer, conte:

- mida del fitxer: en bytes
- propietari del fitxer
- grup al que pertany
- proteccions
- tipus de fitxer: ordinari, directori, pipes, dispositius especials
- temps sobre l'accés al fitxer: data últim accés, modificació...
- nombre de links: nombre total de noms que el fitxer té en la jerarquia del sistema de fitxers. Un fitxer pot tenir associats diferents noms que corresponguin a diferents rutes però que accedeixin al mateix i-node i per tant a les mateixes dades.
- llista de blocs: array de 13 entrades de data bloc pointers (10 directes i 3 indirectes), ja que els blocs de dades poden ser no contigus.

### 4.2.2 Fitxer ordinari

L'inode de tipus fitxer => més genèric i segueix l'estrucció d'inode com apartat anterior  
L'única cosa es que no conte el nom del fitxer

#### 4.2.3 Directori

Un directori és un fitxer organitzat en forma de taula

Cada entrada conté un nom de fitxer-> 14 bytes  
i número de inode ->16 bytes

Es guarda en un bloc de dades - es mapeja com un fitxer ordinari i tmb pel seu i-node  
Ocupa només un bloc de dades - però si ocupa més funciona igual que un fitxer ordinari

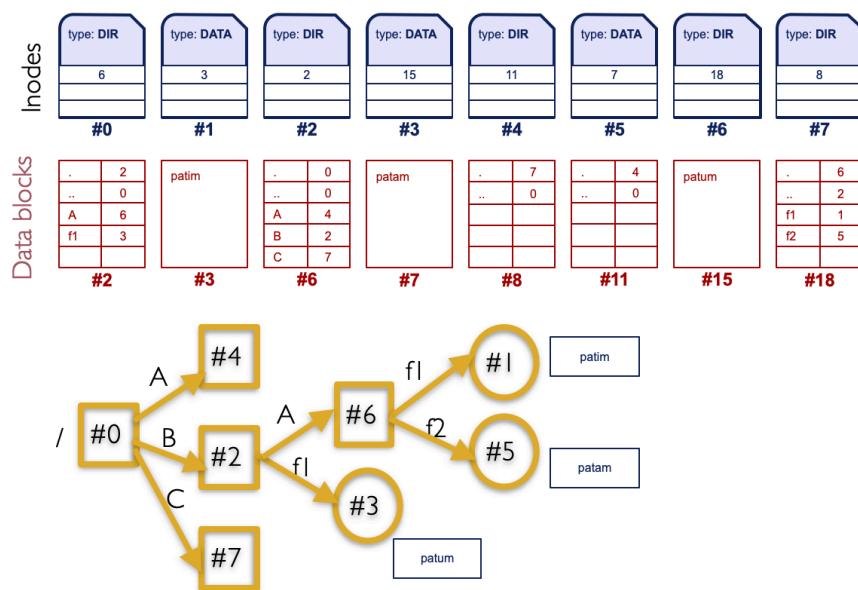
El contingut del bloc de dades es la taula que conté la correspondència nom de fitxer i número de inode associat → mínim dues entrades a taula ( . ) directori actual

( .. ) directori taula

El directori inode corresponent al directori arrel o root "/" es el número 0.

En el root el (..) apunta a ell mateix (en comptes del pare, perquè no en té).

Un mateix i-node pot ser referenciat per diferents noms (links), això es pot implementar com diferents i-nodes tipus directori amb noms que apunten al mateix inode (hard link).



#### 4.2.4 Dispositius

En el cas dels dispositius l'inode conté:

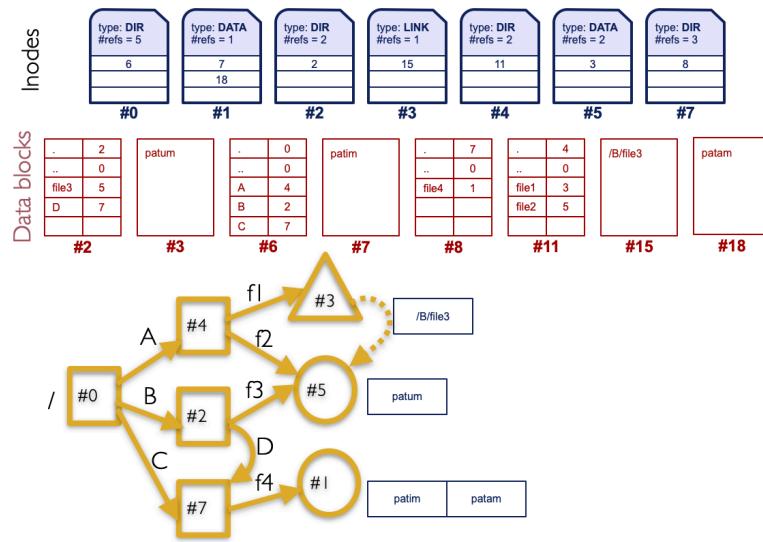
- El tipus de dispositiu char/block
- El major number: indica el tipus de dispositiu (disc, terminal, impressora)
- El minor number: número unitat dins del dispositiu
- No té associats blocs de dades

El major/minor permeten l'accés a una taula on es troben les adreces de les funcions per accedir-hi proporcionades pel Device Driver

#### 4.2.5 Links

Per links entrem a referències a un mateix fitxer:

- Hard links: crea una entrada nova al directori que apunta a un inode existent.  
No té un inode específic sinó que augmenta el contador de links del inode que apunta.  
No té blocs de dades associat // (.) i (..) son hard links
- Soft links: (symbolic link) Fitxer que conté el nom(path) d'un altre fitxer/directorii.  
El inode conté el tipus: links i la resta de camps de fitxer ordinari i bloc de dades que conté el path del fitxer que apunta



Quants accessos al disc calen fer per:

`fd=open("/A/file1",O_RDWR);`

El Open: i-node arrel (0) + Bloc 6 + i-node 4 + Bloc 11 + i-node 3 + Bloc 15 + i-node arrel 0 + bloc 6 + i-node 2 + Bloc 2 + i-node 5 = 11 accessos

`Iseek(fd,0,SEEK_END);`

`read(fd,buf,100);`

El Iseek no fa accessos al disc perquè el punter de l/e es troba a la TFO. El read tampoc fa accessos a disc en aquest cas, perquè el Iseek ens ha posicionat a final del fitxer.

#### 4.3 Permisos Fitxers: Access Control Lists

El sistema de Fitxers permet assignar diferents permisos als fitxer, es pot establir diferents nivells d'accés segons quin usuari accedeix al fitxer i quina operació es vulgui fer:

A Unix hi ha tres tipus d'usuaris: propietat, el grup i resta d'usuaris  
tres tipus d'accés: llegir (Read), escriure (Write) i executar (eXecute)

Els permisos del fitxer es donen en el moment de creació i es poden canviar

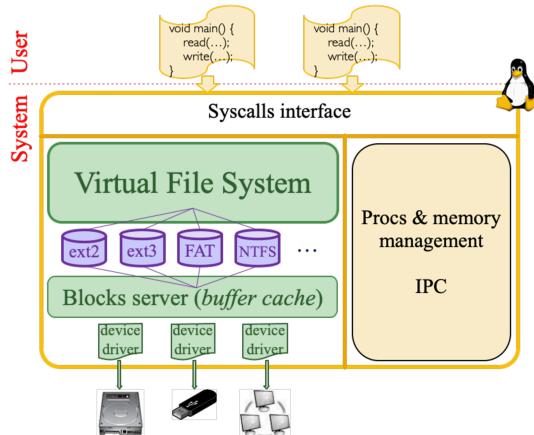
Els 9 bits: 3 permisos del propietari del fitxer //3 pel grup //3 pels altres usuaris

Determinen en aquest ordre de permisos. READ/WRITE/EXECUTE

User	R	W	X		%ls -la sortida.txt -rw-r--r-- 1 montse staff 21 20 Oct 13:38 sortida.txt
Owner	1	1	1	7	% chmod 761 sortida.txt
Group	1	1	0	6	% ls -la sortida.txt
Others	0	0	1	1	-rwxrwx---x 1 montse staff 21 20 Oct 13:38 sortida.txt

#### 4.4. The virtual File System (VFS)

El sistema de Fitxers Virtual (VFS) es una abstracció sobre el sistema de fitxers específic



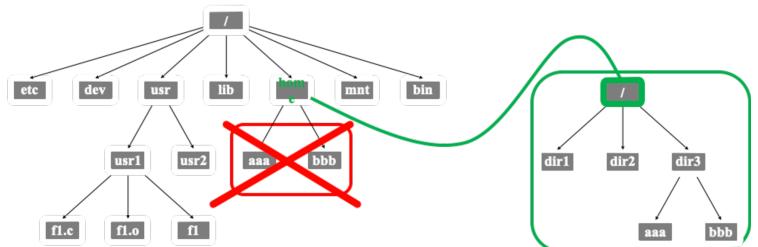
- El sistema de Fitxers FS:  
informació en disc, no volàtil, per emmagatzemar i organitzar fitxers segons format predeterminat.

Diferents FS ofereixen diferents opcions, atributs, nivells de seguretat ... i és independent de la interfície d'usuari i de la implementació del Sistema de Fitxers Virtual (VFS).

- El Sistema de Fitxers Virtual VFS és part del SO, informació que es guarda en memòria, per accedir i manipular els fitxers en temps d'Execució (run-time) desde un procés Conte una copia de la informació en disc (metadata) per accelerar el temps d'accés

##### 4.4.1 Mounting a File Sistem

Des del sistema podem accedir a diferents dispositius/particions cadascun té el seu propi format, sistema de fitxers i estructura de directoris



Per accedir a un Sistema de Fitxers primer ha de montar:

Existeix un dispositiu arrel que monta a la "/" del Sistema de Fitxers

La resta de dispositius d'emmagatzematge es poden montar a qualsevol directori del FS  
Amb la comanda mount incloem el dispositiu/partició a la FS perquè sigui accessible a través d'un directori → torna a l'inici (arrel inicial) per arribar al directori que es vol  
**mount -t ext2 /dev/hda1 /home umount /dev/hda1**

##### 4.4.2 Estructures de Linux/UNIX

Gestionar els dispositius E/S el SO fa unes estructures:

- **Taula de Canals:** es l'abstracció a nivell virtual, permet interacció entre els processos i l'e/s. N'hi ha una per procés.
- **Taula de Fitxers oberts:** Conte informació sobre cada obertura de dispositiu (fitxer) del sistema (i els seus accessos). Un mateix fitxer pot aparèixer més d'un cop si ha estat obert amb diferent mode etc... N'hi ha una per tot el SO. La crida "lseek" només implica actualitzar el punter de l/e en l'entrada corresponent sense necessitat d'accendir a disc.
- **Cache de Directoris (Dentry)** Manté en memòria entrades de directoris (nom de fitxer en el directori) que s'han fet servir anteriorment per accelerar el processat de rutes (path). Conte punters a inodes en memòria (vnodes)

- **Taula d'inodes:** manté en memòria una copia de cada inode que s'està utilitzant. Ja hem vist la informació que conté un inode en detall. Es una cache de inodes en ús. Una copia de l'objecte inode més informació necessària en temps d'execució com el número de referències. De fet els inodes en memòria, s'anomenen vnodes.
- **Buffer cache:** Es una caché de blocs de dades, unificada per tots els accessos a dispositiu. Optimitza el temps d'accés i facilita la compartició. Similar a la Dentry que es per entrades de directori però la Buffer cache es per blocs de dades.

#### 4.4.3 Open i el VFS

Les crides al sistema modifiquen i/o accedeixen a les estructures de VFS i necessiten acceder o modificar el disc (FS)

La crida open agafa com a paràmetre un nom de fitxer(path) i accedeix a tota la seqüència de inodes i blocs de dades de tipus directori fins arribar al inode que correspon al fitxer. Pel camí va afegint inodes que accedeix a la taula d'inodes (IT) ← si no hi eren es perquè es el primer cop que s'accedeixen.

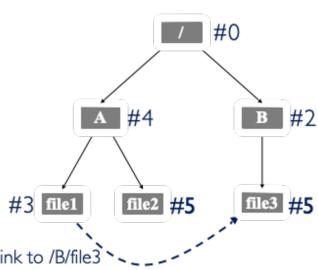
Un open modifica:

- La Taula de Canals (TC) del procés: s'afegeix una entrada
- La Taula de Fitxers Oberts del sistema: s'afegeix una entrada
- La cache de Directoris (Dentry) i la Taula d'inodes en memòria: afegeix totes les entrades corresponents si accedim per primer cop i no hi eren.

Amb les caches ens podem estalviar la majoria dels open sinó ens el podem estalviar per posar a la cache i seran reutilitzats al futur.

Fitxer nou (opció CREATE o crida create) al afegir una nova línia en el bloc de dades corresponent al directori ha d'anar el nou fitxer implica:

- Crea un nou inode (i escriure'l al disc)
- Modificar el bloc de dades del directori corresponent
- Modificar el inode del directori corresponent (data de darrera modificació, mida)
- Modificar el super bloc, la llista de inodes lliures, ja que n'haurem ocupat un.

Conta el número d'accisos al dist que generen les cries open	
	<pre>open(VA/file1", O_RDONLY) i-node 0 + data bloc + i-node 4 + data bloc + inode 3 (it's a link!) + data bloc + inode 0 + data bloc + inode 2 + data bloc + inode 5 = 11 accessos</pre> <pre>open(VA/file2", O_RDONLY); i-node 0 + data bloc + i-node 2 + data bloc + i-node 5 = 5 accessos</pre>
Si fem servir caches (Dentry i buffer cache )	Pel primer open de file1 afegim a la Dentry els directoris a mida que hi anem accedint, per això el segon cop que em trobo al directori arrel no cal accedir al inode 0 i el bloc de dades perquè ja les tinc al Dentry. Per tant els accessos que em calen son: i-node 0 + data bloc + i-node 4 +

	<p>data bloc + inode 3 (it's a link!) + data bloc + inode 2 + data bloc + inode 5 = 9 accessos</p> <p>Pel segon open de file2, si assumim que hem fet el open anterior primer i se'n ha afegit el camí a la Dentry, no ens cal cap accés a disc.</p>
	<ul style="list-style-type: none"> <li>- Pel primer open de file1: (a) Afegim inode 5 a la Taula d'inodes; (b) Objectes Dentry pels directoris / i A s'afegeixen a la Dentry cache; (c) Es crea nova entrada a la TFO i (d) S'ocupa la primera entrada lliure de la TDVO (o TC) del procés</li> <li>- Pel segon open de file2, si assumim que hem fet el open anterior primer, ja se'n ha afegit el camí a la Dentry i el inode 5 a la taula d'inodes. Per tant ens cal: (a) nova entrada a la TFO i (b) Ocupar la primera entrada lliure de la TDVO del procés</li> </ul> <p>Per últim, veiem que passaria si creem un nou fitxer. P.e.  <code>open("/A/file4", O_RDWR   O_CREATE, 777);</code></p> <p>A la Dentry cache ja trobem el camí (path) /A/. Ens cal:</p> <ul style="list-style-type: none"> <li>- Crear un nou inode (pel fitxer nou file4) i afegir-lo a la taula d'inodes.</li> <li>- Nova entrada a la TFO i ocupa entrada a la TDVO del procés</li> <li>- Es modifica el bloc de dades del directori A per incloure el nou fitxer</li> <li>- Es modifica l'inode del directori A per actualitzar la mida etc ..</li> <li>- Es modifica la taula d'inodes lliures (perquè n'hem fet servir un) que es troba al superbloc</li> </ul>

#### 4.4.4. Read i el VFS

La crida a sistema read sobre fitxer llegeix blocs de dades del disc, si no son a la Buffer cache. Per tant genera un o més accessos a disc, depenent de la mida i el offset. També depèn de si els blocs que cal accedir estan sent apuntats per indexació directe o indirecte en el inode.

#### 4.4.5 Write i el VFS

La crida a sistema write sobre fitxer, escriu blocs de dades a la Buffer cache (si n'hi ha!). Més tard (periòdicament), blocs de dades de la Buffer cache s'escriuen de nou al disc.

Assumint que hem d'escriure N blocs (dependrà de la mida de dades a escriure i del offset), s'ha de fer especial atenció al primer i últim blocs. Si l'offset inicial no es al inici del bloc, aquestes dades des de l'inici fins al offset han de ser llegides abans d'escriure. Si l'offset final no es al final del bloc, passa el mateix.

Com en el cas del read el nombre d'accessos també depèn de si els blocs de dades s'apunten amb apuntadors directes o indexos indirectes en el inode.

#### 4.4.6 Esborrar un fitxer en el VFS

Els passos que el SO pren alhora d'esborrar un fitxer del Sistema de Fitxers depèn del tipus de fitxer:

- Fitxer ordinari: Decrementa el numero de referencies al inode i si es 0, esborra el inode i els blocs de dades associats. Esborra l'entrada del directori corresponent a aquest fitxer.
- Directori: Alguns VFS comproven si el directori es buit i avisen l'usuari o donen un error; sinó s'esborra recursivament tots els fitxers i directoris que conte. Decrementa el numero de referencies i si es 0 s'esborra el inode i els blocs de dades associats. Esborra l'entrada del directori corresponent (del pare).
- Soft link: Decrementa el numero de referencies si 0 esborra inode i els blocs de dades associats. NO esborra el fitxer apuntat pel soft link. Si esborra l'entrada del directori corresponent.

Per altres fitxers especials nomes s'esborra el inode

#### 4.4.7 Consideracions finals

La presència de links en el Sistema de Fitxers transforma el arbre de directoris en un graf. Si es un graf cíclic, el SO ha de controlar bucles infinites durant els accessos, backups etc.

Existeixen diferents estratègies per controlar aquests grafs cíclics:

Els bucles creats per soft-links són més fàcils de tractar. Per exemple, el procés de backup no segueix cap soft-link.

Els bucles creats per hard-links són molt difícils. Per tant, de vegades es prohibeix crear un hard link si aquest causa un bucle, sinó cal implementar algun mecanisme per detectar aquests bucles.

- Taula de Canals: abstracció a nivell virtual interacció entre els processos i l'e/s. N'hi ha una per procés.
- Taula de Fitxers oberts: informació sobre cada obertura de dispositiu (fitxer) del sistema (i els seus accessos). Un mateix fitxer pot aparèixer més d'un cop si ha estat obert amb diferent mode etc... N'hi ha una per tot el SO. La crida "lseek" només implica actualitzar el punter de l/e en l'entrada corresponent sense necessitat d'accendir a disc.
- Cache de Directoris (Dentry) Manté en memòria entrades de directoris (nom de fitxer en el directori) que s'han fet servir anteriorment per accelerar el processat de rutes (path). Conte punters a inodes en memòria (vnodes)
- Taula d'inodes: manté en memòria una copia de cada inode que s'està utilitzant. conte un inode en detall. Es una cache de inodes en ús. Una copia de l'objecte inode més informació necessària en temps d'execució com el número de referències. De fet els inodes en memòria, s'anomenen vnodes.
- Buffer cache: Es una caché de blocs de dades, unificada per tots els accessos a dispositiu. S'utilitza per optimitzar el temps d'accés i facilitar la compartició. Similar a la Dentry que es per entrades de directori però la Buffer cache es per blocs de dades.

#### 4.4.3 Open i el VFS

Les crides del sistema modifiquen i/o accedeixen a les estructures en el VFS i poden accedir i modificar el disc (FS)

Crida open → pren un nom de fitxer (path) com a paràmetre → accedeix a tota la seq de inodes i blocs de dades de tipus directori fins arribar al ultim inode corresponent al fitxer

Pel camí va afegint els inodes que accedeix a la taula d'inodes (IT)

Un open modifica:

- La Taula de Canals (TC) del procés: s'afegeix una entrada
- La Taula de Fitxers Oberts del sistema: s'afegeix una entrada
- La cache de Directoris (Dentry) i la Taula d'inodes en memòria: afegeix totes les entrades corresponents si accedim per primer cop i no hi eren.

Com que tenim les caches, la majoria d'accessos a disc del open ens els podem estalviar, o sinó ens els podem estalviar els afegirem a la cache i seran reutilitzats en futurs accessos.

Si el fitxer que obrim es nou (fem servir l'opció de CREATE o bé la crida create), al afegir una nova línia en el bloc de dades corresponent al directori on ha d'anar el nou fitxer, això implica:

- Crea un nou inode (i escriure'l al disc)
- Modificar el bloc de dades del directori corresponent

- Modificar el inode del directori corresponent (data de darrera modificació, mida)
- I també modificar el super bloc, la llista de inodes lliures, ja que n'haurem ocupat un

	<pre>open(VA/file1", O_RDONLY); i-node 0 + data bloc + i-node 4 + data bloc + inode 3 (it's a link!) + data bloc + inode 0 + data bloc + inode 2 + data bloc + inode 5 = 11 accessos</pre> <pre>open(VA/file2", O_RDONLY); i-node 0 + data bloc + i-node 2 + data bloc + i-node 5 = 5 accessos</pre>
--	---

Amb la cache (Dentry i la buffer cache)

	<pre>open(VA/file1", O_RDONLY); afegim a la Dentry els directoris a mida que hi anem accedint, per això el segon cop que em trobo al directori arrel no cal accedir al inode 0 i el bloc de dades perquè ja les tinc al Dentry. Per tant els accessos que em calen son: i-node 0 + data bloc + i-node 4 + data bloc + inode 3 (it's a link!) + data bloc + inode 2 + data bloc + inode 5 = 9 accessos</pre> <pre>open(VA/file2", O_RDONLY); si assumim que hem fet el open anterior primer i se'n ha afegit el camí a la Dentry, no ens cal cap accés a disc</pre>
--	---

Estructures que s'accedeixen o es modifiquen:

Pel primer open de file1:

- Afegim inode 5 a la Taula d'inodes
- Objectes Dentry pels directoris / A s'afegeixen a la Dentry cache;
- Es crea nova entrada a la TFO
- S'ocupa la primera entrada lliure de la TDVO (o TC) del procés
  
- Pel segon open de file2, si assumim que hem fet el open anterior primer, ja se'n ha afegit el camí a la Dentry i el inode 5 a la taula d'inodes. Per tant ens cal:
  - nova entrada a la TFO
  - Ocupar la primera entrada lliure de la TDVO del procés

Si es crees un nou fitxer:

<pre>open("/A/file4", O_RDWR   O_CREATE, 777);</pre>	<p>Crear un nou inode (pel fitxer nou file4) i afegir-lo a la taula d'inodes.</p> <p>Nova entrada a la TFO i ocupa entrada a la TDVO del procés</p> <p>Es modifica el bloc de dades del directori A per incloure el nou fitxer</p> <p>Es modifica l'inode del directori A per actualitzar la mida etc ..</p> <p>Es modifica la taula d'inodes lliures (perquè n'hem fet servir un) que es troba al superbloc</p>
--	--

#### 4.4.4. Read i el VFS

La crida a sistema read sobre fitxer llegeix blocs de dades del disc, si no son a la Buffer cache. Per tant genera un o més accessos a disc, dependent de la mida i el offset. També depèn de si els blocs que cal accedir estan sent apuntats per indexació directe o indirecte en el inode.

#### 4.4.5 Write i el VFS

La crida a sistema write sobre fitxer, escriu blocs de dades a la Buffer cache (si n'hi ha!). Més tard (periòdicament), blocs de dades de la Buffer cache s'escriuen de nou al disc.

Assumint que hem d'escriure N blocs (dependrà de la mida de dades a escriure i del offset), s'ha de fer especial atenció al primer i últim blocs. Si l'offset inicial no es al inici del bloc, aquestes dades des de l'inici fins al offset han de ser llegides abans d'escriure. Si offset final no es al final del bloc, passa el mateix.

Com en el cas del read el nombre d'accessos també depèn de si els blocs de dades s'apunten amb apuntadors directes o indexos indirectes en el inode.

#### 4.4.6 Esborrar un fitxer en el VFS

Els passos que el SO pren alhora d'esborrar un fitxer del Sistema de Fitxers depèn del tipus de fitxer:

- Fitxer ordinari: Decrementa el número de referències al inode i si es 0, esborra el inode i els blocs de dades associats. Esborra l'entrada del directori corresponent a aquest fitxer.
- Directori: Alguns VFS comproven si el directori és buit i avisen l'usuari o donen un error; sinó s'esborra recursivament tots els fitxers i directoris que conté. Decrementa el número de referències i si es 0 s'esborra el inode i els blocs de dades associats. Esborra l'entrada del directori corresponent (del pare).
- Soft link: Decrementa el número de referencia si 0 esborra inode i els blocs de dades associats. NO esborra el fitxer apuntat pel soft link. Si esborra l'entrada del directori corresponent.
- Per altres fitxers especials només s'esborra el inode

#### 4.4.7 Consideracions finals

La presència de links en el Sistema de Fitxers transforma l'arbre de directoris en un graf. Si es un grafo cíclic, el SO ha de controlar bucles infinites durant els accessos, backups etc.

Existeixen diferents estratègies per controlar aquests grafs cíclics:

- Els bucles creats per soft-links son més fàcils de tractar. Per exemple, el procés de backup no segueix cap soft-link.
- Els bucles creats per hard-links son molt difícils. Per tant, de vegades es prohibeix crear un hard link si aquest causa un bucle, sinó cal implementar algun mecanisme per detectar aquests bucles.

## Tema 5: Gestió de memòria:

### 5.1 Conceptes:

La memòria d'un ordinador es compartida per a tots els processos → garantizar la confidencialitat i la integritat de les dades x cada procés

Cada procés disposa d'un/varis conjunts d'adreces continguts independents a la resta de processos i mida suficientment gran per la necessitat del procés → SO ofereix mem virtual als processos

#### 5.1.2 Memòria física x memòria lògica

La cpu no entra directament a memòria/registres - les dades dels programes s'han de carregar en memòria per poder referenciar-se  
per carregar un programa → reservar memòria, escriure el programa en l'espai i executar

- L'adreça lògica: és la referència generada per la CPU.
- L'adreça física: és la posició en memòria

#### 5.1.3 Espai d'adreces

- Espai lògic del processador: amplada del bus d'adreces del processador. Si el bus d'adreces és de 64 bits, la mida de l'espai lògic del processador = posicions de memòria.
- Espai lògic del procés: subconjunt de l'anterior indica quina porció esta ocupada procés.
- Espai físic: quantitat de memòria instal·lada en una màquina. L'espai físic d'un procés és el conjunt d'adreces físiques associades al espai d'adreces lògiques del procés.

La correspondència entre adreces lògiques i físiques

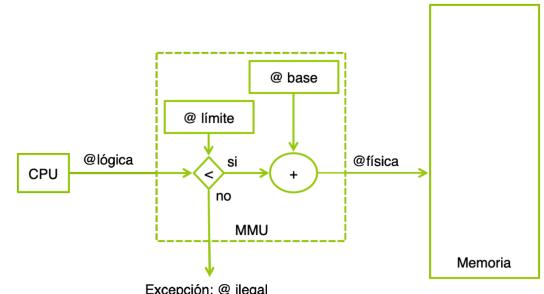
- Fixa, és a dir l'espai d'adreces lògic coincideix amb l'espai d'adreces físic.
- Mecanisme de traducció d'adreces: el processador genera direccions dintre de l'espai lògic del processador (direccions lògiques). A continuació, un hardware anomenat MMU (Memory Management Unit), que ha estat configurat/inicialitzat pel Sistema Operatiu s'encarrega de:
  1. Validar que la direcció lògica pertany a l'espai lògic del procés implementa la protecció dels espais d'adreces assignats a cada procés.
  2. Traduir la direcció lògica a la posició de l'espai físic (direcció física) on realment es guarda la informació.
  3. Accedir a la direcció de memòria física on es guarda la dada
  4. Cada accés a memòria generat pel processador, la MMU efectua una traducció dinàmica (en temps d'execució).

#### 5.1.5 Assignació de memòria

Diferents estratègies d'assignació de memòria:

1. Traducció estàtica en temps de compilació: determina en temps de compilació.  
L'adreça física coincideix amb la lògica perquè en execució no hi ha traducció d'adreces.  
És un esquema d'ubicació estàtic en un mateix fitxer executable sempre tindrà les mateixes posicions en memòria assignades. Convenient en alguns casos per exemple: per implementar un esquema de memòria compartida sobre un supercomputador amb memòria físicament distribuïda.

2. Traducció estàtica en temps de càrrega: temps de càrrega, quan el procés és carregat en memòria se li assigna l'espai d'adreces que és el mateix durant tota l'execució del programa, l'adreça física coincideix amb la lògica no traducció en temps d'execució.
3. Traducció dinàmica d'adreces en temps d'execució: l'adreça física diferent a la lògica ja que la traducció en temps d'execució i els processos poden canviar de posició en memòria durant la seva execució sense modificar el seu espai lògic d'adreces. Aquí distingim dos casos:
  - a. Espai d'adreces contigu. La MMU té un registre amb l'adreça base del procés, L'espai d'adreces es contigu, les adreces lògiques són relatives a l'inici del programa i l'adreça física es calcula sumant l'adreça base a la lògica
  - b. Espai d'adreces no contigu



#### 5.1.6 Protecció d'adreces

la MMU detecta adreces lògiques que no es poden traduir i genera una excepció que el SO gestionarà de la manera que convingui dependent del tipus d'excepció. El SO és per tant el responsable de configurar la MMU de forma adequada i de gestionar les excepcions. Com que l'espai d'adreces és contigu i les adreces relatives al inici, només cal garantir que el procés no accedeixi a més memòria de la que li ha estat assignada

#### 5.1.7 Mida del espai lògic del procés

SO també ha de tenir en compte si el procés cap a memòria es física o no → funció de la mida:

- Espai lògic de procés < memòria física:  
MMU ha de fer una tasca de comprovació de validesa de les direccions lògiques i de traducció. Si cada procés té associat un únic conjunt d'adreces lògiques, que la direcció lògica inicial del conjunt d'adreces és la direcció 0, que cada conjunt d'adreces té una mida definida, que coneixem la direcció de memòria física on es guarda la primera posició de memòria (direcció base) i que la memòria física es gestiona contiguament, Esquema de reubicació dinàmica.  
Per exemple: si el registre mida conté el valor 1000 i el registre base conté el valor 20000, el rang de direccions lògiques [0...999] serà traduït al rang de direccions físiques [20000...20999]. Intentar traduir qualsevol altra direcció lògica provocarà una excepció. Per tant, si la MMU està ben programada un procés només podrà accedir al rang d'adreces que li corresponguin. Si canvi de context, el SO modificar el valor registres.
- Espai lògic de procés > memòria física  
Oferir espais lògics de procés més grans que la memòria física cal utilitzar un àrea d'emmagatzemament secundari al disc → àrea de swap.  
La MMU ha de controlar quina part de l'espai lògic està a memòria física i quina a l'àrea de swap. Quan s'intenta accedir a una direcció lògica que físicament es troba a l'àrea de swap, la MMU generarà una excepció del tipus "fallada de pàgina", on la seva rutina d'atenció intenta portar a memòria física la data sol·licitada, re-executarà la instrucció que ha provocat l'excepció. La implementació es basa en paginar l'espai lògic (dividir-lo en porcions de la mateixa mida) i utilitzar les pàgines com a unitat mínima d'informació a nivell de gestió de memòria.

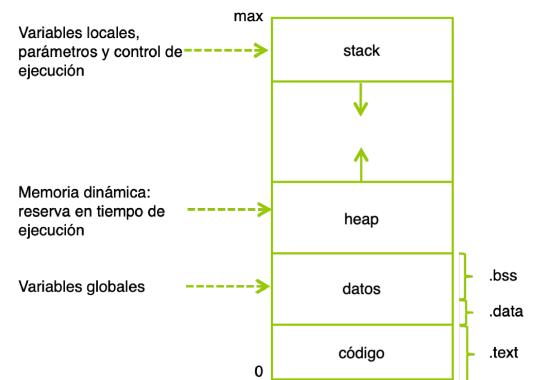
### 5.1.8 Estructura fitxer executable

defineix a la capçalera del fitxer, on podem trobar diferents seccions: tipus, mida i posició binari

Podeu provar d'executar: objdump - hprograma.

Diferents formats de fitxer executable → més estès en sistemes POSIX és el ELF (Executable and Linkable Format). Ho podeu veure amb la comanda file

Algunas secciones por defecto de un ejecutable ELF	
.text	código
.data	datos inicializados
.bss	datos sin valor inicial
.debug	información de debug
.comment	información de control
.dynamic	información para enlace dinámico
.init	@ primera instrucción a ejecutar



El SO carregar el programa en memòria per executar-lo ha de preparar l'esquema del procés en memòria lògica, és el que anomenem imatge en memòria del procés

Part que carrega els programes a memòria: Program Loader

- Interpretar el format del fitxer executable
- Preparar l'estructura del procés en memòria lògica
- Carregar seccions del programa del disc cap a memòria física. assignació de memòria física al procés i configurar la MMU amb les adreces traducció i protecció d'adreces
- Actualitzar les estructures de dades associades al procés
- Inicialitzar el PC (Program Counter) amb la primera instrucció a executar del programa

Optimitzacions Per fer la càrrega més eficient hi ha dues optimitzacions comuns:

- Càrrega sota demanda: busca retardar la càrrega de trossos del programa fins que no es necessiten, p.e. una rutina no es carrega fins que es crida. no es carreguen funcions que no es cuiden mai (p.e. les de gestió d'errors). S'accelera el procés de càrrega perquè només carrega una part del programa. falta un mecanisme que detecti si les routines estan o no carregades en memòria.
- Llibreries compartides i enllaços dinàmics: Les llibreries poden ser estàtiques (.a) o dinàmiques (.so). Quan enllacem (linkem) el nostre programa amb una llibreria dinàmica res retraca l'enllaç fins al moment de l'execució. Amb aquesta estratègia estalviem espai en disc, espai en memòria. També facilita l'actualització dels programes per a que facin ús de les noves versions de les llibreries del sistema ja que no cal regenerar l'executable. El mecanisme conté el codi d'una rutina stub que comprova si algun procés ja ha arreglat sinó la càrrega la llibreria. substitueix la crida a ella mateix per la crida a la rutina de llibreria compartida.

### 5.1.11 Memòria dinàmica

Quan el SO assigna una certa quantitat de memòria al procés de compilació no es pot estendre o modificar en temps d'execució ⇒ assignació o allocation estatica de memòria te certes límitacions:

- no s'adapta bé a mida de les estructures de dades depenen dels paràmetres
- fixant la mida en el temps de compilació no es adequat o assignem memòria de massa-desaprofitant memòria o ens quedem curts-error d'execució per falta memòria

El SO ofereix crides al sistema pq un procés pugui demanar més memòria en l'execució SO actualitza la seva estructura de dades amb la nova regió-retrasar el moment d'assignació d'adreces físiques fins la nova regió sigui implementada

Crides al sistema de Unix per demanar memòria dinàmica: **brk**, **mmap**, **malloc/free (C)**.

## 5.2 Serveis del SO relacionats per gestió de memòria

Responsabilitats del SO relacionades:

- Gestió de la memòria física
- Asignar memòria als processos: Càrrega del programa en memòria, o crides al sistema com fork, exec i altres
- Gestionar la MMU
- Gestionar les excepcions
- Suport a la traducció d'adreces
- Suport a la protecció i compartició de memòria entre processos
- Implementar optimitzacions de rendiment (COW, VM, prefetching)

Els serveis de la crides del sistema poden ser:

- Implicits: fork, exec
- Explícits: Memòria dinàmica, memòria compartida ...
- Gestió de la memòria i optimitzacions
  - Fork:

Service	System call
Create process	fork
Mute process	execvp
Heap management	brk / sbrk
[C-lang.] Allocate	malloc
[C-lang.] Free	free

So necessita allocatar memòria pel nou procés.

allocatar memòria per la imatge en memòria del procés -el nou procés és una copia del pare + actualitzar les estructures de dades amb informació del nou procés. Una optimització possible que alguns SO implementen és la compartició amb el procés pare de regions que no es poden modificar.

- Exec:

SO carrega el programa en memòria: Interpreta el format del fitxer executable, prepara les estructures del procés, càrrega de disc les seccions necessàries i actualitza les estructures de dades del procés referents a la memòria. SO que se'n ocupa és el Program Loader→ Carrega sota demanda i Llibreries compartides amb enllaç dinàmic.

- Gestió del heap: brk i sbrk

Funcions modifiquen el punter de l'àrea del Heap per incrementar o decrementar la memòria assignada al procés.

Funcions de baix nivell-conèixer i controlar bé la imatge del procés en memòria

**brk:** rep un punter indicant la nova adreça del punter a uninitialized data per sobre .bss

```
int brk( void *addr );
```

**sbrk:**rep un increment número de bytes i retorna un punter a la base del nou espai si es exitos sinó retorna -1 si error=> errno te un codi d'error indicant el motiu

```
void *sbrk( int increment );
```

- Malloc i free

Pertanyen a la llibreria de c-optimitzen i faciliten gestió del Heap

**malloc:** valida la nova regió i retorna adreça lògica inicial-la regió validada pot ser més gran a la demanda pel programador i la llibreria registra quina zona es la que es fa servir.

```
void *malloc( int size );
```

El Heap s'implementa amb una llista de zones de memòria ocupades i zones lliures→ el malloc busca per la llista una zona lliure suficientment gran per mirar de satisfer la petició sense recorre al sistema← sinó ho troba crida **brk/sbrk** per demanar més memòria al SO

```
void *free( void *ptr );
```

**free:** El programador ha d'allibrera la regió quan ja no sigui necessària

## Exemples:

```
ptr = sbrk(1000); //ptr = malloc(1000);
```

Diferències: Sbrk: creara/modifica el heap mentre que el malloc validara la nova regió i intenta satisfer la demanda de la memòria amb les zones lliures del heap (només estendre si és necessari)

```
ptr=malloc(1000); //for (i=0; i<10; i++) ptr=malloc(100);
```

Diferències: malloc(1000) creara i tindrem una zona de memòria lògica continguda que comença a l'adreça que retorna ptr si es fan 10 mallocs de 10 espais de mem segurament no seran continguts, pq es mirarà les zones lliures que hi hagin a memòria, i l'adreça que retorna ptr anirà canviant

## 5.3 Gestió interna de la Memòria

So necessita estructura de dades internes per poder fer la gestió:

- Gestió de la memòria física: estructures de dades per controlar la memòria lliure i la memòria que ja ha estat assignada
- Gestió de l'espai d'adreses dels processos: Cal mantenir una taula MMU per procés amb informació sobre la traducció d'adreses, les proteccions, els tipus d'accés etc. També cal mantenir informació sobre les regions assignades a cada procés (els límits), aquesta informació la trobem a la PCB (Process Control Block) El SO s'encarrega de actualitzar la MMU amb la informació del procés en execució.
- Gestió de les excepcions de memòria: que es fan servir en molts casos per implementar optimitzacions. P.e. fallada de pàgina.

### 5.3.1 Gestió de la memòria física

Els SO allocate i allibrera memòria dinàmica pels processos i les tasques del sistema, per maximitzar l'ús de la memòria física. Problema es la fragmentació de la memòria → queden zones que estan lliures i no es poden assignar a cap procés:

- Fragmentació interna: memòria assignada a un procés que no la necessita -no la fa servir- Es dóna el sistema de gestió de la memòria treballa amb particions de mida fixa.
- Fragmentació externa: memòria lliure, és a dir no assignada a cap procés, però que per alguna raó no es pot assignar. P.e. perquè és un espai no contigu, una solució seria compactar la memòria però és una operació costosa.

Estratègies d'assignació de memòria:

- Assignació/Allocació contigua: Un procés obté un espai d'adreses físic continu
  - Particions de mida fixa: on memòria física dividida en particions de la mateixa mida (de mida fixa). Es desaprofita memòria ja que la mida escollida no s'adaptarà a tots els processos del sistema (fragmentació interna).
  - particions de mida variable: El SO manté una llista de particions lliures, per cada procés selecciona una partició suficientment gran i l'ajusta a la mida del procés. En aquest cas es produeix fragmentació externa ja que pot ser que hi hagi suficient memòria lliure per un procés però no es pugui utilitzar perquè no està contigua en memòria.
- Assignació/allocació no contigua és l'estratègia més comú en sistemes de propòsit general. partitionar l'espai físic en particions "petites" de mida fixa (paginació) o variable (segmentació). El SO assigna al procés tantes particions com li calguin l'espai d'adreses físic del procés no és contigu en memòria. disminueix la fragmentació, augmenta la flexibilitat però també augmenta la granularitat en la gestió de memòria d'un procés i per tant la complexitat del SO i de la MMU. Veurem dos (o tres) esquemes:

- Paginació: particions fixes
- Segmentació: particions variables
- Esquemes combinats: p.e. segmentació paginada

### 5.3.2 Paginació

Idea mida de les particions es fixa, l'espai d'adreces lògiques del procés es divideix en pàgines en la memòria fixa, dividida amb la mateixa mida: marc (frame)

Algoritme d'assignació de mem busca per cada pàgina un procés un marc lliure

El SO té una llista de marcs lliures (bitmap of free frames) si el procés demana més memòria el SO comprova si hi ha mem a les pàgines ja assignades, així actualitza el límit del procés sinó troba un nou marc

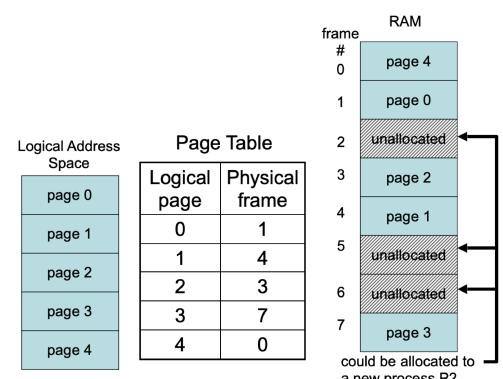
Quan un procés acaba la seva execució torna els marcs assignats a la llista de marcs lliures Tot es fa a nivell de pàgina (com unitat) facilita implementació i la compartició de memòria entre processos

### Taula de pàgines

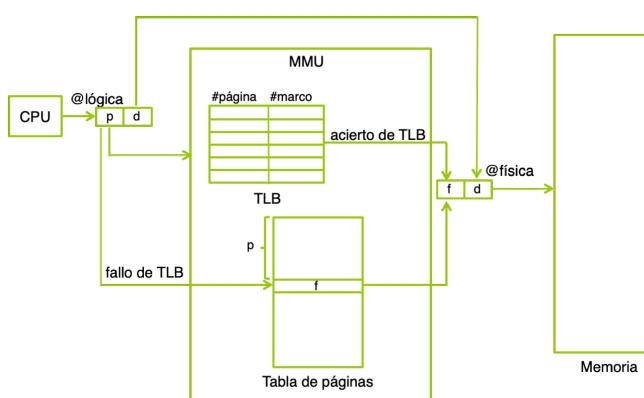
esquema de paginació necessari una taula de pàgines => una taula de pàgines per procés

te entrada per cada pàgina i manté la info a nivell de la pàgina

So guardarse en memòria i el SO guarda per cada procés l'adreça base de la taula



Accelerar el mecanisme de traducció d'adreces el processador actuals tenen TLB (Translation Lookaside Buffer) que és un cache de les pàgines actives



MMU implementa un esquema en paginació  
Cada adreça lògica dividida dos parts:

- bits més pes-(p) num pàgina (indexar la taula de pàgines i recuperar marc corresponent)
- bits menys pes-(d) desplaçament dins de la pàgina

p-depèn de l'espai lògic del processador/mida de la pàgina

d-depèn de la mida de la pàgina

f-depèn de la mida de la memòria física

Un processador amb un bus d'adreces de 32 bits, ens dona un espai d'adreces del processador de  $2^{32}$  bytes = 4Gb ( $1\text{Gb} = 2^{30}$ ). Si suposem una mida típica de pàgina de 4kb ( $2^{12}$ ).

Quantes pàgines tenim:  $2^{32}$  bytes espai /  $2^{12}$  mida pagina ==  $2^{20}$  pàgines

L'adreça lògica sera de:

- p=20 bits major pes indexar les pàgines (si hi han  $2^{20}$  pàgines)
- d=10 bits menys pes determinar offset (dins de la pàgina) (32-10 bus d'adreces)

Un processador amb un espai d'adreces de 128Mb:  $2^{27}$  necessita 27 bits per adreçar-lo. Si suposem una mida de pagina de 1kb ( $2^{10}$ ).

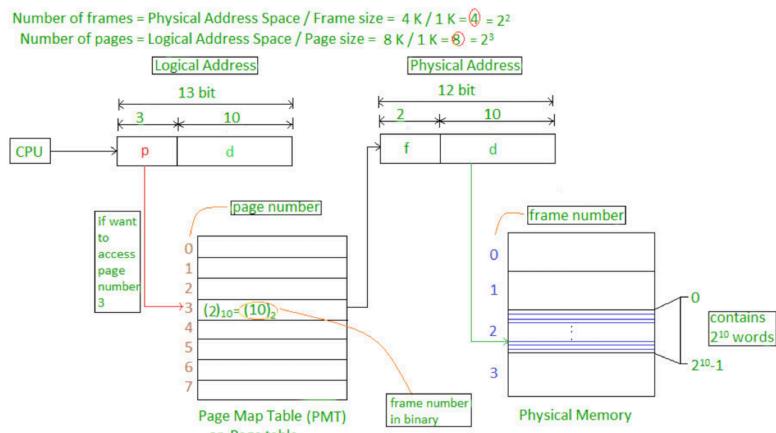
Quantes pàgines tenim:  $2^{27}$  bytes espai /  $2^{10}$  mida pagina ==  $2^{17}$  pàgines

L'adreça lògica serà de:

- $p=17$  bits major pes indexar les pàgines (si hi han  $2^{17}$  pàgines)
- $d=10$  bits menys pes determinar offset(dins de la pàgina) (27-17 bus d'adreces)

Considerem també la memòria física:

- Adreça física de 12 bits :
- Espai d'adreces físic 4Kb
- Adreça lògica de 13 bits :
- Espai d'adreces lògic 8Kb
- Mida de la pàgina 1K words

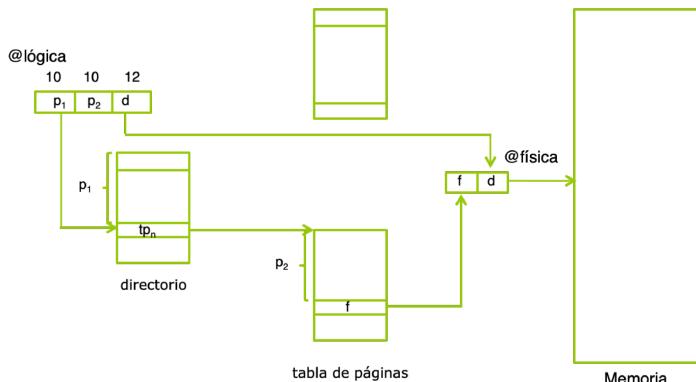


### Rellevància de la mida d'una pàgina:

la mida de la pàgina determina varis factors que influencien el funcionament del sistema.

Ha de ser una potència de 2. Normalment els sistemes generals fan servir es de 4KB ==  $2^{12}$

	Espacio lógico de procesador	Número de páginas	Tamaño TP	Determina la granularitat i menys fragmentació interna- però llavors les pàgines seran molt grans Pàgines grans donen fragmentació interna o redueixen la mida de la taula de pàgines
Bus de 32 bits	$2^{32}$	$2^{20}$	4MB	
Bus de 64 bits	$2^{64}$	$2^{52}$	4PB	



Solucionar aquests problemes existeixen taules multinivell: 2 nivells

1r-directori de pàgines 1024 entrades, cada entrada conte l'adreça base de la taula de pàgines

2n-son les taules seccionades-cada ocupa una pàgina(1024 entrades) i conte zona traducció (4MB)-(Mida de una pagina es de 4Kb)

Mida típica pàgina -4 a 8kb

Permet entrades 32bits i mida pàgina de 4Kb  $\Rightarrow$  adreçar a  $3^2 \times 2^{12} \times 2^{44} = 16\text{TB}$

Fragmentació interna si:

Exemple 1: si el meu procés ocupa una mica més de 4Kb -p.e. 4097 bytes- cada pàgina és de 4Kb (4096 bytes) llavors el SO li ha d'allocatar dues pàgines (8Kb) i la major part de la segona (4095 bytes) no els fa servir produint així fragmentació interna.

Proteccions a nivell de pàgina: La unitat de gestió de la memòria del SO és la pàgina això implica que si determinades zones de memòria com per exemple el codi i les dades d'un procés tenen proteccions diferents, el codi generalment és `read_only` mentre que les dades han de poder-se modificar `rdwr`, si comparteixen pàgina les proteccions s'assignen a nivell de pàgina i per tant no podem definir-les, una solució senzilla és que codi i dades d'un procés allocation a pàgines diferents, tot i que això pot comportar fragmentació en alguns casos.

Exemple 2: el codi del meu procés és de 2K i les dades 2K però el SO els assignarà pàgines diferents per poder donar `read_only` al codi i a les dades `rdwr`, desperdigant així una pàgina sencera (2K del codi i 2K de les dades), i causant així fragmentació interna.

### 5.3.3 Segmentació

Consisteix dividir l'espai lògic del procés en segments que tenen en compte el contingut.  
Aproxima la gestió de la memòria a la visió de l'usuari. l'espai d'adreces logic es divideix de mida variable → segments → mínim un segment per codi /un per la pila/un per les dades  
Una adreça lògica consisteix un segment desplaçament dins del segment (gestiona mem fisica)  
Cada partició es d'una mida different

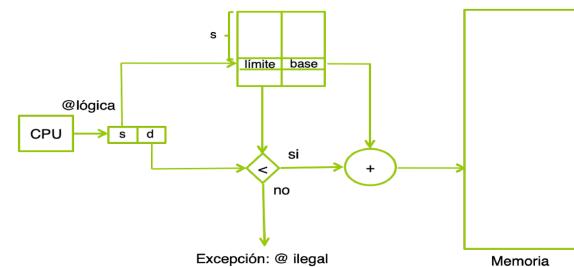
L'assignació de mem: per cada segment el SO busca partició on el segment hi càpiga, hi han possibles polítiques: first-fit, best-fit worst-fit, el SO selecciona la quantitat necessària pel segment i la resta continua amb les particions lliures

Pot produir fragmentació externa si les regions lliures que queden son massa petites per utilitzar  
Quan el procés acaba torna la porció assignada a la llista de regions lliures

### 5.3.4 MMU implementar un esquema basat en segmentació

MMU necessita mantenir una taula de segments i cada entrada de la taula conte info del segment

- l'adreça base del segment(on comença)// mida (mida variable)



### 5.3.6 Traducció d'adreces

Per hardware, la MMU tradueix les adreces lògiques → adreces físiques

Incialització: crear nou procés i durant la càrrega de l'executable en memòria el SO guarda la info sobre l'assignació de la memòria que se li hagi fet

Canvi de context: el procés abandona la CPU: si encara no ha acabat l'execució el SO emmagatzema la info necessària per reconfigurar la MMU quan torni a executar-se

Pel procés que passa a ocupar la CPU: es configura la MMU amb la info del procés que passa a executar-se

El SO s'encarrega de mantenir la MMU actualizada amb el procés actualment en execució. El SO reconfigura la MMU quan hi ha un canvi de context

### 5.3.7 Protecció d'adreses

L'espai d'adreses físic d'un procés ha d'estar protegit dels altres processos. La protecció d'adreses també es garanteix a través del hardware: la MMU valida les pàgines i els límits dins de la Taula de Pàgines. Permet permisos de Lectura,escriptura o execució per la pàgina.

### 5.3.8 Memòria Compartida

La compartició de memòria entre processos pot ser de diferents maneres:

- Dos processos poden compartir codi llibreries- codi no modificable i si els dos processos executen el mateix codi no cal carregar dos còpies a mem física
- Crides al sistema pq un procés crei una zona de memòria al seu espai lògic d'adreses per compartir i així permet mapejar aquell espai en el seu espai de memòria
- Es gestiona a nivell de pàgina i l'entrada corresponent trobem num de ref per marc Tota la mem que no es compartida -privada per un procés cap altre pot accedir

#### Threads:

Tot l'espai d'adreses d'un procés pot ser compartit entre threads o fluxes d'execució d'aquest Casi tot menys la pila, cada thread té na part privada on es reserva un espai per les variables locals, els paràmetres i el control de la seva execució.

Race condition: quan el resultat de l'execució del programa depèn de l'ordre en què s'alterni l'ús de la CPU entre els fluxes- pot donar resultats incoherents

La regió de codi que no es convenient fer un canvi de context- text de regió crítica

Cal executar aquestes regions de codi en exclusió mútua, garantir que no es produeix un canvi de context i que cap altre procés no pugui entrar en la regió crítica fins que el procés n'hagi sprit, si hi ha algun procés que vol entrar queda bloquejat, i fins que la regió quedí lliure no podrà entrar en la regió crítica.

SO ofereix mecanismes per garantir l'exclusió:

- Inici: si hi ha algún procés en la regió crítica el procés es bloqueja, sinó, en SO marca que hi ha algú a dins i permet que el procés continui (entrant a la regió crítica)
- Fi: Quan el procés surt de la regió crítica, el SO allibera aquesta regió, permetent així l'entrada a nous processos

Programador tota la responsabilitat que funcioni  
programa utilitzant programes→

```
init_mutex();
if (first) {
    first--;
    end_mutex();
    task1();
} else {
    end_mutex();
    task2();
}
```

```
init_mutex();
if (first) {
    first--;
    end_mutex();
    task1();
} else {
    end_mutex();
    task2();
}
```

## 5.4 Optimitzacions en la Gestió de la Memòria

### 5.4.1 Copy on Write COW

Fer una copia de memòria retrasa el moment de la copia mentre només es produexin accessos de lectura → evitar la copia física si els processos només fan servir la regió per llegir.

Implementació: Al fer la còpia → l'espai lògic del procés el SO marca la regió destí amb els permisos d'accés que hagi de tenir (reals).

la MMU el SO marca la regió destí i la regió font amb permisos de només lectura.

SO configura la MMU per associar a la regió destí les direccions físiques associades a la regió font.

⇒ Si un procés intenta accedir a una de les dues regions amb accés d'escritura la MMU generarà una excepció (perquè el SO la ha marcat amb permisos de només lectura) i el SO gestiona l'excepció fent la copia real.

Reserva direccions físiques noves per la regió destí, copia la informació, actualitza la MMU per ambdues regions i reinicia la instrucció que ha provocat l'excepció i que ara ja podrà completar l'accés. En un sistema basat en paginació el COW s'aplica a nivell de pàgina.