Comunicació interprocés

Sistemes Operatius I

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. Per fer-ho fan falta crides a sistema atès que els processos són independents entre sí.
- El concepte de "comunicació interprocés" (interprocés communication o IPC) s'aplica exclusivament a la comunicació entre processos diferents.

En aquestes transparències tractem (algunes) de les eines que proveeix el sistema operatiu perquè diferents processos es puguin comunicar entre sí.

Comunicació interprocés

Tècniques de comunicació interprocés que existeixen[1]

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

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

Conceptes introductoris

POSIX (Portable Operating System Interface) és un estàndard per mantenir compatibilitat entre diferents implementacions del sistema operatiu Unix.

- 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 del sistema operatiu Unix, 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 permetre la comunicació.

- 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.

Els senyals

Què són els senyals?

- És una forma "limitada" de comunicació interprocès. Els senyals proveeixen d'un mecanisme perquè es puguin enviar notificacions asíncrones a processos. Es poden interpretar, de fet, com interrupcions de programari (és el que al primer tema hem anomenat "upcall").
- El sistema operatiu pot enviar una notificació asíncrona a un procés. En aquell moment l'execució del procés s'interromp i es crida a una determinada funció del procés. El procés ha de d'indicar, prèviament al sistema operatiu, quina funció vol que es cridi en rebre aquell senyal.
- Un procés també pot enviar una notificació asíncrona a un altre procés. Ho fa mitjançant una crida a sistema en que cal indicar el PID del procés així com el senyal a enviar. El sistema operatiu envia aleshores la notificació asíncrona al procés.

Els senyals

Exemples d'ús

- En produir-se una excepció (divisió per zero, accés a memòria invàlida, ...) el sistema operatiu fa la notificació al procés que ha fet l'excepció. El procés pot aleshores enviar informació al seu web amb un "log" del problema abans de morir...
- En polsar "Control+C" (o fer un kill) al terminal el sistema operatiu fa la notificació al procés que volem aturar. En aquell moment el procés pot guardar a disc totes les dades que té a memòria abans de morir...
- En Sistemes d'Alimentació Ininterrompuda (SAI) el sistema pot fer una notificació asíncrona als processos si li queda poca bateria. D'aquesta forma els processos es poden guardar a disc les dades necessàries abans que el sistema s'apagui.

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. 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`.
2. Capturar el senyal: el procés indica al sistema operatiu quina funció cal cridar en produir-se un determinat esdeveniment.
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 i es continuï l'execució com si res hagués passat.

Els senyals

Exemple amb el codi `exemple_sigterm.c`.

```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 cridi a la funció `finalitzar`.
  - 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[2]. És un mètode "a prova de foc" per matar un procés si no ens fa cas...

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

[2] Hi ha dos senyals que un procés no podrà ignorar mai, `SIGKILL` i `SIGSTOP`, perquè el sistema operatiu tingui un mètode segur per matar/aturar un procès.

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.

Agafeu un procès qualsevol (pex el navegador) i proveu

- Enviar un SIGTOP

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

- Enviar un SIGCONT

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

Els senyals

Altres senyals

- Els senyals `SIGUSR1` i `SIGUSR2` estan "reservats" 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`.

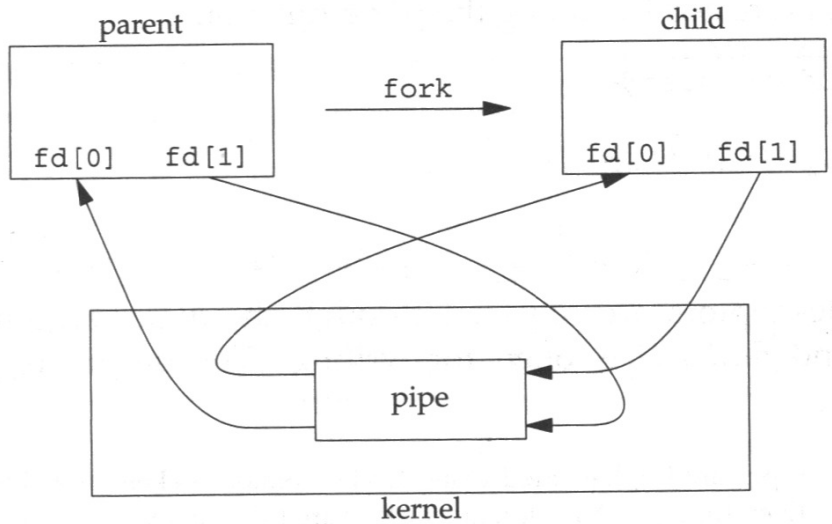
Alguns detalls

- 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`.

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*)

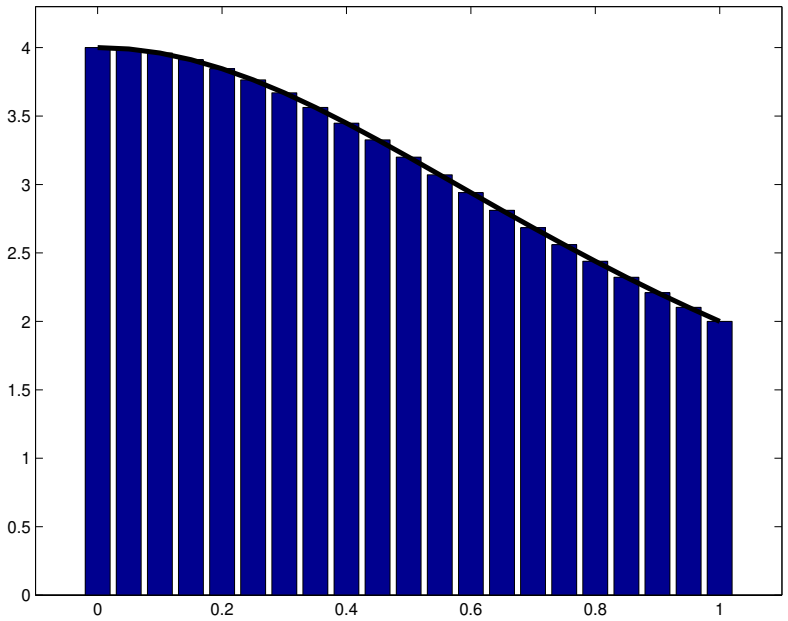
- Hem vist els principis bàsics en un tema anterior, i els heu utilitzat a les pràctiques des de la línia de comandes.
- En aquest exemple els processos tenen una relació pare-fill (codi `pipe2.c`).

```
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);
  }

  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 1E09
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.

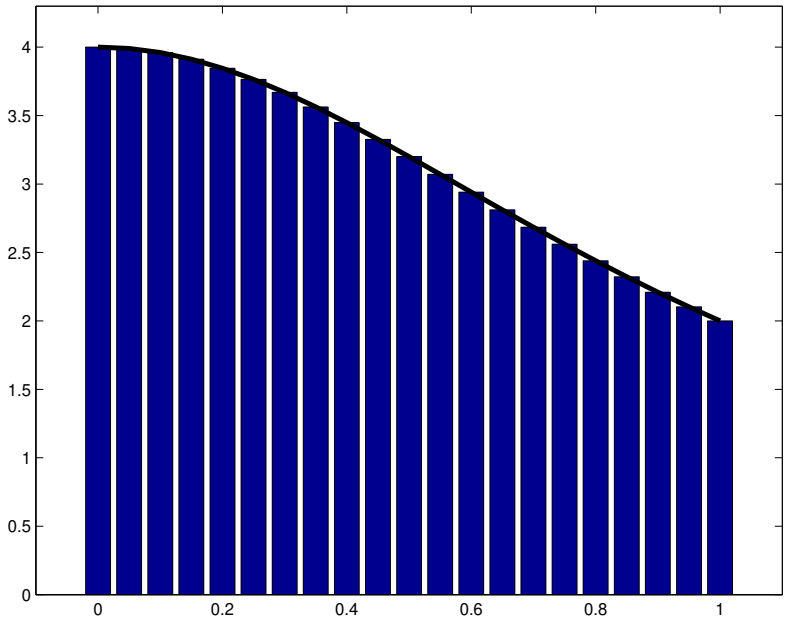
Canonades (*pipes*)

- 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 FIFO. 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 anònimes 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[3]

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

[3] 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.
- Múltiples processos poden accedir al mateix temps a un fitxer, sigui per escriptura o lectura.

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*)

Les funcions amb què podem accedir a fitxers per llegir o escriure (en C) són, bàsicament

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

Quines característiques tenen?

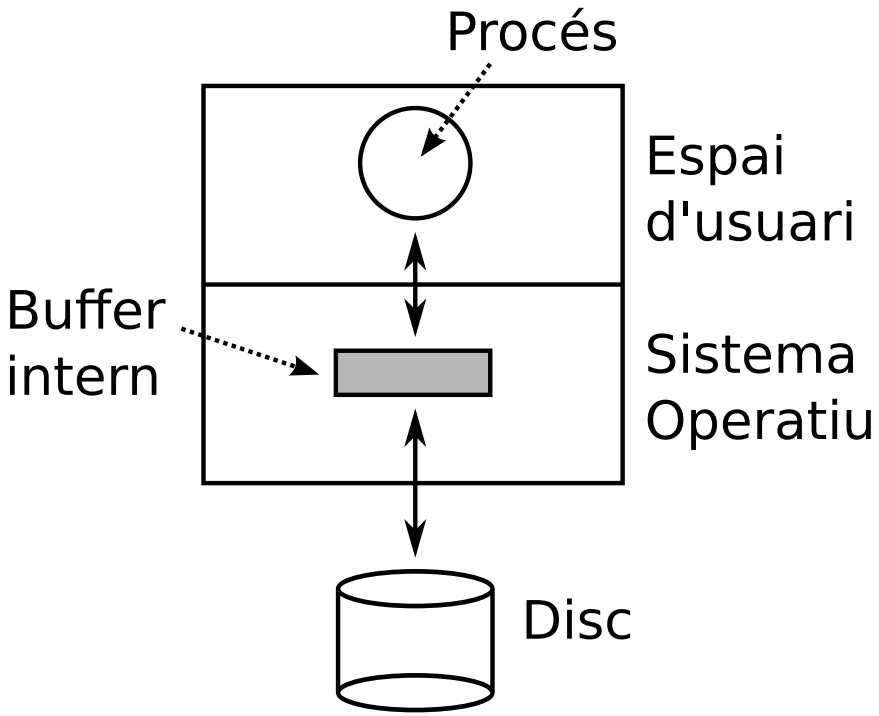
Arxius (*regular files*): crides a sistema

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

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("file.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, sizeof(int));
  }

  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("file.data", O_RDONLY);

  for(i = 0; i < 100; i++)
  {
    read(fd, a, 5);
    read(fd, &k, sizeof(int));

    a[5] = 0; // Equivalent a[5] = '\0'
    printf("Llegit: %s y %d\n", a, k);
  }

  close(fd);
}
```

Arxius (*regular files*): crides a sistema

Podem escriure i llegir vectors de dades amb una sola instrucció (i així evitem múltiples crides a sistema, que és més ineficient).

Fitxer `write_vector_int.c`

```c
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

void main(void)
{
  int i, fd, vector[1000];

  fd = open("file.data",
        O_WRONLY | O_CREAT | O_TRUNC,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

  for(i = 0; i < 1000; i++)
     vector[i] = 2 * i + 1;

  write(fd, &vector, sizeof(int) * 1000);

  close(fd);
}
```

Fitxer `read_vector_int.c`

```c
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>

void main(void)
{
  int i, fd, vector[1000];

  fd = open("file.data", O_RDONLY);

  read(fd, vector, sizeof(int) * 1000);

  close(fd);

  for(i = 0; i < 1000; i++)
     printf("%d\n", vector[i]);
}
```

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. Mestrestant, el procés que ha fet la crida a read queda "bloquejat".
- 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.

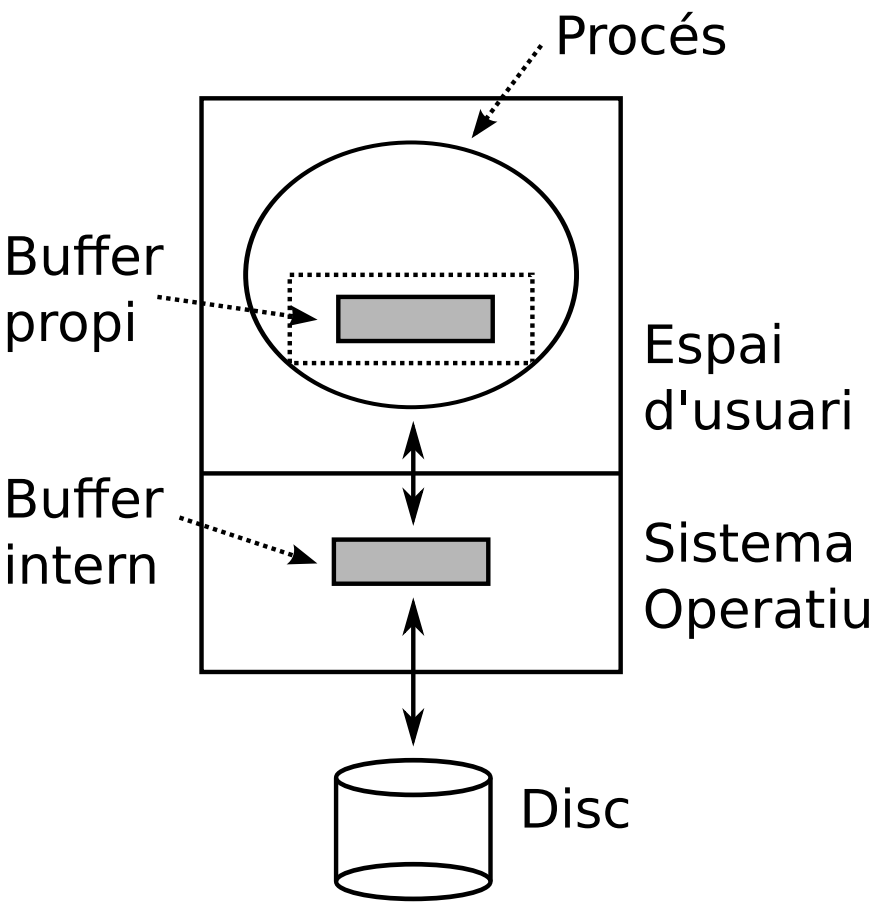
Arxius (*regular files*): llibreria estàndard

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

Funcionament de les les funcions de fitxer de la llibreria estàndard:



- 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.
- Les funcions de la llibreria estàndard fan, internament, les a crides a sistema quan és necessari.

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);
}
```

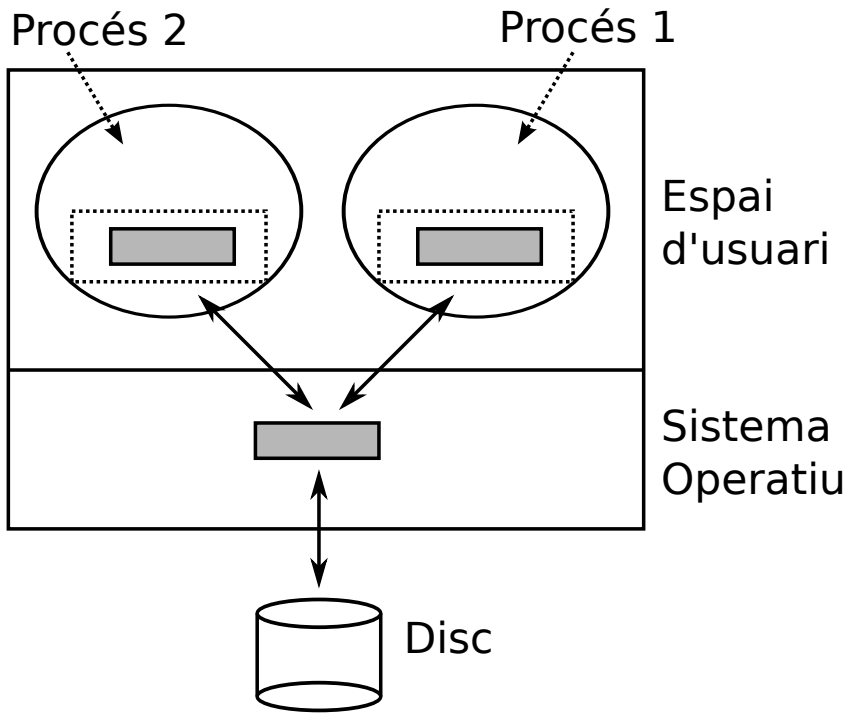
Arxius (*regular files*): llibreria estàndard

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 crida a sistema 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 crida a sistema read per omplir el buffer de FILE. Mestrestant, el procés que ha fet la crida queda "bloquejat".
- La llibreria estàndard aconsegueix així "minimitzar" l'ús de les funcions write i read.

Arxius (*regular files*): llibreria estàndard

En alguns casos pot no ser adequat fer servir aquestes funcions per comunicació interprocés. 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

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

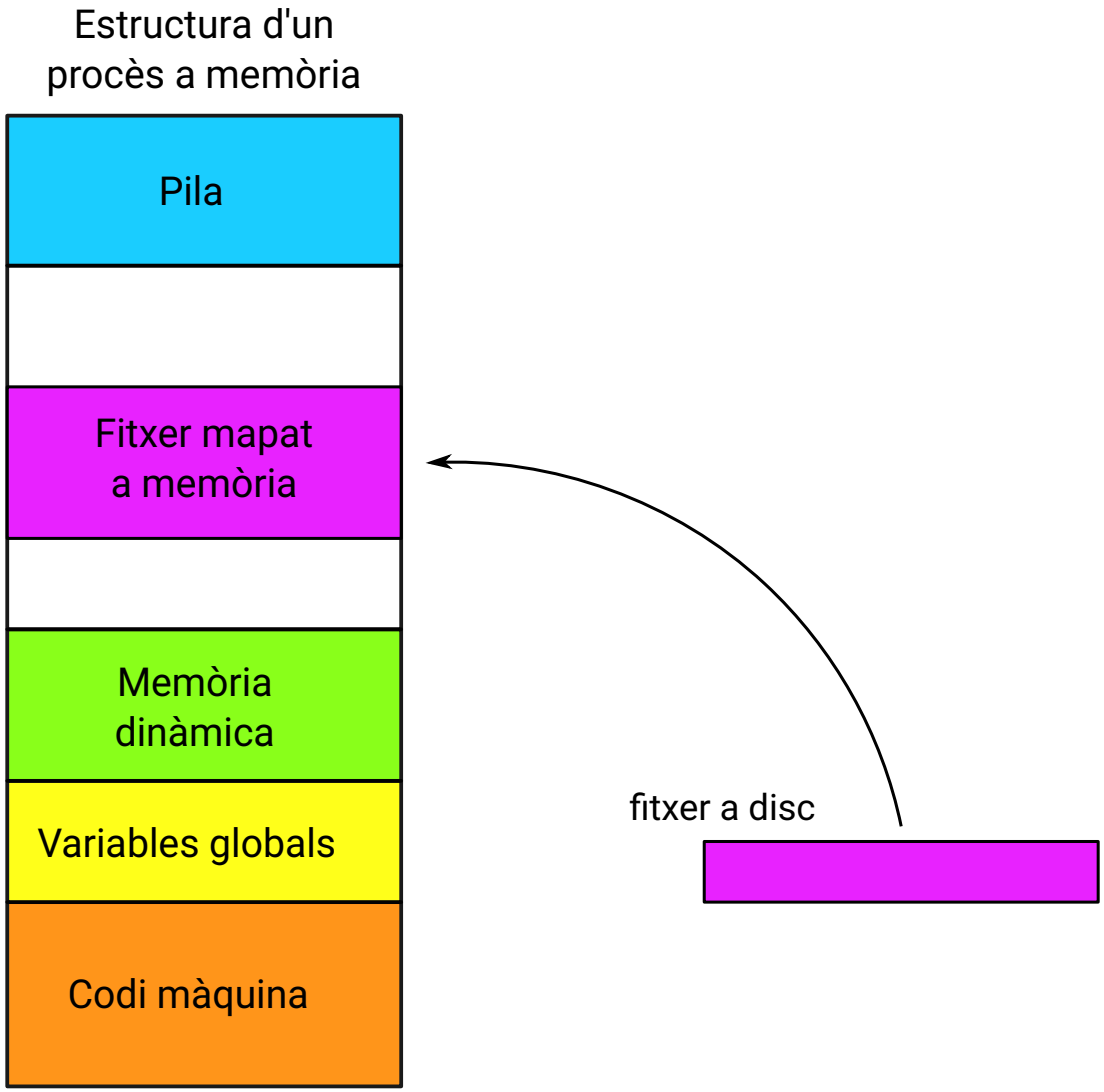
- Són adequades per comunicació interprocés.
- Utilitzar-les directament pot reduir l'eficiència ja que una crida a sistema es costosa.

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

- Internament fan les crides a sistema esmentades a sobre (open, read, write, ...)
- Estan dissenyades per ser eficients a l'accès a disc ja que "minimitzen" les crides a sistema.
- Poden no ser adequades comunicació interprocés. Recordar que fan servir un buffer d'usuari.

Arxius mapats a memòria

Es poden manipular arxius mapant un arxiu de disc a memòria.
Quines avantatges té?



Arxius mapats a memòria

Alguns beneficis dels arxius mapats a memòria

- Podem accedir al contingut d'un fitxer com si fos un vector de dades sense necessitat de fer servir explícitament les funcions read i write. Veure exemple `mmap.c`.
- El sistema operatiu s'encarrega, automàticament, de "carregar" a memòria la part necessària del fitxer en llegir-lo. De forma similar s'encarrega, automàticament, de desar la part necessària al fitxer si canviem el vector.
- Aquesta tècnica és la que fa servir el sistema operatiu per implementar la memoria virtual dels processos!!!! Ja ho veurem...

Arxius mapats a memòria

És una forma molt eficient i fàcil de comunicació interprocés entre processos que tenen una relació pare-fill (o qualssevol processos, encara que no tinguin una relació pare-fill).

- Exemple del fork amb un malloc en què no es comparteix memòria entre processos pare-fill. Codi `fork-variables-malloc.c`.
- Exemple equivalent en què fent servir la crida a sistema mmap es pot compartir memòria pare-fill. Codi `fork-variables-mmap.c`.

Comunicació interprocés

Tècniques de comunicació interprocés que veurem[4]

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

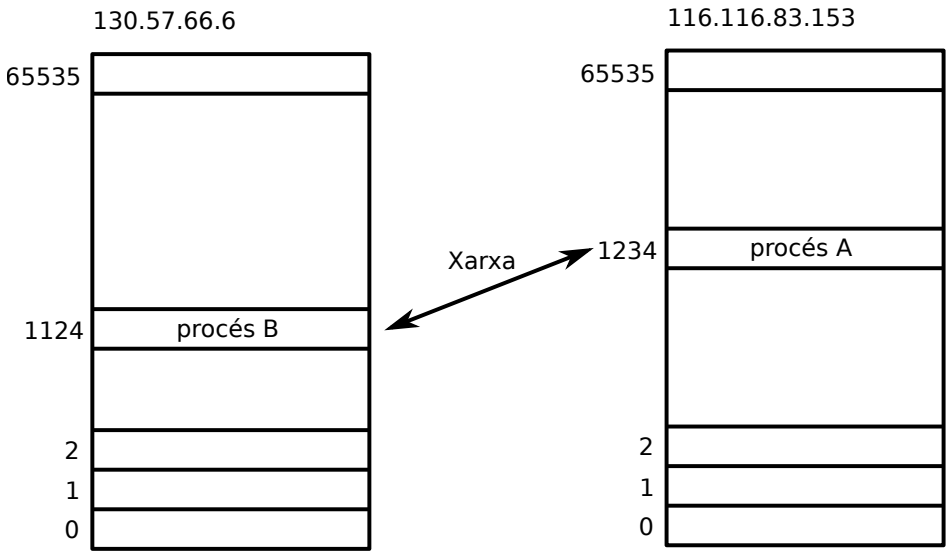
[4] La llista no és exhaustiva. N'hi ha d'altres tècniques.

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.