Comunicació interprocés : Sistemes Operatius I - Universitat de Barcelona

Lluis Garrido

1. Conceptes introductoris

Definició: Comunicació interprocés: Moltes vegades els processos han de comunicar-se entre si (enviar informació amb altres processos). El concepte de çomunicació interprocés"(*interprocess communication - IPC* s'aplica exclusivament a la comunicació entre processos (i no al fet que diferents parts d'un mateix procés es comuniqui entre sí).

Aquestes són algunes de les tècniques de comunicació interprocés més importants dins dels sistemes operatius, encara que no es una llista exhaustiva:

1. Canonades anónimes *pipe* i canonades amb nom *named pipe* o *FIFO pipe*.
2. Arxius (*files*)
3. Xarxa (*sockets*)
4. Senyals

Definició: POSIX: Del anglès, *Portable Operating System Interface* és un estàndard Unix per mantenir compatibilitat entre sistemes operatius.

Per exemple, un **fitxer** a POSIX pot fer referència a la xarxa, a disc, a qualsevol sistema de comunicació: tots es tracten 'igual" des d'un punt de vista de programador.

En un sistema operatiu tipus POSIX tots els fitxers oberts per un procés tenen associat un **descriptor de fitxer**. El descriptor de fitxer és, per als programadors, un sencer no negatiu.

Aquest sencer no negatiu és, en el context dels SO, un índex a un vector del procés que conté la informació dels fitxers oberts.

Per tant el sistema operatiu ens ofereix una sèrie de crides a sistema per poder proveir de comunicació interprocés:

1. Obrir una comunicació:
   a) Funció open: Serveix per obrir un fitxer de disc.
   b) Funció pipe: Serveix per crear una canonada.
   c) Funció mkfifo: Serveix per crear una canonada amb nom.
   d) Funció socket: Serveix per crear una connexió via xarxa (reomta)

e) I moltes més

totes aquestes funcions retornen el descriptor del fitxer (sencer no negatiu) que identifica el fitxer.

2. Per enviar (funció write) i rebre (funció read) informació a través d'un fitxer. Aquestes funcions tenen com a 1r argument el descriptor del fitxer.

3. Per tancar una comunicació (funció close). Aquesta funció té un únic argument: el descriptor del fitxer a tancar.

2. Canonades

Una de les primeres formes de comunicació interprocés és la **canonada**, conegut amb el nom de canonada anònima. La canonada s'utilitza en processos que tenen una relació **pare-fill**.

El funcionament s'ha vist al bloc anterior de *Processos* de la assignatura.

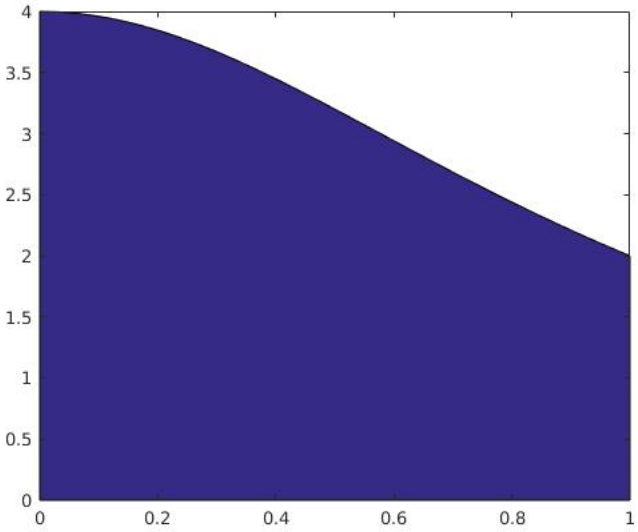
Ara es veurà un exemple el·laborat d'ús de les canonades:

Exemple: Càlcul del valor π

Per a calcular π utilitzarem la seva expressió integral següent:

$$\pi = \int_0^1 \frac{4}{1+x} dx$$

Utilitzant la definició d'integral Riemann, tenim que la integral es composa de diverses àrees rectangulars disjuntes dos a dos que sumades donen l'àreea de la corva, com es veu a la següent imatge:



Utilitzarem el següent codi per calcular la integral:

```
#define NUM_RECTS 100000000
int main()
{
int i;
double mid, height, width, sum = 0.0;
double area;
width = 1.0 / (double) NUM_RECTS;
for(i
for
= 0; i < NUM_RECTS; i++) {
mid = (i + 0.5) * width;
height = 4.0 / (1.0 + mid * mid);
sum += height;
}
area = width * sum;
printf("pi = %e\n", area);
return 0;
}
```

Utilitzant el codi proporcionat al temari de teoria, podem veure dues implementacions:

1. Implementació fent servir **1 sol procés** (calcul_pi_1proces.c)
2. Implementació fent servir **2 processos** i una canonada per comunicar el resultat parcial entre els processos (codi calcul_pi_ 2processos.c)

A l'exemple implementat per als 2 processos, el procés pare fa els rectangles senars i el procés fill els rectangles parells. El procés comunica al pare el seu resultat.

El temps d'execució amb 2 processos és menor que amb 1 procés: El sistema operatiu planifica cada processes en CPUs diferents si tenim un processador multinucli!

Nota: Normalment s'acostumen a fer servir fils per fer aquest tipus de tasques (i.e. computació paral·lela.)

Les **canonades anònimes tenen un gran desavantatge: permeten la comunicació interprocés només si els processos tenen relació de pare-fill**

Tanmateix, una **canonada amb nom** (*FIFO pipe*) combina les característiques d'un fitxer de disc i una canonada. És a dir:

- En obrir una canonada amb nom, se li associa un nom de fitxer de disc. Aquest fitxer funciona com una canonada. El SO decideix on s'emmagatzema aquest fitxer.
- Qualsevol procés pot obrir aquest fitxer per lectura o escriptura. Només cal que els processos facin servir el mateix nom.
- *Nota:* Windows implementa les canonades fent servir canonades amb nom sempre, no pot implementar canonades anònimes ja que no fa servir el for-exec.

Exemple d'us (codi fifo_write.c i fifo_read.c)

La funció mkfifo crea la canonada amb nom. La funció open obre la canonada (per lectura o escriptura).

Codi escriptor:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
int main()
{
int fd;
mkfifo("myfifo", PERM_FILE);
printf("Obrint pipe per escriptura\n");
fd = open("myfifo", O_WRONLY);
printf("Escrivint missatge\n");
write(fd, "test", 5);
printf("Tancant\n");
close(fd);
}
```

Codi Lector:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
int main()
{
char buf[20];
int fd;
printf("Obrint pipe per lectura\n");
fd = open("myfifo", O_RDONLY);
printf("Llegint missatge\n");
read(fd, buf, 5);
printf("Rebut %s, tancant.\n", buf);
close(fd);
}
```

3. Arxius (files)

Utilitzar arxius té els següents avantatges:

- El disc és un dispositiu al qual poden accedir tots els processos.
- El sistema operatiu ofereix funcions perquè un procés es pugi situar, en un fitxer, a la posició que vulgui. No cal doncs utilitzar l'arxiu com a FIFO, a diferència de les canonades.

Tanmateix, utilitzar arxius per a la comunicació interprocés també té els següents desavantatges:

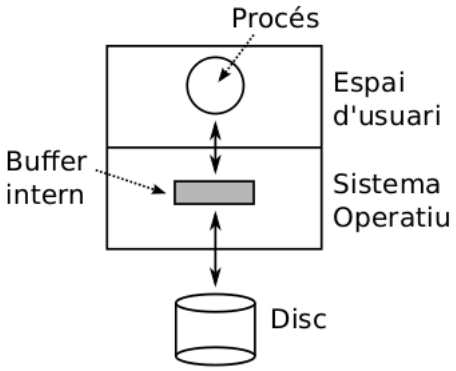
- Manipular arxius és més complicat que manipular directament la memòria.
- Cal assegurar que en tot moment el fitxer conté dades consistents en cas que múltiples processos hi pugin escriure o llegir, si no es produiran errors sensibles no detectables per la màquina.

Amb els arxius disposem de diferents funcions per accedir, llegir i escriure, i disposem de més opcions que les bàsiques que ofereixen les canonades (encara que pràcticament funcionen igual).

Per una banda, disposem, proporcionades pel sistema operatiu, **crides a sistema** per manipular fitxers. Les més importants són les següents:

- open i create Serveixen per obrir i crear un fitxer a **disc**. La funció retorna un sencer: que es correspon amb el descriptor del fitxer obert.
- close Serveix per tancar un fitxer qualsevol. Cal passar com a paràmetre el descriptor de fitxer.
- read i write Serveix per llegir i escriure dades (*i.e. bytes*) de qualsevol fitxer. Cal passar com a paràmetre el descriptor de fitxer corresponent.
- lseek Serveix per establir la posició actual dins del fitxer a disc (per llegir o escriure-hi). Cal passar com a paràmetre el descriptor del fitxer.

El sistema operatiu ens ofereix una sèrie de crides a sistema per manipular fitxers. A més a més, el sistema operatiu utilitza el buffer intern per augmentar l'eficiència d'accés a disc.



Exemple: En aquest exemple veurem dos porcions de codi per manipular arxius amb crides a sistema. Les dades s'emmagatzemen a disc (resp. es llegeixen de disc) tal com estan emmagatzemades a memòria.

write_char_int.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
void main(void)
{
int i, fd;
char *a = "lluis";
fd = open("log_open.data",
O_WRONLY | O_CREAT | O_TRUNC,
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
for(i
for
```

```
= 0; i < 100; i++)
{
write(fd, a, 5);
write(fd, &i, 4);
}
close(fd);
}
```

read_char_int.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
void main(void)
{
int i, k, fd;
char a[6];
fd = open("log_open.data", O_RDONLY);
for(i
for
= 0; i < 100; i++)
{
read(fd, a, 5);
read(fd, &k, 4);
a[5] = 0; // Equivalent a[5] = '\0'
printf("Llegit: %s y %d\n", a, k);
}
close(fd);
}
```

Les funcions open, read, write... són funcions que criden directament al sistema operatiu.

- El sistema operatiu manté un buffer intern (al SO) per a cada fitxer obert, comú a tots els processos.
- A més a més, l'operació write escriu les dades al buffer intern i retorna de seguida. L'operació d'escriptura real a disc es realitzarà més endavant.
- L'operació read comprova primer si les dades estan al buffer intern del sistema operatiu. Si les dades no estan disponibles al buffer, es llegeixen de disc.
- Si un procés escriu a un fitxer i un altre hi llegeix després, es llegirà la dada correcta tot i que les dades no s'hagin escrit a disc.
- Utilitzar aquestes funcions directament pot reduir l'eficiència de l'accés a disc

3.1. Llibreria estàndard

La llibreria estàndard stdio ens ofereix les següents funcions per manipular un fitxer a **disc**. Són les que usualment s'utilitzen per manipular fitxers a l'hora de programar.

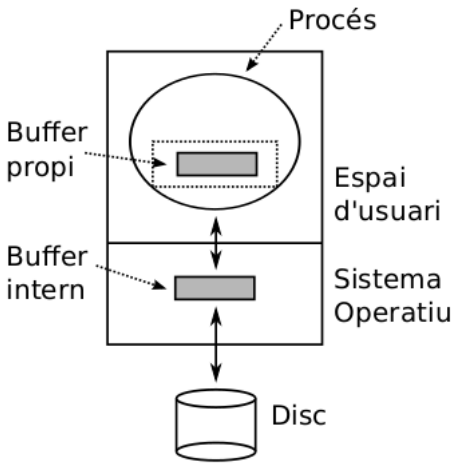
- fopen: Permet obrir un fitxer a disc (per lectura, escriptura,...)

- fclose: Permet tancar un fitxer.
- fread i fwrite: Permeten llegir i escriure dades (*i.e. bytes*). Internament es crida a read i write del sistema operatiu.
- fscanf i fprintf: Permet llegir i escriure dades en format ASCII. Internament es crida a read i write del sistema operatiu.
- fseek: Permet definir la posició actual dins del fitxer.

La llibreria estàndard (stdio) conté funcions per gestionar fitxers i manipular cadenes, per exemple, les funcions de fitxer:

- Utilitzen una **estructura FILE**. Aquesta conté, entre altres,
  1. Un sencer per emmagatzemar el **descriptor de fitxer** associat.
  2. Un **buffer propi** a la memòria d'usuari per emmagatzemar dades.
- Internament fan les crides a sistema que hem vist anteriorment.

Podem veure l'exposat ara amb el següent esquema:



Exemple: Farem exactament el mateix que al exemple anterior però utilitzant les funcions proporcionades per la llibreria estàndard. Les dades s'emmagatzemen a disc (resp. es llegeixen de disc) tal com estan emmagatzemades a memòria.

fwrite_char_int.c

```
#include <stdio.h>
void main(void)
{
FILE *fp;
int i;
char *a = "lluis";
fp = fopen("log_fopen.data", "w");
for(i
for
= 0; i < 100; i++)
```

```
{
fwrite(a, sizeof(char),5, fp);
fwrite(&i, sizeof(int),1, fp);
}
fclose(fp);
}
```

fread_char_int.c

```
#include <stdio.h>
void main(void)
{
FILE *fp;
int i, k;
char a[6];
fp = fopen("log_fopen.data", "r");
for(i
for
= 0; i < 100; i++)
{
fread(a, sizeof(char),5, fp);
fread(&k, sizeof(int), 1, fp);
a[5] = 0; // Equivalent a[5] = '\0'
printf("Llegit: %s y %d\n", a, k);
}
fclose(fp);
}
```

Si en lloc d'utilitzar fwrite i fread utilitzem fprintf i fscanf, les dades s'emmagatzemen a disc (resp. es llegeixen de disc) a una 'representació" humana.

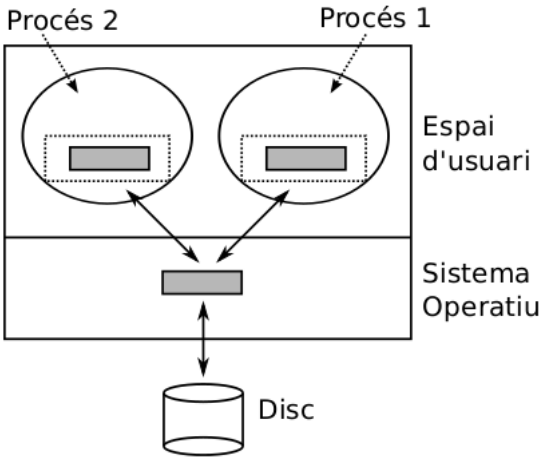
L'estructura FILE i les funcions fopen, fprintf, fscanf, fwrite, fread... són funcions de la llibreria estàndard stdio.

Cal destacar les següents propietats de la llibreria estàndard:

- S'utilitza un buffer propi del procés (no visible als altres processos).
- Les operacions fprintf o fwrite escriuen les dades al buffer de FILE. Quan el buffer es ple, s'escriuen les dades a disc fent servir la funció write (que les escriu al buffer intern).
- Les operacions fscanf o fread llegeixen les dades buffer de FILE. Si no hi estan disponibles, es fa servir la funció read per omplir el buffer de FILE.
- La llibreria estàndard aconsegueix així minimitzar l'ús de les funcions write i read.

Cal destacar que no és adequat fer servir aquestes funcions per comunicació interprocés (*i.e. intercanviar dades entre processos*).

En cas que hi hagi múltiples processos cada procés té el seu propi buffer! Una dada que un procés escrigui al buffer (propi) no es podrà veure de forma immediata a l'altre procés.



Conclusions:

1. Les funcions open, write, etc. són funcions que criden directament al sistema operatiu. Tanmateix utilitzar-les directament pot reduir l'eficiència de l'accés a disc. Són adequades per **comunicació interprocés**.

2. Les funcions fopen, fprintf, fscanf, fwrite, fread, etc són funcions de la **llibreria estàndard** stdio. Aquestes funcions **no** són adequades per **comunicació interprocés** ja que fan servir un buffer d'usuari. Tanmateix estan dissenyades per ser eficients.

*Nota: A Sistemes Operatius 2 s'especificarà com els processos poden comunicar-se i llegir/escriure sense conflictes i com sincronitzar els procesos entre si per assegurar que les dades són sempre consistents.(**semàfors i monitors**)*

3.2. Eficiència: Comparativa crides del sistema / funcions de la llibreria estàndard.

Uns petits apunts relacionats amb l'eficiència d'ambdós mètodes per utilitzar la comunicació interprocés emprant arxius.

Suposem que volem copiar un fitxer d'una gran mida d'un lloc a un altre utilitzant les funcions vistes anteriorment (crides al sistema i funcions de stdio). Si fixem una mida de bloc per anar copiant el arxiu (*Per exemple, una mida de bloc d'1 bit haurà de fer un procés de còpia 3 vegades per copiar un arxiu de 3 bits*) llavors:

- L'eficiència de les crides a sistema depèn molt de la mida de bloc utilitzada. La mida de bloc òptima depèn de la mida de bloc del sistema de fitxers (mida del fitxer).
- Per a crides de sistema, la baixa eficiència per a valors baixos de mida de bloc és perquè passar de mode kernel a usuari és molt costós i es fa reiteradament ja que hem de fer el procés de còpia cridant al sistema reiteradament i moltes vegades quan la mida del bloc és baixa..

- A l'aplicació amb crides a la llibreria estàndard el tems d'usuari és major que amb crides a sistema. Això és pel codi *addicional* que s'executa en mode d'usuari: les crides a funcions estàndard.
- Les crides a funcions de la llibreria estàndard optimitzen *automàticament* l'ús dels recursos del sistema. El temps de sistema és pràcticament independent de la mida de bloc. Això és perquè les llibreries estàndard gestionen les crides a sistema.
- En cas que es vulgui optimitzar al màxim la còpia, s'ha de fer servir crides al sistema amb mides de bloc gran.

4. Xarxa (sockets)

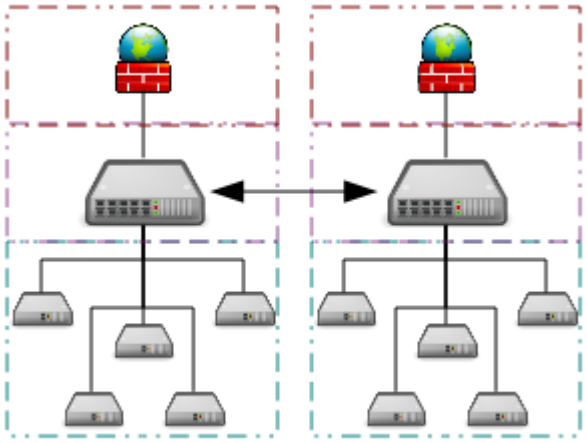
La comunicació interprocés via xarxa permet comunicar múltiples processos entre el mateix o diferents ordinadors.

Tanmateix, la comunicació via sockets és complexa i es donarà amb profunditat a una altra assignatura del grau.

El sistema operatiu ofereix 8 funcions bàsiques per manipular sockets: socket, bind, listen, accept, connect, read, write i close.

Uns petits apunts relacionats amb la comunicació interprocés emprant sockets:

- Un ordinador s'identifica mitjançant l'**adreça IP**, *Per exemple, 161.116.83.153*
- Un procés A pot escoltar les peticions entrants "*engantxant-se*" a un **port**. *Per exemple, 8080*
- Qualsevol altre procés B pot enviar un missatge al procés A obrint un canal de comunicació bidireccional amb l'adreça IP (161.116.83.153) i el port (8080). Si B escriu un missatge, arribarà a A, i viceversa.



5. Senyals

Definició: Senyal: Un senyal és, bàsicament, una interrupció de programari. Els senyals proveeixen d'un mecanisme perquè es puguin gestionar notificacions asíncrones mitjançant les senyals.

El sistema operatiu pot enviar una notificació asíncrona a un procés (és el que al primer tema hem anomenat "*upcall*"). Un procés també pot enviar una notificació asíncrona a un altre procés.

Funcions d'ús:

- En produir-se una excepció (*divisió per zero, accés a memòria invàlida...*) el SO pot cridar a una funció del procés que ha fet l'excepció.
- En polsar Control+C (i.e. fer un kill) al terminal es pot cridar a una funció del procés que volem finalitzar. Abans que el procés mori, podem guardar a disc totes les dades que tenim a memòria.
- En Sistemes d'Alimentació Ininterrompuda (SAI) el sistema pot cridar a una funció de cada procés que s'està executant. D'aquesta forma es poden guardar a disc les dades necessàries.

Es pot trobar una llista completa de senyals al següent link:
`https://en.wikipedia.org/wiki/Unix_signal`. Existeixen un munt de senyals com SIGFPE, SIGSEGV, SIGPWR...

Cada procés pot indicar al sistema operatiu què fer en produir-se un determinat esdeveniment. En concret es pot:

1. **Capturar el senyal**: El procés indica al sistema operatiu quina funció cal cridar en produir-se un determinat esdeveniment.
2. **Acció per defecte**: En cas que el procés no indiqui res de forma explícita, es realitzarà l'acció per defecte (Explicat al link escrit anteriorment).
3. **Ignorar el senyal**: El procés pot indicar al sistema operatiu que ignori el senyal. Això pot ser perillòs (pel procés) en cas que, per exemple, es produeixi una divisió per zero.

   Hi ha dos senyals que no es podran ignorar mai, SIGKILL i SIGSTOP perquè el sistema operatiu tingui un mètode per matar/aturar un procés de manera que no es pugui evitar l'ignorament del senyal. És a dir, el senyal SIGKILL no és capturable i sempre mata el procés, per tenir un mètode "a prova de foc". SIGSTOP per la seva banda permet aturar l'execució d'un procés, i tampoc es pot ignorar.

Existeixen uns altres senyals interessants com SIGCONT que permet continuar l'execució d'un procés prèviament aturat, per exemple, amb el senyal SIGSTOP.

Els senyals SIGUSR1 i SIGUSR2 són per a aplicacions d'usuari, perquè un procés pugui enviar una notificació a un altre procés.

Nota: La funció pause() *bloqueja un procés fins que rep un senyal.*

Nota 2: L'esquema de senyals no està "dissenyat" per saber quin procés ha enviat el senyal.

La funció alarm() *permet especificar un temporitzador. En arribar el temporitzador a zero el sistema operatiu envia un* SIGALARM *al procés que ha fet la crida a* alarm.