

El processos

Sistemes Operatius 1

Oliver Díaz / Lluís Garrido

Grau d'Enginyeria Informàtica

Els processos

- Al tema anterior, “El nucli”, hem vist els mecanismes que necessita un sistema operatiu per poder gestionar de forma fiable l'execució dels processos.
- En aquest tema ens centrarem en les **crides a sistema** que ens ofereix el sistema operatiu per crear processos, per gestionar la seva seguretat i en les eines bàsiques de comunicació entre processos.
- Explorarem un subconjunt de la interfície de programació dels sistemes UNIX, la base de Linux, MacOS i Android.

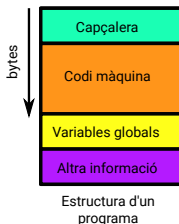
Organització

- 1 Què és un procés (recordatori)
- 2 Creació i gestió de processos
- 3 Gestió del context d'execució dels processos.

El concepte de procés

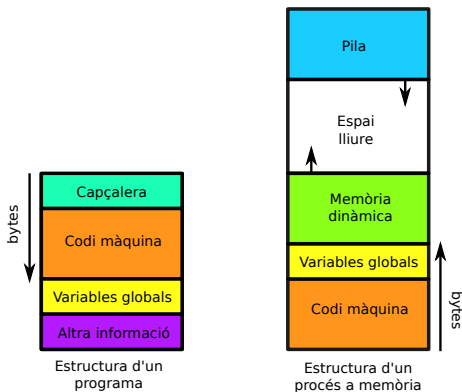
Aquest és el model de programació al qual estem acostumats

- 1 Es realitza un programa en un llenguatge d'alt nivell (per exemple, C).
- 2 S'utilitza un compilador per convertir el codi en instruccions màquina.
- 3 El compilador genera un codi executable del programa. El codi executable té una “estructura interna” coneguda pel sistema operatiu.
- 4 En executar el codi, el sistema operatiu crea el procés associat.



El concepte de procés

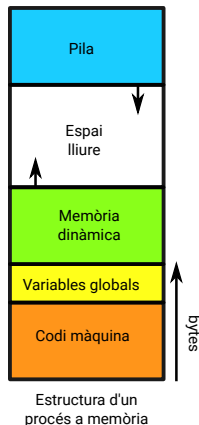
Un procés és un **programa en execució**, una **instància d'un programa**, de la mateixa forma que un objecte és una instància d'una classe a programació orientada a objectes.



El concepte de procés

Internament, el sistema operatiu manté, mitjançant el **Bloc de Control de Processos** (BCP, en anglès Process Control Block), una llista dels processos que s'executen . El BCP emmagatzema, per a cada procés,

- L'identificador de procés.
- Quin és l'usuari que l'executa.
- Quins privilegis té.
- Quines restriccions de seguretat té.
- En quina part de la memòria física resideix (memòria virtual).



El concepte de procés

Els ordinadors d'avui en dia poden fer moltes coses a la vegada

- Tot i que només hi hagi **una única CPU**, el sistema operatiu s'encarrega d'executar múltiples processos al mateix ordinador al mateix temps.
- El sistema operatiu ho aconsegueix executant cada procés només per unes desenes o centenars de milisegons. En acabar aquesta “**llesca de temps**” per al procés el sistema operatiu s'encarrega de canviar de procés.
- En cada instant de temps només s'executa un únic procés, però en un segon poden executar múltiples processos (inclús el mateix procés múltiples vegades), donant a l'usuari l'**il·lusió de paral·lelisme**.

Els ordinadors d'avui en dia poden fer moltes coses a la vegada

- En sistemes **multiprocessador** el paral·lelisme es real, tot i que el sistema operatiu hi aplica la mateixa idea: en un segon múltiples processos poden executar en una determinada CPU.
- El sistema operatiu utilitza algorismes específics per decidir quin procés s'ha d'executar en cada moment a cada CPU. És la base de la multiprogramació.

Veurem tots aquests detalls en el tema de planificació!

Ens preguntem quin tipus de crides a sistema (interfície) ha de proveir un sistema operatiu als processos. Entre altres coses,

- **Gestió de processos:** Pot un procés crear un altre procés? Pot un procés esperar que un altre procés acabi d'executar? Aturar o continuar l'execució d'un procés?
- **Entrada-sortida:** Com poden el processos comunicar-se amb dispositius connectats a l'ordinador? Poden els processos comunicar-se entre sí?
- Altres crides a sistema inclouen coses com la gestió de memòria, la gestió de la xarxa, etc. No hi entrarem en aquest tema.

Respecte la gestió de processos i l'entrada-sortida

- La funcionalitat està implementada a nivell de nucli i els processos hi poden accedir mitjançant crides a sistema. En total hi ha una dotzena de crides associades.
- Pel que fa sistemes UNIX, la interfície pràcticament no ha canviat respecte el seu disseny inicial del 1973 i continua essent molt utilitzada avui en dia. La interfície UNIX és senzilla, potent i portable. No ha sigut necessari canviar la interfície del sistema operatiu i, per tant, els desenvolupadors s'han pogut concentrar en desenvolupar aplicacions d'usuari.

La interfície de programació ha de ser segura en front a qualsevol ús maliciós.

- Antigament, en els sistemes de processament per lots, el nucli s'encarregava de crear els processos. Realment no hi havia problemes de “seguretat” en crear processos ja que era el sistema operatiu mateix qui ho controlava tot.
- A l'actualitat els mateixos processos poden crear i gestionar altres processos. Això ha permès un munt d'innovació, essent una de les primeres aplicacions l'interpret de comandes. Altres aplicacions que creen i gestionen processos són els gestors gràfics de finestres, els servidors web, els navegadors web, les bases de dades, els compiladors, els editors de text, etc. Cal gestionar la seguretat!

Permetre que un procés pugui crear altres processos té beneficis importants

- Un navegador web acostuma a fer servir aplicacions externes per tal de dibuixar una pàgina a pantalla. Això permet aconseguir funcionalitats molt complexes.
- El mateix passa amb el servidor web que és qui “entrega” la pàgina al navegador: abans d’entregar-la al navegador es poden invocar a processos externs per tal de formatar-la adequadament perquè pugui ser visualitzada al navegador.
- Als sistemes UNIX, a la línia de comandes, hi ha disponibles múltiples aplicacions que fan una tasca especialitzada. Les funcionalitats de les aplicacions es poden combinar fent servir la redirecció i les canonades per aconseguir funcionalitats molt potents.

Anem a veure a continuació les bases de la interfície de programació per a

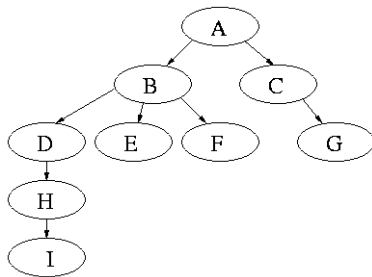
- 1 La gestió de processos
- 2 Gestió del context d'execució dels processos.

El procés que crea el procés s'anomena **pare** mentre que el procés que es crea s'anomena **fill**. Les tasques que ha de realitzar el nucli en crear un nou procés són:

- 1 Crear i inicialitzar el Bloc de Control de Processos.
- 2 Crear i inicialitzar l'espai d'adreces a memòria.
- 3 Carregar el programa a memòria.
- 4 Preparar el nucli per començar a executar al "main".

La gestió de processos

Atès que els processos poden crear altres processos es crea una **jerarquia de processos**. El sistema operatiu emmagatzema informació sobre aquesta jerarquia.



Podem veure la jerarquia de processos mitjançant aquesta instrucció a la línia de comandes

`$ pstree`

A Windows la funció que permet crear un procés s'anomena `CreateProcess`

```
// Start the child process
if (!CreateProcess(NULL, // No module name (use command line)
    argv[1],             // Command line
    NULL,                 // Process handle not inheritable
    NULL,                 // Thread handle not inheritable
    FALSE,                // Set handle inheritance to FALSE
    0,                    // No creation flags
    NULL,                 // Use parent's environment block
    NULL,                 // Use parent's starting directory
    &si,                   // Pointer to STARTUPINFO structure
    &pi)                   // Pointer to PROCESS_INFORMATION structure
)
```

Els primers dos arguments permeten especificar el programa a executar i els arguments a passar al main. Tota la resta estan associats a restringir l'entorn en què s'executa el procés fill.

Als sistemes UNIX s'utilitza un altre enfoc. En comptes d'utilitzar una única funció, s'utilitzen dues funcions: **fork** i **exec**

- La funció **fork** crea un nou procés, una còpia del procés pare. La funció **fork** retorna dues vegades, una pel pare i una altre pel fill. Pel pare, retorna el número que identifica al fill; pel fill retorna 0. Veure codi **fork.c**.

```
int main(void)
{
    int ret;

    ret = fork();

    if (ret == 0) { // fill
        printf("Soc el fill i el meu id es %d\n", getpid());
        return 0;
    } else { // pare
        printf("Soc el pare del proces %d\n", ret);
        return 0;
    }
}
```

El procés fill creat amb fork és independent del procés pare. Proveu d'executar el codi `fork-variables.c`.

```
int main(void)
{
    int ret, a;

    a = 1;

    ret = fork();

    if (ret == 0) { // fill
        printf("Soc el fill i el meu id es %d\n", getpid());
        printf("Fill a = %d\n", a); // s'imprimeix a = 1!
        return 0;
    } else { // pare
        a = 2;
        printf("Soc el pare del proces %d\n", ret);
        printf("Pare a = %d\n", a); // s'imprimeix a = 2!
        return 0;
    }
}
```

La gestió de processos a UNIX

Un cop creat un procés fill amb fork, podem executar un nou programa amb exec.

- La (família de) funcions **exec reemplacen la imatge del procés amb l'especificada a exec**. No es crea cap procés nou!
- La funció exec no retorna en acabar l'execució del programa especificat a exec.

```
int main(void)
{
    int ret;
    char *argv[3] = {"/usr/bin/ls", "-l", NULL};

    ret = fork();

    if (ret == 0) { // fill
        printf("Soc el fill i el meu id es %d\n", getpid());
        execv(argv[0], argv);
    } else { // pare
        printf("Soc el pare del proces %d\n", ret);
        return 0;
    }
}
```

Per què dues funcions en comptes d'una per crear processos?

- 1 En fer fork el procés fill hereta el context del pare (privilegis, fitxers oberts, ...).
- 2 Després del fork i abans de fer l'exec, es pot establir el **nou context** del programa a executar amb crides a sistema.

```
int main(void)
{
    int ret;
    char *argv[3] = {"/usr/bin/ls", "-l", NULL};

    ret = fork();

    if (ret == 0) { // fill
        printf("Soc el fill i el meu id es %d\n", getpid());
        // Aquí s'estableix el nou context del procés fill!
        // Es fan les crides necessàries abans de fer exec
        execv(argv[0], argv);
    } else { // pare
        printf("Soc el pare del procés %d\n", ret);
        return 0;
    }
}
```

Exemple: fork bomb

Què passarà si s'executa la següent aplicació coneguda com “fork-bomb”? Podeu dibuixar la jerarquia de processos que es crearan?

```
0 #include <stdio.h>
1 #include <unistd.h>
2
3 int main(void)
4 {
5     while (1) { fork(); }
6 }
7
```

Cal que el pare esperi que el procés fill finalitzi l'execució?

- A vegades és interessant que ho faci. Per exemple, és possible que el pare necessiti el resultat del fill per poder continuar l'execució. La funció **wait** permet fer-ho.

```
int main(void)
{
    int ret;
    char *argv[3] = {"/usr/bin/ls", "-l", NULL};

    ret = fork();

    if (ret == 0) { // fill
        printf("Soc el fill i el meu id es %d\n", getpid());
        execv(argv[0], argv);
    } else { // pare
        printf("Soc el pare del proces %d\n", ret);
        wait(NULL);
        printf("Torno a ser el pare un cop el fill ha acabat\n");
        return 0;
    }
}
```

Cal que el pare esperi que el procés fill finalitzi l'execució?

- En alguns contextos no és interessant. El fet de no posar wait permet que pare i fill s'executin “en paral·lel”. Pel cas de la línia de comandes això es pot especificar mitjançant un “&” al final de la línia. Per exemple, proveu d'executar¹

```
$ kwrite fork.c &  
$ kwrite while-infini.c
```

¹La primera instrucció s'executa en paral·lel amb la línia de comandes. El pare (la línia de comandes) no fa cap wait i la línia de comandes torna a aparèixer de seguida. La segona instrucció fa que el pare faci un wait perquè finalitzi el procés que s'executa. És per això que veurem que la línia de comandes no torna a aparèixer fins que sortim de l'aplicació executada.

Usualment el fork i l'exec apareixen “en parella”... té algun interès fer un fork sense un exec?

Sí, i tant! Al Google Chrome cada pestanya és un procés creat amb un fork².

- Cada procés s'executa en paral·lel amb la resta de processos. Això fa que les pestanyes es “carreguin” en paral·lel.
- Si a una de les pàgines hi ha un problema (un virus, per exemple) només fallarà aquella pestanya però no tot el navegador ja que els processos són independents entre sí.

A altres navegadors com el Firefox s'utilitza un altre mecanisme per obtenir el paral·lelisme: els fils. Els veurem més endavant i tenen les seves avantatges i desavantatges respecte el fork.

²A Windows no existeix el fork i per això s'utilitza una cua de processos que “esperen” que el procés principal els doni enllaços per processar.

Anem a veure a continuació les bases de la interfície de programació per a

- ① La gestió de processos
- ② Gestió del context d'execució dels processos.

Un procés pare, en crear un procés fill, pot controlar el **context del fill**:

- Els privilegis.
- Quin és el temps màxim d'execució
- Quina és la memòria màxima que pot ocupar
- Quina prioritat té per executar-se sobre la CPU respecte altres processos
- On s'envia tot el que s'imprimeix amb "printf" (i.e. la sortida)
- D'on rep el que es captura amb "scanf" (i.e. l'entrada)
- Etc.

Vegem un exemple en què establim el temps màxim d'execució del procés fill. La funció `setrlimit` permet fer-ho.

- Provar d'executar l'aplicació `while-infinite.c` directament des de línia de comandes. No s'aturarà mai!
- L'aplicació `fork-exec-setrlimit.c` executarà l'aplicació anterior limitant el temps màxim d'execució a 1 segon.

La funció `setrlimit` permet establir múltiples límits: temps de CPU, ús de memòria dinàmica, ús de la pila, nombre de fitxers que es poden obrir, etc.

Ens atrevim a executar el fork-bomb limitant algun recurs?
Provem-ho!

La línia de comandes de Unix ofereix una comanda equivalent, `ulimit`, que permet limitar els recursos assignats als processos que s'executen.

- Podem executar l'aplicació `while-infinet.c` limitant el temps màxim d'execució.
- Podem executar altres aplicacions limitant, per exemple, la mida màxima de la pila.
- I moltes coses més!

A més d'establir límits de privilegis sobre el procés fill, també es pot controlar

- On s'envia tot el que s'imprimeix amb “printf” (i.e. la sortida). A pantalla? A un altre procés? A un fitxer de disc? Per la xarxa?
- D'on rep el que es captura amb “scanf” (i.e. l'entrada). Del teclat? D'un altre procés? D'un fitxer de disc? De la xarxa?

Observar que el sistema operatiu ofereix protecció entre processos. Els processos no es poden “interferir” entre sí. Si es volen comunicar, necessiten permís del sistema operatiu.

Veiem a continuació tècniques bàsiques de comunicació interprocés: la redirecció i la canonada.

- On s'envia tot el que s'imprimeix amb “printf” (i.e. la sortida). A pantalla? A un altre procés? A un fitxer de disc? Per la xarxa?
- D'on rep el que es captura amb “scanf” (i.e. l'entrada). Del teclat? D'un altre procés? D'un fitxer de disc? De la xarxa?

Recordem primer la funció `printf`: és una crida a una llibreria de l'espai d'usuari

```
printf("El valor es %d\n", i);
```

Aquesta instrucció genera, a l'espai d'usuari, la cadena a imprimir. Un cop generada es fa la crida al sistema operatiu amb un `write`.

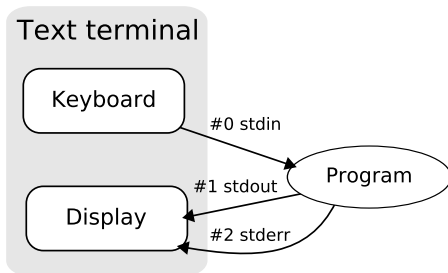
```
count = write(fd, vector, nbytes);
```

La funció `write`, disponible a una llibreria d'usuari, fa la crida a sistema. El primer paràmetre és el **descriptor de fitxer**, un sencer que identifica el dispositiu on escriure, i els altres dos fan referència al vector de bytes a escriure. Quin descriptor de fitxer s'utilitza en imprimir per pantalla? I per llegir de teclat?

La gestió del context: entrada-sortida

Tot procés, en crear-se pel sistema operatiu (sigui Unix o Windows), té creats 3 fitxers per defecte

- Descriptor **0**: conegut com “**entrada estàndard**”, està associat per defecte el teclat (es captura amb scanf).
- Descriptor **1**: conegut com “**sortida estàndard**”, està associat per defecte a la pantalla del terminal (s'imprimeix amb printf).
- Descriptor **2**: conegut com “**sortida d'error**”, està associat per defecte al terminal.



Les instruccions

```
i = 25;  
printf("El valor es %d\n", i);
```

Es “converteixen” en la següent crida a sistema

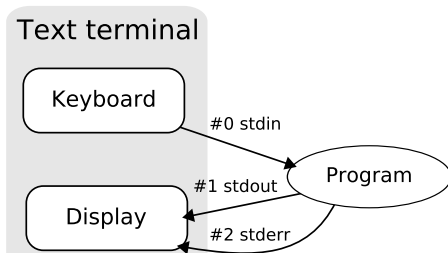
```
char *vector = "El valor es 25\n";  
int nbytes = 15;  
count = write(1, vector, nbytes);
```

Proveu-ho amb el codi `stdstreams.c`. Recordem que també tenim la utilitat `strace` per rastrejar les crides a sistema que fa una aplicació!

Entrada-sortida: redirecció sortida estàndard a fitxer

Per defecte el descriptor 1 (printf) està associat al dispositiu “pantalla”. Ara volem associar el descriptor 1 a un fitxer de disc.

- 1 Fer un fork. Després de fer el fork la situació del procés fill és la que es mostra a la figura.
- 2 S'obre el fitxer on volem que es bolqui tot allò que s'escriu al descriptor 1 (printf). Fem unes crides a sistema per associar el descriptor 1 al fitxer.
- 3 Executem el programa. Tot el que s'escriu al descriptor 1 s'escriu a disc!



Entrada-sortida: redirecció sortida estàndard a fitxer

Aquest és el codi C que ho permet fer, veure [fork-redirect-1.c](#).

```
int ret;
char *argv[3] = {"/usr/bin/ls", "-l", NULL};

ret = fork();

if (ret == 0) { // fill
    // obrim fitxer
    int fd = open("fitxer.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    dup2(fd, 1); // associem fd al descriptor 1
    close(fd); // tanquem fd
    execv(argv[0], argv); // executem programa
} else { // pare
    ...
}
```

Tot el que s'imprimeixi al descriptor 1 estarà associat a fd, el fitxer de disc, en comptes de la pantalla. Típicament es diu que “**es redirigeix la sortida estàndard de pantalla a un fitxer**”.

Proveu d'executar altres programes al fill, per exemple, [stdstreams.c](#).

Entrada-sortida: redirecció sortida estàndard a fitxer

La línia de comandes permet redirigir la sortida estàndard a un fitxer

```
$ ls -l > fitxer.txt
```

La línia de comandes interpreta la línia anterior i executa el codi de la transparència anterior. Proveu també amb

```
$ ./stdstreams > fitxer.txt
```

Una cosa similar passa amb

```
$ var=$(ls)
```

L'interpret de comandes redirigeix la sortida estàndard de la comanda `ls` (pex a un fitxer temporal) per després llegir-ho i assignar-ho a la variable `var`.

Entrada-sortida: redirecció sortida estàndard a fitxer

A l'exemple anterior capturem la sortida estàndard (descriptor de fitxer 1), però no la sortida d'error (descriptor de fitxer 2), que és on típicament s'imprimeixen els missatges d'error.

```
$ cd toto
```

```
bash: cd: toto: El fitxer o directori no existeix  
Proveu de redirigir la sortida estàndard
```

```
$ cd toto > fitxer.txt
```

```
bash: cd: toto: El fitxer o directori no existeix  
Podem capturar la sortida del descriptor 2 amb
```

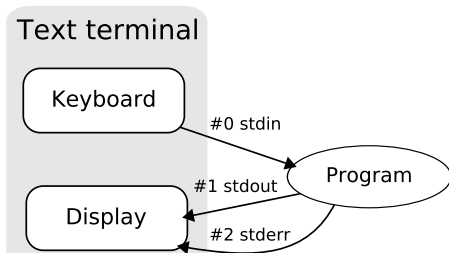
```
$ cd toto 2> fitxer.txt
```

Aquest comportament el trobem a altres aplicacions com el compilador, ...

Entrada-sortida: redirecció entrada estàndard des d'un fitxer

De la mateixa forma que hem redirigit la sortida estàndard a un fitxer, podem redirigir un fitxer a l'entrada estàndard. Volem associar el descriptor 0 (scanf) a un fitxer de disc. Passos:

- 1 Fer un fork. Després de fer el fork la situació del procés fill és la que es mostra a la figura.
- 2 S'obre el fitxer d'on volem que el descriptor 0 (scanf) agafi les dades.
- 3 Executem el programa. La funció scanf agafarà les dades de disc!



Entrada-sortida: redirecció entrada estàndard des d'un fitxer

Aquest és el codi C que ho permet fer, veure [fork-redirect-0.c](#).

```
int ret;
char *argv[2] = {"/scanf", NULL};

ret = fork();

if (ret == 0) { // fill
    // obrim fitxer
    int fd = open("linies.txt", O_RDONLY);
    dup2(fd, 0); // associem fd al descriptor 0
    close(fd); // tanquem fd
    execv(argv[0], argv); // executem programa
} else { // pare
    ...
}
```

Proveu d'executar primer l'aplicació [scanf.c](#) directament des de terminal. Després executeu-ho fent servir [fork-redirect-0.c](#).

Entrada-sortida: redirecció entrada estàndard des d'un fitxer

La línia de comandes permet redirigir un fitxer a l'entrada estàndard a un fitxer

```
$ ./scanf < linies.txt
```

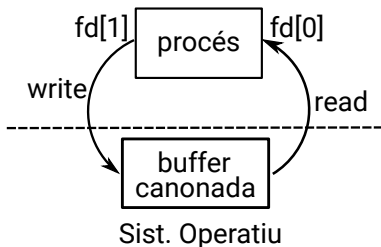
Inclús podem redirigir l'entrada i sortida estàndard de l'aplicació scanf. Què fa això?

```
$ ./scanf < linies.txt > output.txt
```


Entrada-sortida: canonades

La **canonada** es una forma bàsica de comunicació interprocés: l'objectiu és dirigir les dades que un procés A genera cap a un altre procés B. Com es fa això? Mitjançant una crida a sistema, **pipe**, que crea:

- Un buffer, al sistema operatiu, que gestiona la comunicació
- Dos descriptors de fitxers: un d'escriptura i un altre de lectura



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

int main(void)
{
    int fd[2];

    if (pipe(fd) == -1) {
        fprintf(stderr, "pipe not created");
        exit(1);
    }

    // fd[1] per escriptura
    // fd[0] per lectura

    return 0;
}
```

Podem enviar qualsevol tipus dada per una canonada. Exemple codi `pipe.c`³.

```
int main(void)
{
    int fd[2];
    char buf[30];

    if (pipe(fd) == -1) {
        fprintf(stderr, "pipe not created");
        exit(1);
    }

    printf("writing to file descriptor %d\n", fd[1]);
    write(fd[1], "test", 4);
    printf("reading from file descriptor %d\n", fd[0]);
    read(fd[0], buf, 4);
    buf[4] = 0;
    printf("read \"%s\"\n", buf);

    return 0;
}
```

³Atenció: en aquest exemple, abans d'imprimir la cadena rebuda cal posar un byte 0 al final de la cadena (al 5è byte) ja que printf imprimeix fins que troba el byte 0.

Entrada-sortida: canonades

Les canonades s'utilitzen per comunicar diferents processos. Els processos **han de tenir una relació pare-fill**.

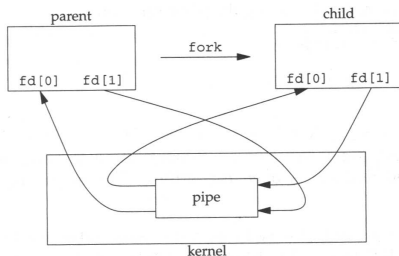
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int fd[2];

    pipe(fd);

    if (fork() == 0) { // child
        ...
    } else { // parent
        ...
    }

    return 0;
}
```



Per exemple: si el pare escriu a fd[1], el fill ho pot llegir de fd[0].

Veure codi `fork-pipe.c`. Es mostra el principi bàsic de funcionament⁴.

```
int main(void)
{
    int fd[2];
    char buf[30];

    pipe(fd);

    if (fork() == 0) { // child
        printf("child writing to file descriptor %d\n", fd[1]);
        write(fd[1], "test", 4);
        exit(0);
    } else { // parent
        printf("parent reading from file descriptor %d\n", fd[0]);
        read(fd[0], buf, 4);
        buf[4] = 0;
        printf("parent read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

⁴Per simplificar l'exemple, no s'inclou el codi per tancar les connexions de les canonades dels extrems que no es fan servir.

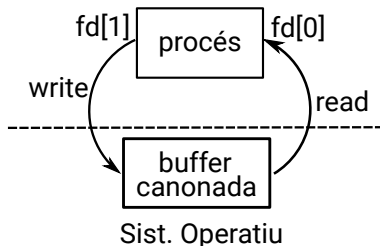
A l'exemple anterior

- Un procés pot anar introduint dades a la canonada i un altre procés pot anar-les llegint. No fa falta que el primer procés acabi perquè el segon pugui llegir les dades introduïdes a la canonada. És tracta d'un **flux de comunicació "contínua"** entre els dos processos.
- Al terminal (shell), per a una canonada, `fd[1]` s'associa a la sortida estàndard del primer procés, i `fd[0]` a l'entrada estàndard del segon procés. Per fer-ho cal fer servir les funcions `dup2` que hem vist abans, així com tancar els extrems de les canonades que no es fan servir.⁵

⁵L'estàndard POSIX requereix que el terminal esperi a la darrera comanda de la canonada abans de continuar. Això implica que la darrera comanda de la canonada sigui filla (directa) del terminal per poder fer el wait corresponent.

Alguns detalls tècnics

- El buffer és un **buffer intern** del sistema operatiu. Totes les dades passen per aquest buffer.
- El buffer és relativament **petit** (al voltant de 64KBytes)
- Si en **llegir** del buffer no hi ha cap dada, el procés que fa la crida es **bloqueja** fins que hi ha dades disponibles.
- Si en **escriure** al buffer aquest és **ple**, el procés que fa la crida es **bloqueja** fins que un altre procés el comença a buidar.



- Típicament les canonades s'utilitzen a la línia de comandes per combinar múltiples comandes “senzilles” i aconseguir una funcionalitat més complexa.

```
$ ls | ./scanf
```

la comanda anterior envia el que imprimeix “ls” per la sortida estàndard (descriptor 1) a l'entrada estàndard (descriptor 0) de l'aplicació scanf, veure codi `scanf.c`. És tracta d'un **flux de comunicació “contínua”**: mentre “ls” va imprimint i enviant per la canonada, scanf ho va llegint.

- Podem combinar les canonades i la redirecció com vulguem

```
$ ls | ./scanf > out.txt
```

la comanda anterior envia la sortida de scanf al fitxer `out.txt`.

Hem vist

- El sistema operatiu ofereix una interfície (crides a sistema) per crear processos. Els processos poden crear nous processos amb un context d'execució limitat.
- Els processos són, en sí, independents entre sí. El sistema operatiu ofereix eines perquè els processos es puguin comunicar entre sí. Dues de les eines “clàssiques” són la redirecció i les canonades.
- Al següent tema veurem altres eines que ens ofereix el sistema operatiu per fer comunicació inter-procés.