Apunts de pipes

Jordi Fornés

13/12/2021

Dedicat al sistema operatiu Irix i a l'O_2 de SGI.

Índex

1 Una mica d'història

Brian Kernighan escriu a les seves memòries [Kernighan (2020)] que la idea de les pipes de UNIX cal apuntar-se-la a Doug McIlroy que ja des de 1964 emprenyava a Ken Thompson demanant-li que els programes en UNIX s'haurien de poder acoblar "com manegues de jardí". Ken, que al principi no li veia cap sortida a això de connectar programes, va continuar pensant en una solució que fes callar a en Doug. A la fi, ja ben entrat l'any 1972, li va arribar la idea feliç. Ken va poder afegir la crida al sistema en una hora, car els mecanismes de redireccionament de l'entrada/sortida ja hi eren. Amb la *syscall* implementada, en Thompson afegí el mecanisme al shell i va quedar meravellat del resultat.

La notació era simple i elegant. Ken Thompson i Denis Ritchie hi hagueren de reescriure el codi de totes les comandes del UNIX de 1972 (cosa que van fer en una nit) i en el camí inventaren la sortida estàndard d'error.

Tot plegat no va ser una tasca gaire feixuga, però en canvi el resultat va ser una de les aportacions més sorprenents de UNIX. Ara no calia crear un nou programa per una nova tasca, sinó combinar-ne programes ja existents. Per exemple, podeu respondre quantes sessions té obertes en `jfornes` amb una sola línia de comandes? Res més senzill:

```
$ who | grep jfornes | wc
```

Les pipes han estat a UNIX des de llavors, per bé d'alguns intents de subsumir-les. Uns anys després, amb la introducció de les xarxes (i dels multiprocessadors), també aparegueren d'altres mecanismes d'IPC (*Interprocess Communications)*. El UNIX de 4.2BSD (Berkeley Software Distributions, agost de 1983) afegí els `sockets`. També, com les pipes, feia servir el mecanisme d'entrada/sortida de fitxers per comunicar processos, mitjantçant els descriptors de fitxers. La gran diferència amb les pipes era (i és encara) que els sockets amplia-ven l'IPC a la comunicació de processos en màquines diferents [Sechrest (1986)]. Però això és una altra història i haurà de ser explicada en un altre moment[1].

2 Des de la línia de comandes

Tornant al nostre exemple (`who | grep jfornes | wc`). Per entendre l'objectiu d'aquesta *pipeline*, us heu de retrotreure a quan les màquines tenien molts usuaris, connectats des de terminals (físics: pantalla de fòsfor verd, teclat i cable sèrie). El primer programa, `who`, mostra per pantalla els usuaris connectats a la màquina. Per defecte, apareixen en columna i si, com acostuma a passar en grans màquines, hi ha moltes sessions obertes, és difícil trobar el que vols:

```
jfornes@sert-entry-2:~$ who
xavim    pts/0        2020-10-30 15:35 (gw-4.ac.upc.es)
marc     pts/1        2020-11-02 16:57 (pcbona-i-amargos.ac.upc.es)
alopez   pts/4        2020-10-09 10:48 (tmux(25651).%4)
xavim    pts/5        2020-11-02 17:38 (gw-4.ac.upc.es)
gvavouli pts/7        2020-10-23 09:26 (gw-5.ac.upc.es)
victor   pts/12       2020-11-03 06:47 (gw-5-vpn-i.ac.upc.edu)
alopez   pts/13       2020-10-09 10:19 (tmux(25651).%0)
alopez   pts/14       2020-10-09 10:21 (tmux(25651).%1)
alopez   pts/15       2020-10-09 10:22 (tmux(25651).%2)
antoniof pts/20       2020-11-02 10:54 (tmux(27966).%5)
victor   pts/22       2020-11-03 07:08 (gw-5-vpn-i.ac.upc.es)
isaac    pts/23       2020-11-03 08:32 (gw-4.ac.upc.es)
jfornes  pts/25       2020-11-03 09:17 (gw-5.ac.upc.edu)
antoniof pts/26       2020-10-29 12:03 (tmux(27966).%1)
antoniof pts/27       2020-10-29 12:03 (tmux(27966).%2)
jfornes  pts/28       2020-11-03 09:18 (sert-entry-1.ac.upc.es)
jfornes  pts/29       2020-11-03 09:19 (sert-entry-1.ac.upc.es)
antoniof pts/30       2020-10-29 15:06 (tmux(27966).%4)
```

El següent programa de la *pipeline*, `grep`, serveix per buscar expressions regulars. Les sigles de `grep` volen dir ***g****eneral sea****r****ch re****g****ular* ***e****xpression and* ***p****rint*. Abans de les pipes, podríem haver guardat la sortida del `who` a un fitxer temporal i després buscar al fitxer les aparicions de l'expressió `jfornes`:

```
$ who > fitxer_who.tmp
$ grep jfornes fitxer.tmp
jfornes  pts/25       2020-11-03 09:17 (gw-5.ac.upc.edu)
```

[1] De fet, la 4.2BSD reimplementà les pipes. Fins llavors, les pipes feien servir el sistema de fitxers, des d'aquesta versió, les pipes s'implementaren amb sockets. Les versions actuals de FreeBSD, no fan servir ni una cosa ni l'altra, sinó una implementació separada i optimitzada per comunicacions locals [McKusick (2005)].

```
jfornes  pts/28       2020-11-03 09:18 (sert-entry-1.ac.upc.es)
jfornes  pts/29       2020-11-03 09:19 (sert-entry-1.ac.upc.es)
```

Això no solucionaria del tot el problema. Com es pot apreciar, els usuaris acostumen a obrir més d'un terminal (ara ja no són físics i cada `pts` és una finestra amb un shell independent). Per saber quantes sessions té obertes un usuari podem fer servir la comanda `wc` (***w**ord **c**ount*) que mostra quantes línies, paraules i lletres té l'entrada que li passem. Caldria, però, tenir guardada la sortida del `grep` anterior:

```
$ who > fitxer_who.tmp
$ grep jfornes fitxer_who.tmp > fitxer_grep.tmp
$ wc fitxer_grep.tmp
 3      15     185
$ rm  fitxer_who.tmp fitxer_grep.tmp
```

Però amb l'ús de les pipes, podem fer totes tres línies de comandes en una de sola i estalviar-nos els fitxers temporals.

```
$ who | grep jfornes | wc
 3      15     185
```

La pipe és un canal de comunicació unidireccional entre processos. És una estructura FIFO, tot el que entra surt en l'ordre d'entrada o es queda dins la pipe fins que algun procés drena el contingut (i llavors desapareix de la pipe).

No ens hem d'amoïnar (per ara) per la capacitat d'emmagatzemament de la pipe. Tots el caràcters que `who` escriu a la pipe es queden allà fins que `grep` els llegeix. En llegir-los desapareixen.

Cal notar que hi ha quatre processos involucrats en aquesta línia de comandes; `who`, `grep` i `wc` són tres processos germans. El seu pare és el shell, `$`, que és qui ha creat tant els processos com les pipes.

Un apunt més. El resultat de la línia de comandes no és del tot adient. En surten tres nombres, quan el que voliem saber era un de sol, el primer, que respón a la pregunta inicial: quantes sessions té obertes en jfornes? Sabries com fer perquè la resposta sigui només el primer nombre?

Esclar! Afegint una altra pipe i una comanda. La comanda `awk` és molt potent, conté tot un llenguatge. Posarem ací només el mínim imprescindible per resoldre l'exercici:

```
$ who | grep jfornes | wc |  awk '{ print $1 }'
3
```

Com ja haureu endevinat, `awk` escriu la primera columna que li arriba per la pipe. Per defecte, el separador de columnes és l'espai en blanc. Si el voleu canviar feu servir l'opció `-F`.

Acabem aquest capítol introductori amb un exemple real de la potència de les pipes a la línia de comandes. Aquesta seqüència de comandes UNIX imprimeix les 10 paraules més frequents de l'entrada[2].

```
$ cat $* | tr -sc A-Za-z '\012' | sort | uniq -c | sort -n | tail  | pr -t -5
```

[2]He actualitzat un exemple del Kernighan que ja apareixia al seu llibre [Kernighan (1984)].

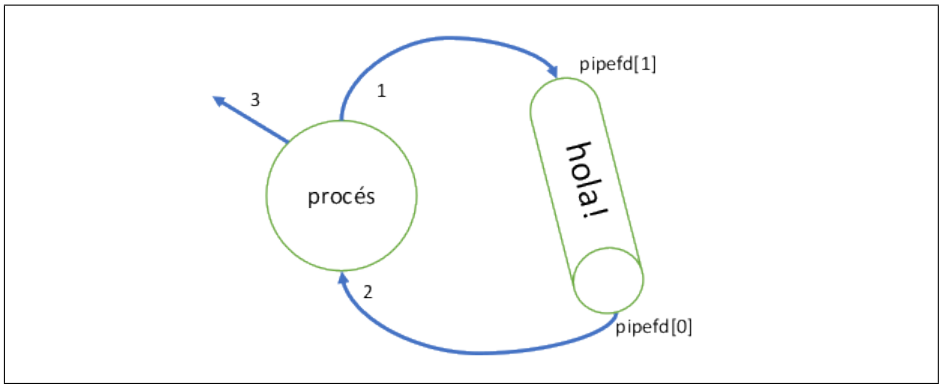

Figura 1: Un procés i una pipe.

Busqueu, amb l'ajuda del `man`, què fa cada comanda, però la part que ens interessa ara mateix és entendre com les pipes connecten la tasca feta pel `cat` amb l'entrada del `tr` i així successivament fins que `pr` escriu per sortida estàndard el que ha llegit de la pipe, formatat a 5 columnes.

3 Crides al sistema: amb nom i sense

Hi ha diferents crides al sistema relacionades amb les pipes. Per veure-les totes piqueu `apropos pipe` a la línia de comandes. Com sempre, la millor font de documentació és el `man`. El capítol 2 ens explica com crear una pipe; mentre que el capítol 7 ens apropa una visió general d'aquest mecanisme IPC. Allà s'inclouen detalls d'implementació específics del vostre sistema.

Us proposo un exercici, feu `man 7 pipe` i esbrineu quina és la mida del buffer circular de la pipe al vostre sistema.

La *syscall* per crear una pipe `|` és

```
int pipe(int pipefd[2]);
```

El paràmetre `pipefd` és un vector d'enters de dues posicions que la crida retorna ple amb dues entrades a la taula de canals (*file descriptors*), `pipefd[0]` és el canal de lectura i `pipefd[1]` és el canal d'escriptura de la pipe.

Mireu l'exemple del llistat 1. El programa només escriu per pantalla "`hola!`", d'una manera una mica llarga. A la figura 1 tenim escenificat el que està passant: abans d'escriure per sortida estàndard, el procés escriu per la pipe "`hola!`" (1), llegeix de la pipe tot el que ha escrit (2) i llavors, allò que ha llegit ho escriu per pantalla (3). Anotem una mica el codi del llistat 1. La pipe es crea a la línia 4. Com a qualsevol altra *syscall*, `pipe` retorna -1 si hi ha hagut algun error (i actualitza la variable `errno`). En cas contrari, a la línia 8, escrivim a l'entrada de la taula de canals (*file descriptor*) apuntada per `pipefd[1]`.

La *syscall* `write` retorna el nombre de bytes que ha escrit. El tercer paràmetre de la funció indica quants bytes volem escriure del buffer que passem al segon paràmetre. No sempre és possible escriure tot el que volem, en aquest cas 6. És per això que guardem el valor retornat a la variable `r`. Ara la pipe té dades. Aquestes dades les llegim (i per tant desapareixen de la pipe) a la línia 12. Noteu que ara llegim del canal de lectura de la pipe (`pipefd[0]`). Finalment, a la

línia 16, escrivim tot el que hem llegit de la pipe (i només això) per la sortida estàndard (canal/*file descriptor* 1).

Tal com hem dit, les pipes són per comunicar processos, però en aquest cas només hi ha un procés que es comunica amb si mateix. Aquí la pipe no serveix per a res (més que com a exemple senzill). Per ara.

Llistat 1: Un procés i una pipe

```
1  int main() {
2  int pipefd[2],r;
3  char c, buff[8];
4  if (pipe(pipefd)<0) {
5          perror("Error en crear la pipe.");
6          exit(1);
7  }
8  if ((r=write(pipefd[1],"hola!\n",6))<0) {
9                  perror("Error en escriure a la pipe");
10                 exit(2);
11 }
12  if ((r=read(pipefd[0],buff,r))<0) {
13         perror("Error en llegir de la pipe");
14         exit(3);
15 }
16 write(1,buff,r);
17 exit(0);
18 }
```

Mirem ara un exemple més sucòs al llistat 2. Diuen que l'emperador romà Juli Cèsar feia servir, per comunicar-se amb els seus generals, una codificació ben senzilla. Cada lletra del missatge pla era substituida per la lletra corresponent a un desplaçament de n posicions a l'alfabet. En aquest exemple, n val 3 i el missatge a codificar és "hola!".

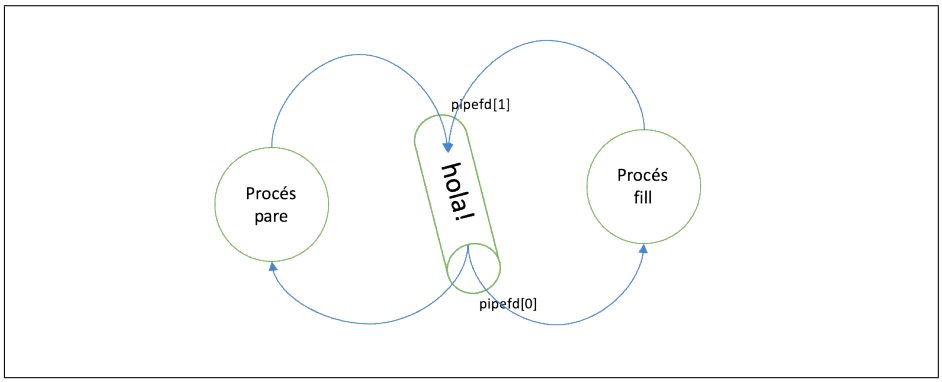

Figura 2: Codi Cèsar. Just després del `fork`.

En aquest cas tenim dos processos comunicats per una pipe. El procés pare escriu a la pipe el missatge pla que el procés fill codificarà tot llegint de la pipe i escriure el text codificat per sortida estàndar.

Llistat 2: Codificació Cèsar

```
1  int main() {
2  int pipefd[2],pid;
3      char c;
```

```
4       char buff[]= "Codificacio Cesar de 3.\n Text pla: hola!\n
            Text xifrat: ";
5       write(1,buff,strlen(buff));
6       if (pipe(pipefd)<0) {
7           perror("Error en crear la pipe.");
8           exit(1);
9       }
10      if (write(pipefd[1],"hola!",5)<0) {
11          perror("Error en escriure a la pipe");
12          exit(2);
13      }
14      pid=fork();
15      if (pid<0) {
16          perror("Error de fork().");
17          exit(1);
18      }
19      if (pid==0){
20          close(pipefd[1]); /* important!!!!*/
21          while(read(pipefd[0],&c,1)>0) {
22              c+=3;
23              if (write(1,&c,1)<0) {
24                  perror("Error en escriure a la stdout");
25                  exit(2);
26              }
27          }
28          close(pipefd[0]);
29          write(1,"\n",1);
30          exit(0);
31  }
32      close(pipefd[1]); /* important!!!!*/
33      close(pipefd[0]);
34      wait(NULL);
35      exit(0);
36  }
```

A la línia 14 hi ha la crida `fork()` que crea (si tot va bé) el procés fill. Just després d'aquesta línia tindrem dos processos comunicats per una pipe. Tant el pare com el fill tenen a la seva taula de canals punters als canals de lectura i escriptura de la pipe, tal com es mostra a la figura 2.

Aquest dibuix no és el que volem i de fet és una situació perillosa. Noteu que el procés fill anirà llegint de la pipe mentre quedin dades (línia 21).

Això vol dir que sortirà del bucle quan la crida `read` retorni 0. I quan retorna 0? Doncs no pas quan acabi de llegir "hola!" perquè podria ser que el procés pare escrivís més dades.

L'única manera que el `read` retorni 0 és que no hi hagi cap escriptor a la pipe, és a dir, quan es tanquin tots els canals d'escriptura (tant del pare com del fill). Mentre hi hagi la possibilitat que algú escrigui a la pipe, `read` bloquejarà al procés, esperant que escriguin.

Per tant, el primer que han de fer (tant el pare com el fill) és tancar les entrades de la taula de canals que no necessiten (i que poden bloquejar processos). Així a la línia 32, el pare tanca el seu canal d'escriptura de la pipe. El fill tanca el seu canal d'escriptura de la pipe a la línia 20. Noteu que pare i fill s'executen concurrentment des del `fork()`, per tant no podem saber quines línies (les del pare o les del fill) s'executaran primer.

Després que el fill executi la línia 20, tenim el dibuix de la figura 3, ja sense perill de bloquejos. El pare escriu per la pipe a la línia 10 (fins i tot abans de crear al fill), però no tanca la pipe (ni lectura, ni escriptura) perquè volem que

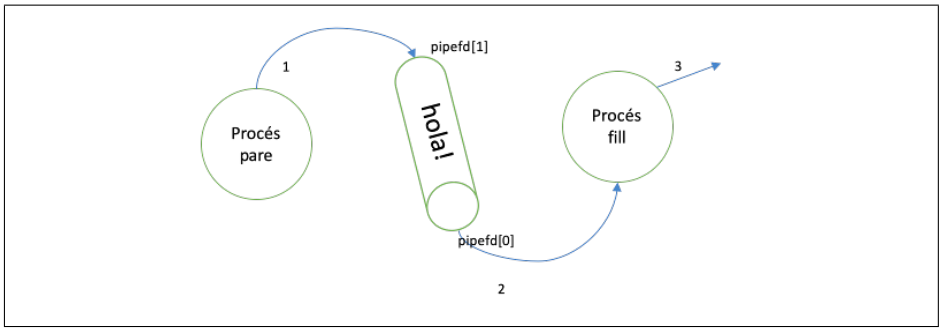

Figura 3: Codi Cèsar. El pare ha tancat la lectura (lin. 32) i el fill l'escriptura de la pipe (lin. 20).

el fill hereti la pipe.

Un cop creat el fill el pare ja no necessita la pipe i el primer que fa es tancar-la (línies 32 i 33). Després espera que el fill acabi (línia 34) i acaba ell també (línia 35). Recordeu que el paràmetre de la funció `exit` s'envia al seu pare i aquest el pot recollir amb el paràmetre del `wait`, tot i que en aquest cas el procés pare no ho fa (a la línia 34 li passa un `NULL`).

3.1 Pipes amb nom

Les pipes amb nom es diuen també FIFOs permeten comunicar processos sense parentiu. Les FIFOs funcionen igual que les pipes[3], les *syscalls* `read()` i `write()` funcionen igual que amb les pipes. En aquest cas, la fifo es pot crear des de la línia de comandes i després obrir-la, com si d'un fitxer qualsevol es tractés, des de dins el codi. O bé, es pot crear dins del codi amb la *syscall* `mknod()`[4], tant com a comanda de *shell* com a *syscall*. Noteu que `mknod` es fa servir per més coses, no només per crear pipes amb nom (antigament fins i tot es feia servir per crear directoris). En canvi, `mkfifo` només serveix per crear pipes amb nom.

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

El primer argument és el nom amb que la fifo apareixerà al sistema de fitxers. Sí, les pipes amb nom es veuen en fer un "ls". El segon argument són els permisos del fitxer (recordeu que, per defecte, el resultat final serà modificat per la `umask` del sistema). El tercer argument ha de ser `S_IFIFO` per crear una pipe amb nom. Feu `man 2 mknod`.

De manera anàloga, la fifo es pot crear des de shell. Noteu el primer caràcter de la línia en fer un llistat del directpori:

```
$ mknod mevaPipe p
$ ls -la
prw-r--r--  1 jfornes users     0 Nov 12 08:55 mevaPipe|
```

Un cop creada, la pipe pot ser oberta des de codi (o des de shell), per lectura o per escriptura, com un fitxer qualsevol, amb la *syscall* `open()`. I podem

[3]Les pipe de vegades es diuen "pipes sense nom", *unnamed pipes*, a manca d'un nom millor i per oposició a les fifos.

[4]POSIX proposà `mkfifo` i des de llavors `mknod` i `mkfifo` conviuen en els sistemes tipus UNIX (en algunes implementacions, `mkfifo` crida a `mknod`).

fer servir les *syscalls* `read()`, `write()`, `close()` sobre el descriptor de fitxer que retorna `open` de la manera habitual. Tot i això, cal tenir en compte que treballem sobre una pipe i per tant:

- Si un procés intenta llegir d'una pipe buida, el `read` es bloqueja fins que hi hagi dades. Corolari: si un procés intenta obrir una fifo sense escriptors, l'`open` es bloqueja fins que un altre procés obri la fifo per escriptura.
- Si un procés intenta escriure en una pipe plena[5], el `write` es bloqueja fins que un altre procés buida part del buffer de la pipe.
- Si tots els canals d'escriptura[6] sobre una pipe han estat tancats, `read` retorna 0. En les cas de les FIFOs, si tots el descriptors de fitxer per escriptura sobre una fifo han estat tancats, la crida `read` retorna final-de-fitxer, és a dir, 0.
- No es pot fer servir `lseek()` sobre una fifo (ni sobre una pipe).
- Si tots els canals de lectura sobre una pipe han estat tancats, el procés que cridi a `write` sobre la pipe (o fifo) rebrà un `SIGPIPE`. Si el procés està ignorant aquest signal, `write` falla, actualitzant la `errno` a `EPIPE`.
- Escriptures sobre la pipe de menys de `PIPE_BUF` (en Linux, 4096) bytes són àtomiques. Si escrivim més, el kernel podria intercalar bytes d'altres processos.

3.2 Un exemple de pipes amb nom

A la figura 3.2 teniu el llistat de dos processos que fan, si fa o no fa, el mateix que el llistat 2. El codi de la dreta rep com a paràmetre una cadena de caràcters que ha de codificar amb Cèsar-3. Crea una pipe anomenada `canal_segur`, l'obre per escriptura, escriu la cadena, tanca la pipe i acaba.

El codi de l'esquerra, simètricament, crea la mateixa pipe, l'obre per lectura, i va llegint, codificant i escrivint per sortida estàndard els caràcters xifrats. Quan no hi ha més bytes a la pipe la tanca i acaba.

El primer que ens ha de sobtar és que tots dos processos intenten crear la pipe. Cal tenir en compte que no sabem a priori quin dels dos s'executarà primer, per això tots dos intenten crear la pipe a la línia 3. No tindria sentit intentar obrir una pipe que no existeix[7]. Però si tots dos criden a crear la *named pipe* en un dels dos casos, el segon intent, la crida `mknod()` retornarà un nombre menor de 0 i actualitzarà l'`errno`. Si la raó de l'error és que ja existeix la pipe (`errno == EEXIST`), no el considerem un error de debò (de fet, no ho és) i tiren endavant.

La segona qüestió que ens hem de plantejar és que passa si executem només un dels dos programes. Si executem primer el llistat 4, podem escriure a una pipe que no té lectors? I si executem primer el llistat 3, podem llegir d'una pipe que no té escriptors encara? La resposta és no. De fet, si només executem un

[5] La capacitat de la pipe depèn de la implementació del sistema operatiu. En Linux és de 65536 bytes.

[6] En general, es diu que una pipe a la que li han tancat els canals lectors o els escriptors és vídua. Un `read` sobre una vídua retorna 0, un `write` sobre una vídua reb un signal SIGPIPE.

[7] Altrament, podriem crear la *named pipe* prèviament des de shell (`mknod canal_segur p`).

Llistat 3: Xifra el text de la pipe

```
#include "apuntsPipes.h"
int main (int argc, char *argv[]) {
int fd; char c;
if (mknod("canal_segur", S_IFIFO |
     0666, 0) < 0 && errno != EEXIST)
    error_i_exit("mknod  named pipe");
if ((fd=open("canal_segur", O_RDONLY))
     < 0)
    error_i_exit("open");
while(read(fd,&c,1)>0) {
    c+=3;
    if (write(1,&c,1)<0)
        error_i_exit("write a 1");
}
write(1,"\n",1);
close(fd);
}
```

Llistat 4: Envia text pla per la pipe

```
#include "apuntsPipes.h"
int main (int argc, char *argv[]){
int fd;
if (mknod("canal_segur", S_IFIFO |
     0666, 0) < 0 && errno != EEXIST)
    error_i_exit("mknod named pipe");
if ((fd=open("canal_segur", O_WRONLY))
     < 0)
    error_i_exit("open named pipe");
if (write(fd,argv[1],strlen(argv[1]))
     <0)
    error_i_exit("write a la named
        pipe");
close(fd);
}
```

Figura 4: Dos processos sense parentiu es comuniquen per una pipe anomenada `canal_segur` creada des del codi.

dels dos programes, el procés es bloquejarà en fer la crida `open()` de la *named pipe*, independentment de si la intenta obrir per lectura o escriptura, fins que tingui un procés recíproc a l'altra banda de la pipe.

Per últim, però no menys important, fixeu-vos en el segon paràmetre del `mknod()`, és de tipus `mode_t`, un reguitzell de bits que codifiquen tant el tipus de fitxer[8] com els seus permisos. Hauria de ser una combinació, una *OR* bit a bit, dels tipus de fitxers `S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO, S_IFSOCK` i els permisos habituals (feu `man 2 stat`), però també és modificat per la `umask` del sistema. Els permisos de la pipe creada seran doncs $mode \wedge \neg umask$.

4 Implementació dins el kernel

El kernel considera la pipe com un dispositiu i li assigna un nou inode. Això li permetrà assignar-li blocs de dades.

4.1 Obrint pipes

El kernel crea una entrada a la taula de fitxers oberts (*File Table*) per lectura i una altra per escriptura i les inicialitza apuntant al nou inode. Crea dues entrades a la taula de canals (*file descriptor table*) del procés que ha fet la crida; una per lectura i una altra per escriptura, inicialitzant-les apuntant a les respectives entrades de la taula de fitxers oberts. El nombre de referències de l'inode s'inicialitza, doncs, a 2. Inicialitza el comptador de lectors i d'escriptors de l'inode a 1.

L'algorisme per a la creació de pipes sense nom el teniu a la figura 5.

Aquesta seqüencia es pot veure a la figura 6. Per arribar fins a aquesta situació, el procés pare ha començat la seva execució amb les entrades 0, 1 i 2 de la seva TC plenes apuntant al terminal (l'entrada 0 de la TFO). L'entrada 0 de la TFO apunta a la entrada 0 de la taula d'inodes, és a dir, de l'inode del

[8] Seria més correcte parlar de *node*, però a UNIX tots són fitxers, així que ja ens entenem.

```
algorisme pipe
entrada: res
sortida: entrades a la TC de lectura i escriptura
{
    assignar nou inode al dispositiu pipe (algorisme ialloc)
    assignar  dues noves entrades a la Taula de Fitxers Oberts
        una per lectura i una altra per escriptura
    inicialitzar les noves entrades de la TFO
       apuntant cap al nou inode
    assignar  dues noves entrades a la Taula de Canals.
       una per lectura i una per escriptura
    inicialitzar les noves entrades de la TC
       apuntant a les respectives entraddes de la TFO
    inicialitzar el comptador  de referències a l'inode a 2 (TI)
    inicialitzar el comptador de lectors, escriptors de l'inode a 1 (TFO)
}
```

Figura 5: Algorisme per a la creació de pipes sense nom. Font [Back (1986)]

terminal en direm X. La primera columna de la TFO indica quantes entrades de TCs (*file descriptors*) estan fent-la servir. L'entrada 0 de la TFO, en crear-se el procés pare valdrà 3 perquè l'apunten només les entrades 0, 1 i 2 de la TC del pare.

Quan el pare crea la pipe, s'omplen dues entrades més de la seva TC, la 3 apunta al cantó de la pipe de lectura i la 4 al cantó de la pipe d'escriptura. Noteu que, seguint l'algorisme de la figura 5, es creen també dues entrades a la TFO (una amb permisos de lectura i una amb permisos d'escriptura), però que totes dues apunten al mateix inode, que en direm Y. Després el pare fa la crida a `fork()` i crea el seu fill.

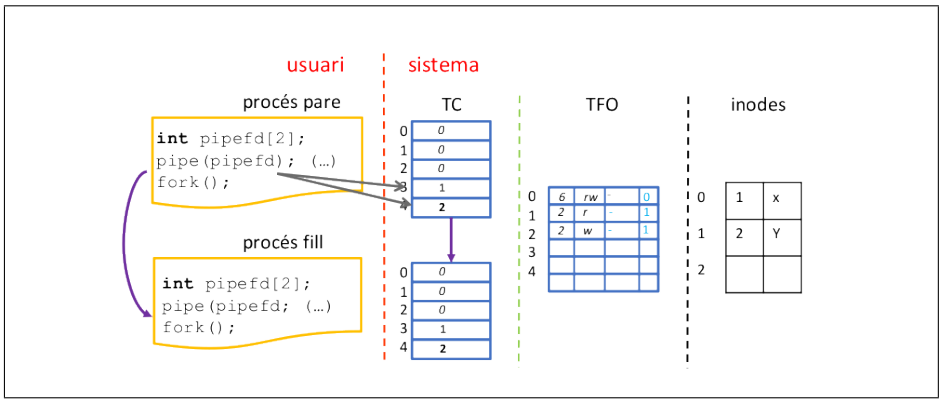

Figura 6: Estructures de dades del kernel relacionades amb la pipe.

Quan el pare crea el fill, aquest inicialitza els valors de la seva TC amb els valors de la TC del seu pare. Això fa que tant pare com a fill apuntin a les mateixes entrades de la TFO. Per la qual cosa, la primera columna de la TFO dobla el seu valor (recordeu que indica quantes entrades de TCs estan fent-la servir). Abans només comptava els punters del pare; ara compta els del pare i els del fill. Dit d'una altra manera, la primera columna de l'entrada 1 de la TFO mostra quantes instàncies de la pipe estan obertes per lectura. Inicialment

valia 1, però després del `fork()` val 2. Noteu que en el cas de les pipes el camp del desplaçament de lectura/escriptura (tercera columna, mostra un '-') no té sentit. No es pot modificar amb `lseek()` i de fet, en molts sistemes, l'`offset` de la pipe es guarda a l'inode de memòria (*in-core inode*).

El nombre de referències a l'inode (primera columna de la taula d'inodes) indica quants cops la pipe ha estat "oberta", o dit d'una altra manera, quantes entrades de la TFO apunten a aquest inode. Des que es va crear la pipe aquest camp val 2. Noteu que aquest taula d'inodes està a memòria i guarda informació només dels inodes carregats a memòria.

L'única diferència entre les pipes (sense nom) i les FIFOs (pipes amb nom) és que aquestes últimes apareixen al directori de fitxers i s'ha d'accedir a elles mitjançant un camí (*path name*), és a dir, amb la crida `open()`. L'algorisme de `open()` d'una és el mateix que per a qualsevol altre fitxer, però abans d'acabar la *syscall* el kernel incrementarà, a la taula d'inodes, el nombre de cops que la pipe ha estat oberta.

El kernel passo a dormir (estat de BLOQ) a aquells processos que intentin obrir una FIFO en lectura sense escriptors (i viceversa) i s'encarregarà de despertar (passar a estat de READY) als processos lectors quan un procés intenti obrir una FIFO per escriptura (i viceversa).

4.2 Llegint i escrivint pipes

Des del punt de vista del kernel, els processos escriuen al final de la pipe i llegeixen del principi. El nombre de processos escriptors no té perquè coincidir amb el nombre de lectors i queda pel programador com aquets processos es coordinen per fer servir la pipe.

En general, el kernel accedeix a la pipe igual que ho fa a qualsevol altre fitxer. Les dades es guarden en blocs que s'assginen en fer cirdes a `write()`, tot i que la pipe només fa servir els blocs de dades directes de l'inode. L'altre diferència és que el kernel mai reescriu dades de la pipe (no es fa servir `lseek()`).

Si un procés intenta llegir més dades de les que hi ha a la pipe, el `read()` retornarà immediatament amb tots les dades de la pipe, tot i que no satisfarà la petició del procés. Si la pipe està buida, el kernel bloquejarà al procés fins que algun escriptor afegeixi dades a la pipe. En aquell moment el kernel despertarà a tots els processos que esperaven llegir de la pipe.

Si un procés intenta escriure a una pipe més dades de la que aquesta pot acceptar en aquell, el kernel bloqueja al procés fins que algun lector dreni la pipe. Quan algun procés faci una crida `read()` de la pipe, el kernel despertarà a tots els processos que volien escriure. Però, compte, si un procés intenta escriure a una pipe més de la capacitat d'aquesta, el kernel escriurà tot el que pugui i bloquejarà el procés fins que hi hagi espai per a la resta. Això vol dir que les dades no quedessin contigües, si un altre procés escriu entremig.

4.3 Tancant pipes

La crida `close()`, com sempre, tanca l'entrada de la TC del procés, decrementa l'entrada de la TFO. Si el nombre d'escriptors d'aquesta arriba a zero, l'allibera, però previament envia un zero i desperta a tots els processos que volien llegir d'aquella pipe, fent que retornin del `read()`. Si el nombre de lectors arriba a zero, allibera l'entrada després de despertar a tots els processos que volien

escriure enviant-los un SIGPIPE. Finalment el kernel decrementa l'entrada de la TI, si arriba a zero, allibera l'inode i tots els blocs que tenia assignats.

Aquest comportament és identic per *named* i unnamed pipes. Noteu que en el cas de les FIFOs podria ser que hi haguessin més processos lectors (o escriptors) en el futur, però el kernel desperta als processos bloquejats igualment, per ser consistent amb la implementació de les pipes sense nom.

5 Resumint

La pipe sense nom (*syscall* `pipe(fd)`) només pot establir-se entre processos amb parentiu. La pipe s'ha de crear abans de cridar a `fork()` perquè la comparteixin pare i fill. Provoca dues noves entrades a la taula de canals (*file descriptor table*), dues entrades a la taula de fitxers oberts (*file table*) i una nova entrada a la taula d'inodes (*inode table*), tot i que no apareix al sistema de fitxers.

La pipe amb nom sí que apareix al sistema de fitxers. Es pot fer servir per comunicar processos amb parentiu o sense. Es crea amb `mknod` o `mkfifo` i després cada procés que hi vol accedir l'obre amb `open` i per tant crea una nova entrada a la seva taula de canals (*file descriptor table*), una nova entrada a la seva taula de fitxers oberts (*file table*) i un increment del nombre d'enllaços de l'inode, si aquest ja estava carregat a memòria, o una nova entrada a la taula d'inodes (*inode table*), si no ho estava. Si un procés fa un `open` per lectura d'una *named pipe* que no té cap procés que l'hagi obert per escriptura el procés es bloquejarà. I a l'inrevés; si un procés fa un `open` per escriptura d'una *named pipe* que no té cap procés que l'hagi obert per lectura el procés es bloquejarà. Però, esclar, l'`open` no provocarà el bloqueig si aquest és per lectura i escriptura.

L'única diferència entre pipes i FIFOs es la manera de crear-les i obrir-les. A partir d'aquí, l'E/S sobre totes dues te la mateixa semàntica. El `read` i el `write` funcionen igual en tots dos casos. Un `read` d'una pipe sense escriptors retorna 0. Un `write` sobre una pipe sense lectors provoca un `SIGPIPE`.

6 Exemples

1. Ens demanen programar, com si fossim el shell, un `ls|wc`, és a dir, que la sortida del procés que `ls`ta els contiguns del directori sigui l'entrada del procés que compta paraules (i línies i lletres), `wc`. Per a fer això, el programa principal farà de shell, crearà la pipe i dos fills, un per a cada comanda. Noteu que el procés pare, el shell, no fa res, només espera a que acabin els seus fills, tal com faria un shell de debó. La solució, al llistat 5.

2. Versió estesa del codificador Cèsar. Fem més general l'exemple de la codificació Cèsar (llistat2). Ara volem que tot el que arribi per entrada estàndar s'escrigui per sortida estàndard codificat. La solució, al llistat 6.

Referències

[Byant & O'Hallaron (2016)] Bryant, Randal E., O'Hallaron David R. 2016. *Computer systems : a programmer's perspective*. Pearson, 2016.

https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711

[Kernighan (2020)] Kernighan, Brian. 2020. *UNIX. A History and a Memoir*. Kindle Direct Publishing, 2020.

[Back (1986)] Back, Maurice J. 1986. *The Design of the UNIX Operating System*. Prentice Hall, PTR. 1986.

https://discovery.upc.edu/permalink/34CSUC_UPC/rdgucl/alma991000067889706711

[Sechrest (1986)] Sechrest, Stuart. *An Introductory 4.3BSD Interprocess Communication Tutorial*, Unix Programmer's Supplementary Documents, Vol. 1 (PS1), 4.3 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, Univ. of California, Berkeley, Calif., Apr 1986.

https://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf

[Kernighan (1984)] Kernighan, B. W., & Pike, R. *The UNIX Programming Environment*. Prentice-Hall, Inc., Englewood Cliffs, NY. 1984

https://discovery.upc.edu/permalink/34CSUC_UPC/rdgucl/alma991000110939706711

[Stevens (2013)] Stevens, W. Richard. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004011309706711

[McKusick (2005)] McKusick, Marshall Kirk. *The design and implementation of the FreeBSD operating system*. Addison-Wesley, 2005. https://discovery.upc.edu/permalink/34CSUC_UPC/8e3cvp/alma991003102659706711

Llistat 5: ls|wc

```
#include <sys/wait.h>
#include <stdlib.h>
int main() {
int     fd[2];
if (pipe(fd) < 0) {
        perror("error pipe");
        exit(1);
}
switch (fork()) {
case -1:
        perror("error fork");
        exit(1);
case 0:
        close(fd[0]);
        close(1);
        dup(fd[1]);
        close(fd[1]);
        execlp("ls", "ls", (char *)0);
        perror("execlp ls");
        exit(1);
default:
        switch (fork()) {
        case -1:
                perror("error fork");
                exit(1);
        case 0:
                close(fd[1]);
                close(0);
                dup(fd[0]);
                close(fd[0]);
                execlp("wc", "wc", (char *)0);
                perror("execlp wc");
                exit(1);
        default:
                close(fd[0]);
                close(fd[1]);
                while (waitpid(-1, NULL, 0) > 0);
        }
}
return 0;
}
```

Llistat 6: Cèsar estès

```c
#include <sys/wait.h>
#define BUFFSZ 256
int main(int argc, char** argv) {
int pipefd[2],pid,n,r,fd;
    char c;
    char buff[BUFFSZ];
    if (argc<2) {
        write(1,"Manquen arguments!\n",19);
    }
    n =atoi(argv[1]);
    if (pipe(pipefd)<0) {
        perror("Error en crear la pipe.");
        exit(1);
    }
     pid=fork();
    if (pid<0) {
        perror("Error del primer fork().");
        exit(1);
    }
    if (pid==0){
        close(pipefd[1]); /* important!!!!*/
        while(read(pipefd[0],&c,1)>0) {
            c+=n;
            if (write(1,&c,1)<0) {
                perror("Error en escriure a la stdout");
                exit(2);
            }
        }
        close(pipefd[0]);
        exit(0);
    }
   pid=fork();
   if (pid<0) {
       perror("Error del segon fork().");
       exit(1);
   }
   if (pid==0){
       close(pipefd[0]);
       while ((r=read(0,buff,BUFFSZ))>0) {
           if (write(pipefd[1],buff,r)<0) {
               perror("Error en escriure a la pipe");
               exit(2);
           }
       }
       close(pipefd[1]); /* important!!!!*/
       exit(0);
}
    close(pipefd[0]);
    close(pipefd[1]); /* important!!!!*/
    while (waitpid(-1, NULL, 0) > 0);
    exit(0);
}
```