Comunicació interprocés

Lluís Garrido

lluis.garrido@ub.edu

Grau d'Enginyeria Informàtica

Comunicació interprocés

Comunicació interprocés

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

En aquestes transparències tractem

1. Com pot un procés passar informació a un altre procés ?
2. Quins problemes poden aparèixer en la comunicació interprocés i com es tracten ?

Comunicació interprocés

Tècniques de comunicació interprocés que existeixen[1]

- Canonades (*pipe*) i canonades amb nom (*named pipe o FIFO*)
- Arxius (*files*)
- Xarxa (*sockets*)
- Senyals

[1] La llista no és exhaustiva. N'hi ha d'altres tècniques.

Conceptes introductoris

POSIX (Portable Operating System Interface) es un estàndard Unix per mantenir compatibilitat entre sistemes operatius.

- 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 del 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 sistema operatiu, un índex a un vector del procés que conté la informació dels fitxers oberts.

Conceptes introductoris

El sistema operatiu ens ofereix una sèrie de crides a sistema per poder proveir de comunicació interprocés

- Obrir una comunicació
  - Funció open per obrir un fitxer de disc.
  - Funció pipe per crear una canonada.
  - Funció mkfifo per crear una canonada amb nom.
  - Funció socket per crear una connexió via xarxa (remota).
  - I moltes més...

  aquestes funcions retornen el descriptor de fitxer.

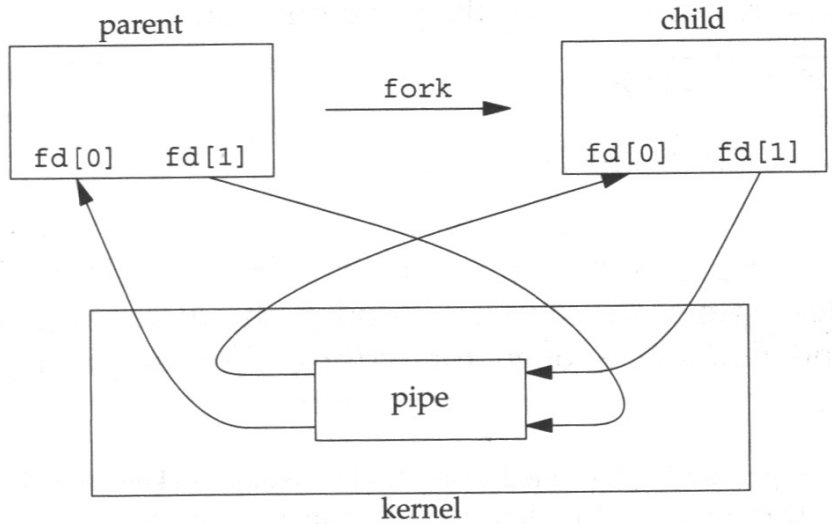
- Per enviar (funció write) i rebre (funció read) informació a través d'un fitxer. Aquestes funcions tenen com a 1r argument el descriptor de fitxer.

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

Canonades (*pipes*)

Una de les primeres formes de comunicació interprocés. També conegudes amb el nom de canonades anònimes. La canonada s'utilitza en processos que tenen una relació pare-fill.

```c
#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;
}
```



Canonades (*pipes*)

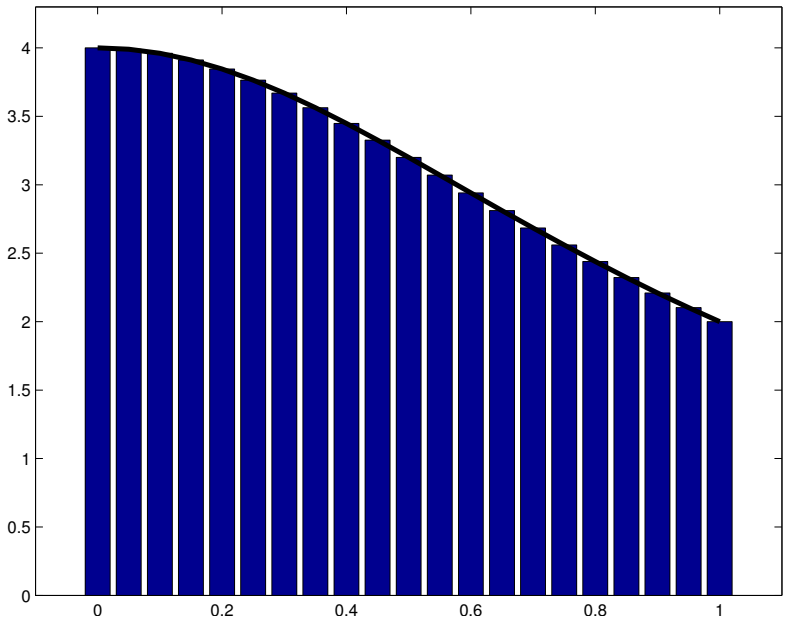
- Hem vist els principis bàsics en un tema anterior, i els utilitzeu a la pràctica 3 des de la línia de comandes.
- En aquest exemple els processos tenen una relació pare-fill (codi `pipe2.c`).

```
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", 5);
    exit(0);
  } else { // parent
    printf("parent reading from file descriptor #%d\n", fd[0]);
    read(fd[0], buf, 5);
    printf("parent read \"%s\"\n", buf);
    wait(NULL);
  }

  return 0;
}
```

Canonades (*pipes*)

Veiem un exemple més "el·laborat": càlcul del valor de π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$




```c
#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 = 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;
}
```

El càlcul de l'àrea de cada rectangle és independent de la resta.

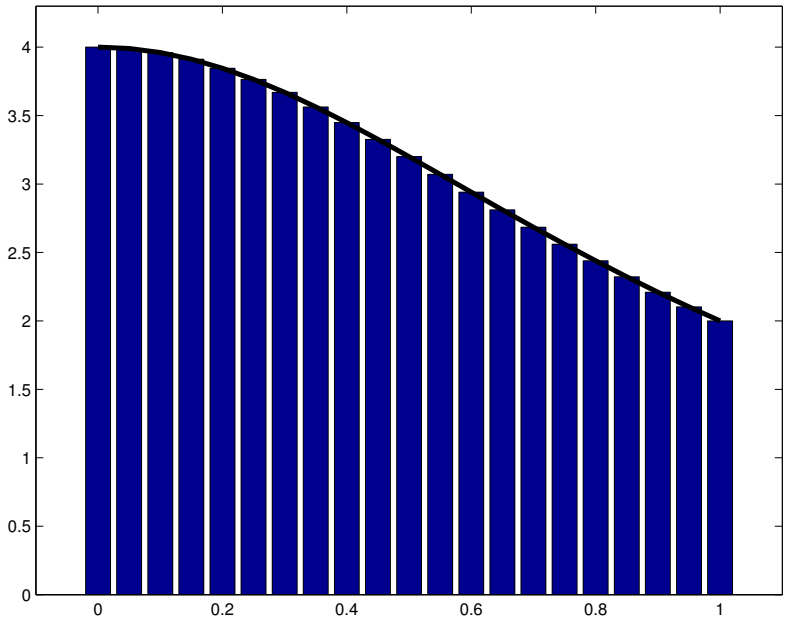
Canonades (*pipes*)

- Implementació fent servir 1 sol procés. Executar amb (codi `calcul_pi_1proces.c`)

  ```
  $ time calcul_pi_1proces
  ```

- Implementació fent servir 2 processos i una canonada per comunicar resultat parcial (codi `calcul_pi_2processos.c`)

  ```
  $ time calcul_pi_2processos
  ```



Canonades (*pipes*)

- A l'exemple anterior el procés pare fa els rectangles senars, el procés fill els rectangles parells. El procés fill comunica al pare el seu resultat.
- El temps d'execució amb 2 processos és menor que amb un procés: el sistema operatiu planifica (i.e. executa) cada procés en CPUs diferents!
- Aquest exemple és una mica "forçat": s'acostumen a fer servir fils per fer aquest tipus de tasques (i.e. computació paral·lela). Ja veurem què són!

Canonades amb nom (FIFO)

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

- Una canonada amb nom (anomenat "FIFO pipe") combina les característiques d'un fitxer de disc i una canonada.
  - En obrir una canonada amb nom, se li associa un nom de fitxer de disc. Aquest fitxer funciona com una canonada. El sistema operatiu 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.
  - Windows implementa les canonades fent servir canonades amb nom (no pot implementar canonades anònimes ja que no fa servir el fork-exec).

Canonades amb nom (FIFO)

Exemple d'ús (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)

Escriptor

```c
#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);
}
```

Lector

```c
#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);
}
```

Comunicació interprocés

Tècniques de comunicació interprocés que veurem[2]

- Canonades (*pipe*) i canonades amb nom (*named pipe o FIFO*)
- Arxius (*files*)
- Xarxa (*sockets*)
- Senyals

[2] La llista no és exhaustiva. N'hi ha d'altres tècniques.

Arxius (*regular files*)

Avantatges

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

Problemes

- 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 puguin escriure o llegir.

Arxius (*regular files*)

Ara els principis bàsics de les funcions amb què podem accedir a fitxers per llegir o escriure.

- Disposem de les crides a sistema: open, read, write, ...
- Disposem de les funcions de la llibreria estàndard: fopen, fread, fwrite, ...

Quines característiques tenen?

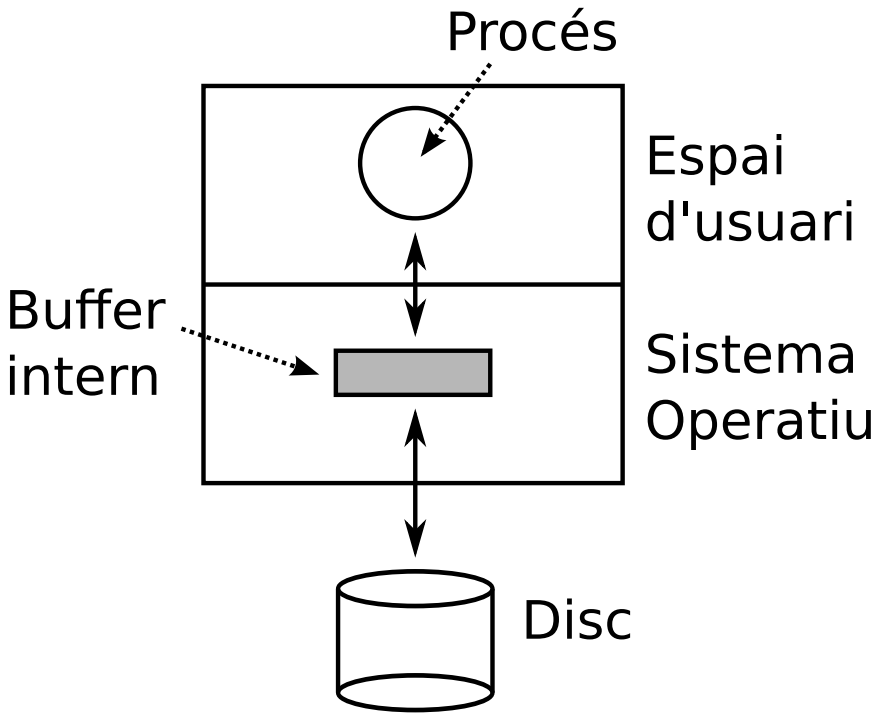
Arxius (*regular files*): crides a sistema

El sistema operatiu ens ofereix crides a sistema per manipular fitxers. Les més importants són:

- open i creat: obrir i crear un fitxer a disc. La funció retorna un sencer: el descriptor del fitxer obert!
- close: tancar un fitxer qualsevol. Cal passar coma paràmetre el descriptor de fitxer.
- read i write: llegir i escriure dades (i.e. bytes) de qualsevol fitxer. Cal passar com a paràmetre el descriptor de fitxer.
- lseek: establir la posició actual dins del fitxer a disc (per llegir o escriure-hi). Cal passar com a paràmetre el descriptor de fitxer.

Arxius (*regular files*): crides a sistema

El sistema operatiu ens ofereix una sèrie de crides a sistema per manipular fitxers.



El sistema operatiu utilitza el buffer intern per augmentar l'eficiència d'accés a disc.

Arxius (*regular files*): crides a sistema

Vegem un exemple per manipular arxius amb crides a sistema. Les dades s'emmagatzemen a disc (resp. es llegeixen de disc) tal com estan emmagatzemades a memòria.

Fitxer `write_char_int.c`

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

  close(fd);
}
```

Fitxer `read_char_int.c`

```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 = 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);
}
```

Arxius (*regular files*): crides a sistema

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

- El sistema operatiu manté un buffer intern (al sistema operatiu), per a cada fitxer obert, comú a tots els processos.
- L'operació write escriu les dades al buffer intern i retorna de seguida. L'operació d'escriptura real a disc es realitza 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 (ara ho veurem!).

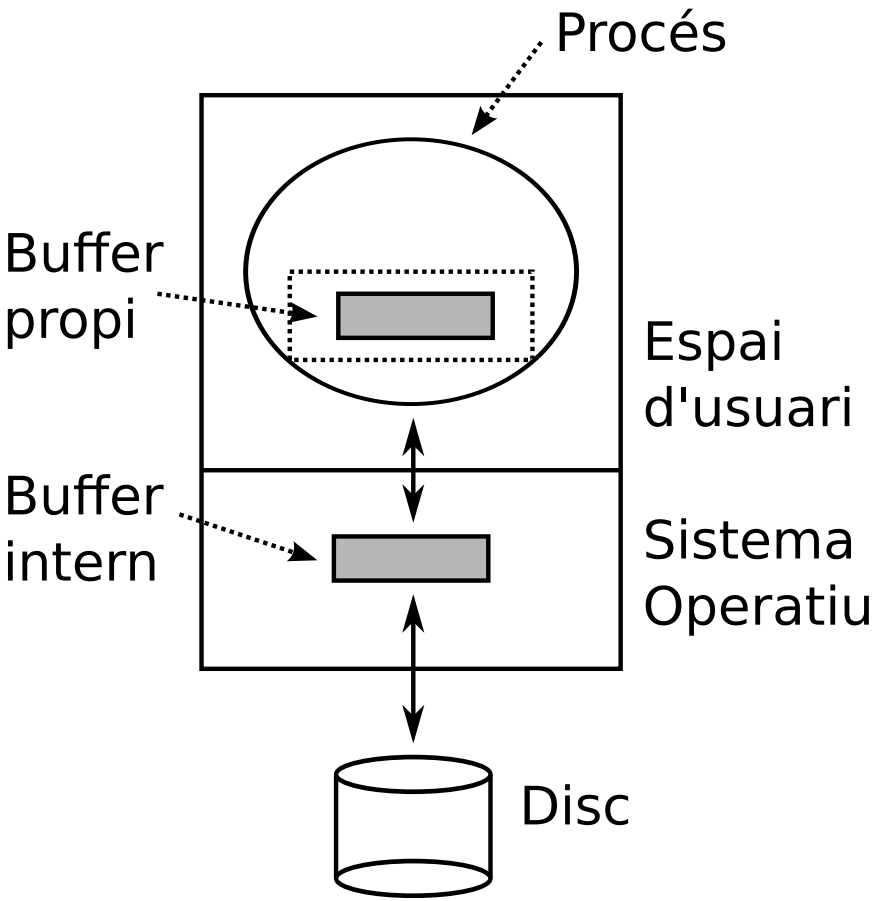
Arxius (*regular files*): llibreria estàndard

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

Arxius (*regular files*): llibreria estàndard

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



- Utilitzen una estructura FILE. Aquesta estructura 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 a crides a sistema que hem vist a les transparències anteriors.

Arxius (*regular files*): llibreria estàndard

Vegem el mateix exemple d'abans fent servir la llibreria estàndard i les funcions fwrite i fread. Les dades s'emmagatzemen a disc (resp. es llegeixen de disc) tal com estan emmagatzemades a memòria.

Fitxer `fwrite_char_int.c`

```c
#include <stdio.h>

void main(void)
{
  FILE *fp;
  int i;

  char *a = "lluis";

  fp = fopen("log_fopen.data", "w");

  for(i = 0; i < 100; i++)
  {
    fwrite(a, sizeof(char), 5, fp);
    fwrite(&i, sizeof(int), 1, fp);
  }

  fclose(fp);
}
```

Fitxer `fread_char_int.c`

```c
#include <stdio.h>

void main(void)
{
  FILE *fp;
  int i, k;
  char a[6];

  fp = fopen("log_fopen.data", "r");

  for(i = 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);
}
```

Arxius (*regular files*): llibreria estàndard

Ara fem servir les funcions fprintf i fscanf. Les dades s'emmagatzemen a disc (resp. es llegeixen de disc) a una "representació" humana.

Fitxer `fprintf_char_int.c`

```c
#include <stdio.h>

void main(void)
{
  FILE *fp;
  int i;

  char *a = "lluis";

  fp = fopen("log_fopen.data", "w");

  for(i = 0; i < 100; i++)
  {
    fprintf(fp, "%s\n", a);
    fprintf(fp, "%d\n", i);
  }

  fclose(fp);
}
```

Fitxer `fscanf_char_int.c`

```c
#include <stdio.h>

void main(void)
{
  FILE *fp;
  int i, k;
  char a[6];

  fp = fopen("log_fopen.data", "r");

  for(i = 0; i < 100; i++)
  {
    fscanf(fp, "%s", a);
    fscanf(fp, "%d", &k);

    printf("Llegit: %s y %d\n", a, k);
  }

  fclose(fp);
}
```

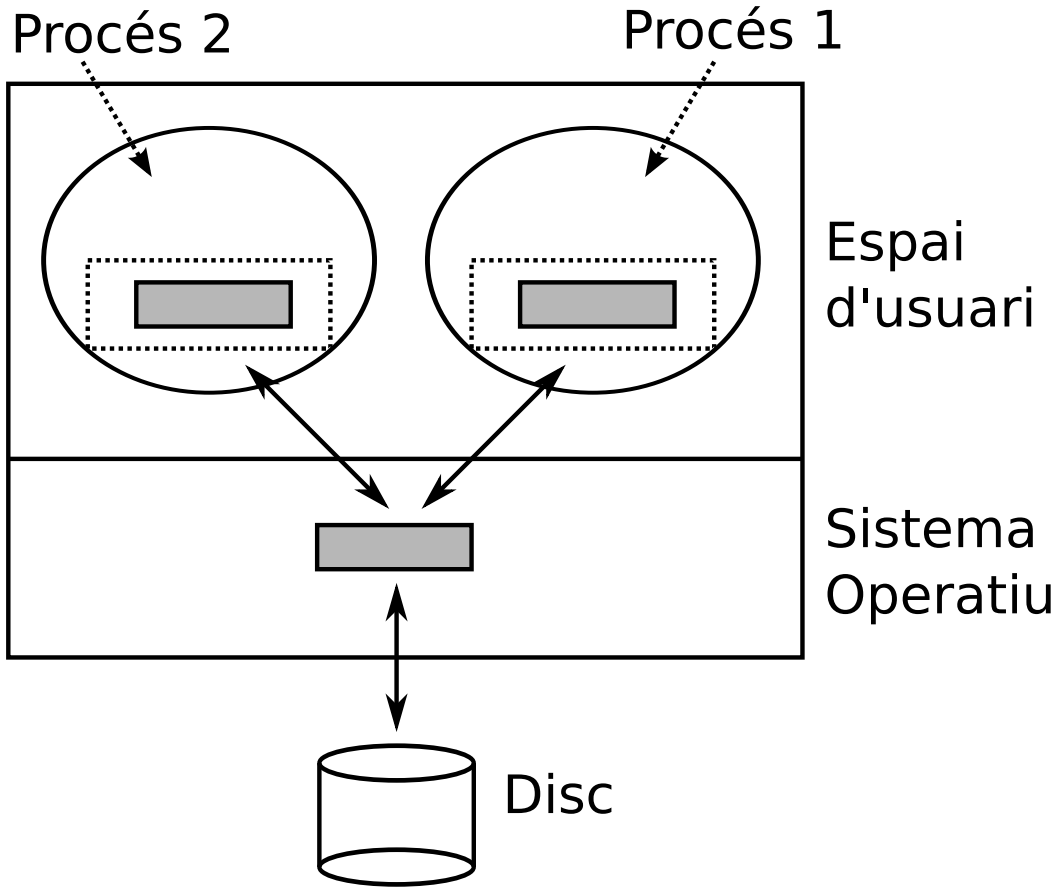
Arxius (*regular files*): llibreria estàndard

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

- 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.
- No és adequat fer servir aquestes funcions per comunicació interprocés (i.e. intercanviar dades entre processos). Per què?

Arxius (*regular files*): llibreria estàndard

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.



Arxius (*regular files*): conclusions

Algunes conclusions...

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

- Utilitzar-les directament pot reduir l'eficiència de l'accés a disc.
- Són adequades per comunicació interprocés.

Les funcions fopen, fprintf, fscanf, fwrite, fread, etc són funcions de la llibreria estàndard "stdio".

- No són adequades comunicació interprocés. Recordar que fan servir un buffer d'usuari.
- Estan dissenyades, però, per ser eficients. Ara ho veurem!

Arxius (*regular files*)

Us podeu preguntar, però,

- Suposem que hi ha un procés A que escriu dades a disc i un altre procés B que les llegeix. Com podem assegurar que el procés B llegeix les dades un cop el procés A les ha escrit?
- Suposem que hi ha múltiples processos escrivint dades al mateix temps a un fitxer, així com d'altres processos que llegeixen dades. Com podem sincronitzar els procesos entre sí per assegurar que les dades al fitxer sempre són consistents?

Totes aquestes respostes a Sistemes Operatius 2, al tema de "semàfors i monitors". La idea: cal sincronitzar els processos entre sí.

Arxius (*regular files*): eficiència de les funcions

Hem vist que disposem de diverses funcions per manipular arxius.
En particular,

- Crides a sistema: funcions read i write
- Funcions de la llibreria estàndard: funcions fread, fwrite, fgetc, fputc, per exemple.

Quina és la eficiència de l'entrada-sortida en fer servir aquestes funcions?
Anem a fer el següent experiment:

- Tenim un fitxer de 4,3GBytes obtingut de `http://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project`.
- Anem a fer una còpia d'aquest fitxer fent servir crides a sistema i funcions de la llibreria estàndard.
- Quant de temps trigarà en fer la còpia?

Arxius (*regular files*): eficiència de les funcions

Aquest és el codi que permet copiar el fitxer fent servir crides a sistema (fitxer `io_system_calls.c`).

```
#include <unistd.h>

#define BUFFSIZE 4096

int main(void)
{
  int n;
  char buf[BUFFSIZE];

  while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
    write(STDOUT_FILENO, buf, n);

  return 0;
}
```

Per fer els experiments utilitzarem diversos valors de BUFFSIZE. Executarem l'aplicació amb

```
./io_system_calls < fitxer.iso > /dev/null
```

on `fitxer.iso` és el fitxer de 4,3GBytes i `/dev/null` és un fitxer especial que descarta tota la informació que s'escriu o redirecciona en ell.

Arxius (*regular files*): eficiència de les funcions

Aquest és el codi que permet copiar el fitxer fent servir funcions de la llibreria estàndard (fitxer `io_stdio_fread.c`).

```
#include <stdio.h>

#define BUFFSIZE 4096

int main(void)
{
  int n;
  char buf[BUFFSIZE];

  while ((n = fread(buf, sizeof(char), BUFFSIZE, stdin)) > 0)
    fwrite(buf, sizeof(char), BUFFSIZE, stdout);

  return 0;
}
```

Es realitzaran els experiments igual que pel cas anterior.

Arxius (*regular files*): eficiència de les funcions

Com podem mesurar el temps d'execució d'una aplicació? Amb la instrucció de terminal `time`

```
time ./io_system_calls < fitxer.iso > /dev/null
```

La sortida és així

```
real 18.944s
user 1.548s
sys 15.528s
```

Els números signifiquen el següent:

- real: temps real (de cronòmetre) entre que comença i finalitza el procés. S'inclou el temps que el procés ha estat bloquejat així com el temps executant altres processos.
- user: temps de CPU que ha estat executant el procés en mode usuari. No s'inclou el temps que el procés ha estat bloquejat.
- sys: temps de CPU que el procés ha estat executant crides a sistema, és a dir, en mode kernel.

Arxius (*regular files*): eficiència de les funcions

Resultats per a les crides a sistema (funcions read i write) amb un disc dur magnètic de l'aula IA

Mida block	Temps d'usuari	Temps de sistema	Temps real
16	8.880s	1m2.356s	1m14.477s
32	4.420s	32.288s	42.748s
64	3.132s	25.124s	43.152s
128	2.460s	17.796s	41.873s
256	1.160s	10.268s	44.389s
512	0.660s	6.356s	41.807s
1024	0.368s	4.080s	41.101s
2048	0.160s	3.068s	40.971s
4096	0.064s	2.604s	41.001s
8192	0.056s	2.364s	41.238s
16384	0.036s	2.444s	41.009s
32768	0.016s	2.380s	40.963s
65536	0.016s	2.416s	41.618s
131072	0.012s	2.384s	41.283s
262144	0.016s	2.368s	41.009s
524288	0.000s	2.376s	41.124s

Arxius (*regular files*): eficiència de les funcions

Resultats per a les funcions de llibreria estàndard (fread i fwrite) amb un disc dur magnètic de l'aula IA

Mida block	Temps d'usuari	Temps de sistema	Temps real
16	16.932s	2.152s	41.155s
32	9.124s	1.912s	41.123s
64	5.244s	2.124s	42.407s
128	3.144s	2.320s	41.248s
256	2.172s	2.280s	41.097s
512	1.564s	2.408s	41.141s
1024	1.440s	2.288s	41.090s
2048	1.128s	2.336s	41.976s
4096	0.512s	2.404s	41.316s
8192	0.400s	2.332s	40.990s
16384	0.248s	2.352s	41.291s
32768	0.112s	2.692s	41.774s
65536	0.096s	2.368s	41.033s
131072	0.084s	2.332s	41.083s
262144	0.036s	2.336s	41.259s
524288	0.016s	2.332s	43.681s

Arxius (*regular files*): eficiència de les funcions

- 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.
- 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!
- A l'aplicació amb crides a la llibreria estàndard el temps d'usuari és major que amb crides a sistema. Això és pel codi "addicional" que s'executa en mode l'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 vulgueu optimitzar al màxim, feu servir crides al sistema amb mides de bloc gran.

Arxius (*regular files*): eficiència de les funcions

Comparem el temps real d'accès a disc dur magnètic (aula IA) amb l'accès a un disc flash (un servidor, disc 250GBytes).

	Crides a sistema		Llibreria estàndard	
Mida block	Disc mag.	Disc flash	Disc mag.	Disc flash
16	1m14.477s	1m42.228s	41.155s	8.660s
32	42.748s	52.265s	41.123s	8.296s
64	43.152s	26.090s	42.407s	8.266s
128	41.873s	13.304s	41.248s	8.282s
256	44.389s	8.197s	41.097s	8.270s
512	41.807s	8.269s	41.141s	8.265s
1024	41.101s	8.273s	41.090s	8.272s
2048	40.971s	8.271s	41.976s	8.314s
4096	41.001s	8.318s	41.316s	8.281s
8192	41.238s	8.269s	40.990s	8.271s
16384	41.009s	8.274s	41.291s	8.286s
32768	40.963s	8.261s	41.774s	8.246s
65536	41.618s	8.286s	41.033s	8.266s
131072	41.283s	8.277s	41.083s	8.272s
262144	41.009s	8.272s	41.259s	8.449s
524288	41.124s	8.285s	43.681s	8.278s

Arxius (*regular files*): eficiència de les funcions

Cal tenir en compte que els proves anterior s'han executat en ordinadors diferents i sistemes opertius diferents (Debian pel disc dur i OpenSuse pel de la flash). Alguns comentaris

- L'eficiència del disc flash és superior a la del disc magnètic. Aquí queda evident tot i que només hi hagi un únic fitxer i estiguem "suposant" que el fitxer està emmagatzemat de forma seqüencial al disc.
- Es conegut que els discos flash són molt més ràpids que els discs durs en accedir aleatòriament a qualsevol part del disc. Això és perquè no tenen parts mecàniques com ho tenen els discs durs.

Comunicació interprocés

Tècniques de comunicació interprocés que veurem[3]

- Canonades (*pipe*) i canonades amb nom (*named pipe o FIFO*)
- Arxius (*files*) i arxius mapats a memòria (*memory mapped files*)
- Xarxa (*sockets*)
- Senyals

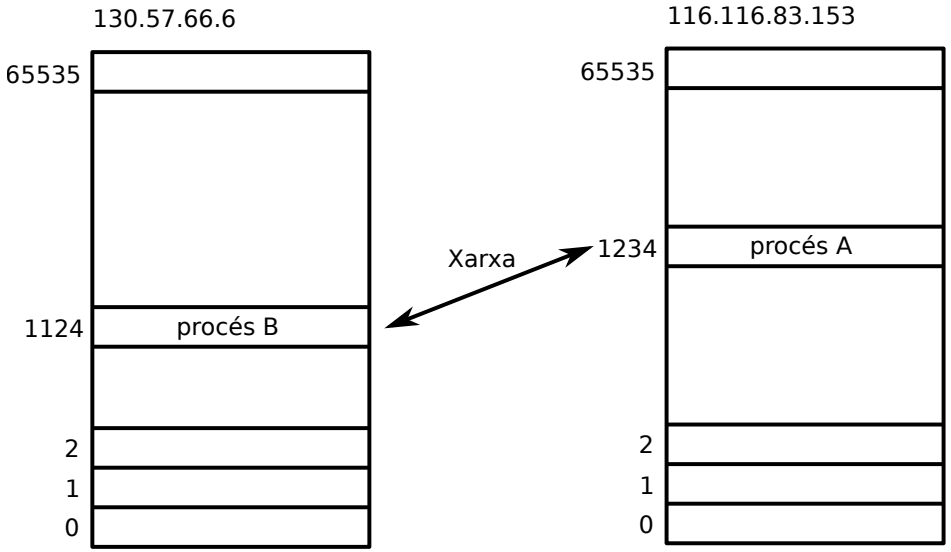
[3] La llista no és exhaustiva. N'hi ha d'altres tècniques.

Xarxa (*sockets*)

- La comunicació interprocés via xarxa permet comunicar múltiples processos entre el mateix o diferents ordinadors.
- La comunicació via *sockets* és complexa, i tindreu una assignatura dedicada a aquest tema.
- El sistema operatiu ofereix 8 funcions bàsiques per manipular sockets: socket, bind, listen, accept, connect, read, write i close.

Xarxa (*sockets*)

- Un ordinador s'identifica mijtançant l'adreça IP, pex. 161.116.83.153)
- Un procés A pot escoltar les peticions entrants "enganxant-se" a un port, pex. 1234.
- 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, port 1234.
  - Si B escriu un missatge, arribarà a A (i a l'inversa).



Xarxa (*sockets*)

Teniu dos exemples a

- `socket_server.c`: servidor (escolta peticions entrants).
- `socket_client.c`: client (fa una petició de connexió amb el servidor).

Tots els detalls – en Java – els tindreu a Software Distribuït.

Comunicació interprocés

Tècniques de comunicació interprocés que veurem[4]

- Canonades (*pipe*) i canonades amb nom (*named pipe o FIFO*)
- Arxius (*files*) i arxius mapats a memòria (*memory mapped files*)
- Xarxa (*sockets*)
- Senyals

[4] La llista no és exhaustiva. N'hi ha d'altres tècniques.

Els senyals

Què són els senyals?

- Són interrupcions de programari. Els senyals proveeixen d'un mecanisme perquè es puguin gestionar notificacions asíncrones.
- 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.

Els senyals

Per a què serveixen?

- En produir-se una excepció (divisió per zero, accés a memòria invàlida, ...) el sistema operatiu pot cridar a una funció del procés que ha fet l'excepció.
- En polsar "Control+C" (o 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.

El llistat de senyals es troba aquí: `https://en.wikipedia.org/wiki/Unix_signal`. Observar que hi ha un munt de senyals: SIGFPE, SIGSEGV, SIGPWR, ...

Els senyals

Quina acció es pren en produir-se un senyal? 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. Veure `https://en.wikipedia.org/wiki/Unix_signal`.
3. Ignorar el senyal: el procés pot indicar al sistema operatiu que ignori el senyal. Això pot ser perillòs (pel procès) en cas que, per exemple, es produeixi una divisió per zero. Hi ha dos senyals que no es podran ignorar mai, `SIGKILL` i `SIGSTOP`[5], perquè el sistema operatiu tingui un mètode per matar/aturar un procés "a prova de foc".

[5] Atenció! No confondre el SIGKILL amb la comanda `kill` de terminal. Hi entrarem amb detall en un moment.

Els senyals

Exemple amb el codi `exemple_sigterm.c`.

```
int done = 0;

void finalitzar(int signo)
{
  printf("He capturat el senyal.\n");
  done = 1;
}

int main(void)
{
  int comptador = 0;

  signal(SIGTERM, finalitzar);

  while (!done)
  {
    sleep(1);
    comptador++;
    printf("He esperat %d segons\n", comptador);
  }

  printf("S'ha acabat el bucle. Finalitzo normalment\n");

  return 0;
}
```

Observar que es crida a la funció `finalitzar` en rebre el senyal `SIGTERM`. Executeu el codi des d'un terminal i fer els següents experiments...

Els senyals

A l'exemple anterior

- Executar des d'un altre terminal la comanda `kill`. Per defecte, la comanda `kill` envia el senyal `SIGTERM` al procés indicat

  ```
  $ kill <pid>
  ```

- Coses a fer
  - En polsar "Control+C" al terminal s'envia el senyal `SIGINT`. Modifiqueu el codi perquè en polsar Control+C es posi `done = 1`.
  - Què passaria si "ignorem" el senyal `SIGTERM`? Ho podem fer indicant SIG_IGN a la funció a cridar en cas que es produeixi un `SIGTERM`. També ho podem fer si evitem posar done = 1.

Els senyals

- Podem enviar qualsevol altre senyal amb la comanda `kill`. Podem veure la llista dels senyals que es poden enviar amb

  ```
  $ kill -l
  ```

- El senyal SIGKILL és similar a SIGTERM però amb una diferència: el SIGKILL no es pot capturar com ho hem fet amb el SIGTERM. És un mètode "a prova de foc" per matar un procés si no ens fa cas...

  ```
  $ kill -SIGKILL <pid>
  ```

Els senyals

Altres senyals "interessants" són el SIGSTOP i SIGCONT

- SIGSTOP: permet aturar l'execució d'un procés (no el mata!). No es pot ignorar.
- SIGCONT: permet continuar l'execució d'un procés prèviament aturat.

Proveu amb

- Enviar un SIGTOP

```
$ kill -SIGSTOP <pid>
```

- Enviar un SIGCONT

```
$ kill -SIGCONT <pid>
```

Altres senyals

- Els senyals `SIGUSR1` i `SIGUSR2` són per a aplicacions d'usuari, perquè un procés pugui enviar una notificació a un altre procés. Veure codi `sigusr.c`.

```
void sigusr(int signo)
{
  if (signo == SIGUSR1)
    printf("He rebut SIGURS1.\n");
  if (signo == SIGUSR2)
    printf("He rebut SIGURS2.\n");
}

int main(void)
{
  signal(SIGUSR1, sigusr);
  signal(SIGUSR2, sigusr);

  while (1)
  {
    printf("Espero senyal!\n");
    pause();
  }

  return 0;
}
```

La funció `pause` bloqueja el procés fins que rep un senyal.

Els senyals

Experiment

- Executar el codi `sigusr.c`. El codi es quedarà a l'espera a rebre senyals d'altres processos.
- Executar el codi `enviar_sigusr.c`. Observar que la funció C que permet enviar un senyal es diu `kill`.

Alguns detalls

- L'esquema de senyals no està "dissenyat" per saber quin procès 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`.