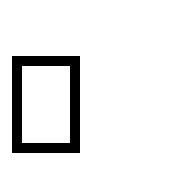
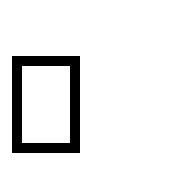
# INTERAZIONE TRA PROCESSI UNIX

L’interazione tra processi può avvenire:

mediante la condivisione di file dato che questa è una eccezione che UNIX presenta rispetto al modello ad ambiente locale:

* complessità nella realizzazione della sincronizzazione tra i processi (si veda esempio nella slide 24 sui processi)

Se però NON consideriamo la precedente eccezione, dato che i processi UNIX non possono condividere memoria (visto che UNIX si basa sul **modello ad ambiente locale**), l’unica possibilità di interazione è la cooperazione

attraverso specifici strumenti di *Inter Process Communication*:

##  per la comunicazione tra processi

***sulla stessa macchina:***

* **pipe** (tra processi della stessa gerarchia)
* **fifo** (qualunque insieme di processi)

***su nodi diversi della stessa rete:***

## socket

 **per la sincronizzazione tra processi**

***sulla stessa macchina:***

## segnali

Gli strumenti di *Inter Process Communication* per la comunicazione tra processi  hanno come scopo quello di risolvere, in generale, il problema noto in letteratura come **problema produttore-consumatore** …

# PROBLEMA PRODUTTORE/I- CONSUMATORE/I

In tantissimi campi applicativi c’è la necessità di affrontare il **problema produttore-consumatore** o più, in generale, il **problema produttori-consumatori**

**A livello teorico**, il problema si enuncia nel seguente modo: un processo PRODUTTORE (o più processi PRODUTTORI) produce (producono) informazioni (di qualunque tipo) che sono consumate da un processo CONSUMATORE (o più processi CONSUMATORI)

Per consentire l’esecuzione concorrente dei processi PRODUTTORI e CONSUMATORI è necessario che i due ‘tipi’ di processi possano utilizzare uno **strumento di bufferizzazione** in modo che i PRODUTTORI producono le informazioni e le depositano usando lo strumento di bufferizzazione e i CONSUMATORI prelevano usando lo strumento di bufferizzazione e quindi usano le informazioni, nel senso che le consumano

**A livello pratico:**

un esempio nella *vita reale*, diversi produttori di cellulari producono, appunto, cellulari e li depositano, ad esempio, in un unico magazzino e i consumatori li possono prelevare (pagandoli, *of course …*) e li usano!

***In ambito informatico*** gli esempi sono innumerevoli; qui, se ne riportano solo 2:

1. Un processo che deve stampare su una stampante produce un insieme di caratteri che deposita in un buffer di spooling da cui vengono prelevati dal processo che gestisce la stampante che quindi li stampa;
2. Piping dei comandi in UNIX ad esempio ls | grep “stringa”

 il processo produttore che esegue ls produce i dati che deposita sullo standard output che però è agganciato allo standard input del processo che esegue il grep e che quindi li estrae e li usa (quindi li consuma)!

# COMUNICAZIONE TRAMITE PIPE

# (solitamente nei compiti d’esame ci sono)

Mediante una pipe, la comunicazione tra processi è **indiretta**

 ***mailbox***

Una pipe è un canale di comunicazione tra processi:



* ***unidirezionale***:

i dati possono fluire in un solo verso, Se serve il verso contrario si crea una nuova pipe!

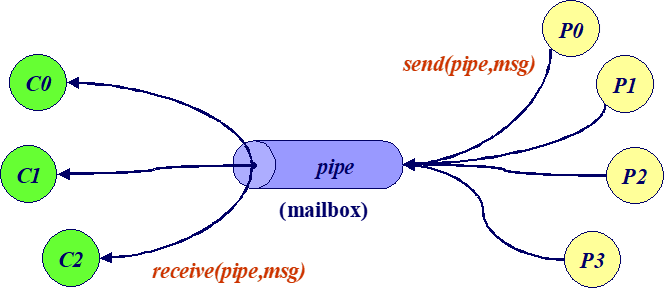


* **capacità limitata** (rappresenta lo strumento di bufferizzazione): la **pipe** è in grado di gestire l’**accodamento** di un numero limitato di messaggi, gestiti in modo FIFO: il limite è stabilito dalla **dimensione** della pipe
* **molti-a-molti:**
  + più processi possono spedire messaggi attraverso la stessa pipe

 produttori

* + più processi possono ricevere messaggi attraverso la stessa pipe

 consumatori



**NOTA BENE:** a seconda della specifica del problema:

* i processi Pi (**produttori**) possono essere denominati anche **client**, o genericamente mittenti (o sender)
* i processi Cj (**consumatori**) possono essere denominati anche

**server**, o genericamente destinatari (o receiver)

# CREAZIONE DI UNA PIPE

**PIPE**: retval = **pipe** (piped); int piped[2];🡪 array di due interi che rappresenterà gli elementi ritornati dall’esecuzione della pipe. Deriva dal fatto che in Unix tutto a parte i processi sono visti come file, e quindi anche la pipe. Si accede alla pipe tramite la TFA del processo e si memorizzano nei primi due elementi liberi. L’array piped conterrà i due valori che indicizzano i FD.

int retval;

retval vale 0 in caso di *successo*, altrimenti un valore negativo

In caso di ***successo***

* vengono allocati **DUE** nuovi elementi nella Tabella dei File Aperti del processo e i rispettivi file descriptor vengono memorizzati in piped[0] e piped[1]

Quindi, dopo la creazione, comunicare tramite una PIPE avviene con le stesse primitive che si devono usare per operare su un FILE

Infatti, piped[0] e piped[1] reppresentano ***RISPETTIVAMENTE***, il lato di lettura e quello di scrittura sulla pipe

 si usano le stesse primitive di lettura e scrittura dei file (READ e WRITE) per operare sui lati della PIPE, così come si può usare la primitiva CLOSE per chiudere i lati della PIPE

**Vd esempio provaPipe.c e lungpipe.c**

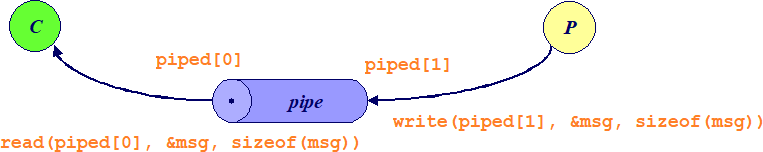


**piped**

|  |
| --- |
|   **piped[0] lato di lettura della pipe** |
| **piped[1] lato di scrittura della pipe**   |

# OMOGENEITÀ CON I FILE

Quindi, ogni lato di accesso alla pipe è visto dal processo in modo **omogeneo** ad un file (file descriptor):





Riceve un messaggio

🡪 lo legge

Manda un messaggio

🡪 lo scrive



File descriptor: identifica di solito un oggetto che permane nel sistema , nel caso della PIPE fa riferimento ad una zona di memoria volatile del kernel per cui non permane

**DIFFERENZE**:



* 1. ai **file descriptor** di una pipe non corrispondono alcun nome nel file system
     + la PIPE è una struttura che ***non permane***

alla TERMINAZIONE dei processi

* 1. la **dimensione** di una PIPE è fissa
     + ad una PIPE è associato un buffer (p.e. di 4 kbytes)
  2. la PIPE usa una **gestione FIFO** 
     + i primi dati scritti in una PIPE sono i primi a essere letti

non si puo cambiare l’elemento da leggere come nei file tramite la primitiva lseek ma i primi che mandiamo sono i primi ricevuti

Nella **pipe** è insito un meccanismo di sincronizzazione tipo

**produttore/consumatore**, per cui:

* un processo (*consumatore*) che legge (*estrae*) da pipe (piped[0]) **si blocca** se la pipe è vuota, e rimane in attesa che arrivino dei dati (transita da running a sleeping in attesa del verificarsi dell’evento: produttore ha messo qualcosa in pipe, ricevila!). invece nei file la read torna 0 ma non si blocca!
* un processo (*produttore*) che scrive (*inserisce*) su una pipe (piped[1]) **si blocca** se la pipe è piena, in attesa che si liberi dello spazio

# ESEMPIO CHE CONSENTE DI DETERMINARE LA LUNGHEZZA DI UNA PIPE: fatto

#include <stdio.h>

…

int count;

main()

{ int piped[2]; char c = ’x’;

if (pipe(piped) < 0) { printf(”Errore\n”); exit(1); } for (count = 0;;) /\* ***ciclo infinito*** \*/

{

write(piped[1], &c, 1); /\* scrittura sulla pipe \*/

if ((++count % 1024) == 0)

printf(”%d caratteri nella pipe\n”, count);

}

}

## OSSERVAZIONE: Dobbiamo abortire l’esecuzione di questo codice!

**Esempio di lunghezza di una pipe su:**

LINUX (in particolare macchina virtuale lx04)

 **65536** (64 KByte)

N.B. Lo verificheremo con esattezza solo in seguito!

# QUALI PROCESSI POSSONO COMUNICARE MEDIANTE PIPE?

Per mittenti (sender, client, produttori, ...) (chi puo scrivere sulla pipe) e destinatari (receiver, server, consumatori, ... che ricevono dalla pipe ) l'accesso al canale di comunicazione avviene tramite un file descriptor che rappresenta rispettivamente il lato di scrittura o di lettura di una pipe

* Quindi, soltanto i processi appartenenti a una stessa gerarchia (cioè, che hanno un *antenato* in comune) possono scambiarsi messaggi mediante pipe

IMPORTANTE:

Le possibilità di comunicazione tramite pipe sono ad esempio:

* tra un processo padre e processi figli;
* tra processi fratelli (che ereditano la pipe dal processo padre)
* tra *nonno* e *nipoti*
* etc.

si crea prima la pipe e poi si creano i figli in modo che nella TFA ottenuta come copia del padre ci siano i riferimenti al lato di lettura e scrittura della pipe

Caso di un processo padre e di un processo figlio che vogliono comunicare (il verso è ininfluente):

main()

{int pid, piped[2]; char msg[]=“ciao”;

**pipe(piped); /\* CREAZIONE PIPE: due elementi in più nella**

**TFA del processo PADRE \*/**

**if ((pid=fork())==0) /\* CREAZIONE FIGLIO \*/**

**{/\* figlio: eredita per COPIA la TFA del PADRE oltre alla COPIA dell'array pd \*/**

**write(piped[1], msg, 4); /\* figlio scrive sulla pipe \*/**

**...**

**exit(0);}**

**/\* padre: legge dalla pipe\*/ read(piped[0], msg, 4);**

**...}**

# ESEMPIO: comunicazione fra padre e figlio

# PROTOCOLLO DI COMUNICAZIONE

# È quel qualcosa che i processi che devono comunicare tramite la pipe e viene stabilito a priori:

# chi comunica con chi 🡪 chi è il processo produttore e chi il processo consumatore

# Padre Figlio

# Consumatore Produttore

CHI è COSA:

Si consideri un primo esempio semplice di comunicazione: il processo padre (che si comporterà come un *consumatore*/receiver nei confronti di una pipe) e un processo figlio (che si comporterà come un *produttore*/sender nei confronti della pipe)

COSA DEVONO COMUNICARE: si devono scambiare un certo numero (NON NOTO A PRIORI) di stringhe C (cioè null-terminated) di lunghezza 4 caratteri (5 con il terminatore di stringa);

in particolare, il figlio legge le stringhe da un file (il cui nome è passato come parametro), le invia al padre, che le deve riportare su standard output

Vediamo quali sono i passi che si devono pianificare perché la comunicazione possa avere luogo:

1. **Primo passo fondamentale** è che il padre, PRIMA DI CREARE UN FIGLIO, deve creare una pipe (con la primitiva **pipe**); quindi nella TFA del padre ci saranno due descrittori che serviranno, uno, come lato di lettura della pipe, e l’altro, come lato di scrittura della pipe  questi due descrittori nell’esempio sono salvati nell’array *piped*!
2. Il padre quindi deve creare un FIGLIO (con la primitiva **fork()**) che si troverà nella sua area kernel una copia della TFA del padre
3. La specifica del nostro esempio stabilisce il verso (***unidirezionale***) che deve avere la pipe  dal figlio al padre, cioè il figlio deve scrivere e il padre deve leggere; per stabilire questo verso il figlio DEVE chiudere il lato di lettura, che non usa (con la primitiva **close()**) e il padre DEVE chiudere il lato di scrittura, che non usa)  si veda slide seguente!

**ATTENZIONE: rispettare la parte di specifica che stabilisce il PROTOCOLLO DI COMUNICAZIONE**

soELab@Lica04:~/pipe$ cat pipe-new.c

*/\* FILE: pipe-new.c \*/*

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <unistd.h>*

*#include <fcntl.h>*

*#include <sys/wait.h>*

*#define MSGSIZE 5 🡪 SONO LINEE DI 4 CARATTERI + TERMINATORE (LA SIZE DEL MESSAGGIO è QUINDI 5)*

*int main (int argc, char \*\*argv)*

*{ int pid, j, piped[2]; /\* pid per fork, j per indice, piped per pipe \*/*

*char mess[MSGSIZE]; /\* array usato dal figlio per inviare stringa al padre \*/*

*char inpbuf [MSGSIZE]; /\* array usato dal padre per ricevere stringa inviata dal figlio:* ***N.B: si poteva usare sempre mess, tanto il padre e il figlio agiscono sulla loro copia delle variabili!*** *\*/*

*int pidFiglio, status, ritorno; /\* per wait padre \*/*

*if (argc != 2)*

*{*

*printf("Numero dei parametri errato %d:ci vuole un singoloparametro\n",argc);*

*exit(1);*

*}*

*/\* si crea una pipe \*/*

*if (pipe (piped) < 0 )*

*{ printf("Errore creazione pipe\n");*

*exit (2);*

*}*

*if ((pid = fork()) < 0)*

*{ printf("Errore creazione figlio\n");*

*exit (3);*

*}*

*if (pid == 0)*

*{*

*/\* figlio \*/*

*int fd;*

*close (piped [0]); 🡪 il figlio CHIUDE il lato di lettura: perché PRODUCE*

*if ((fd = open(argv[1], O\_RDONLY)) < 0)*

*{ printf("Errore in apertura file %s\n", argv[1]);*

*exit(-1); /\* torniamo al padre un -1 che sara' interpretato come* ***255*** *e quindi identificato come errore \*/*

*}*

*printf("Figlio %d sta per iniziare a scrivere una serie di messaggi,ognuno di lunghezza %d, sulla pipe dopo averli letti dal file passato come parametro\n", getpid(), MSGSIZE);*

*j=0; /\* il figlio inizializza la sua variabile j per contare i messaggi che ha mandato al padre \*/*

*while (read(fd, mess, MSGSIZE)) /\* il contenuto del file e' tale che in mess ci saranno 4 caratteri e il terminatore di linea \*/*

*{*

*/\*il padre ha concordato con il figlio che gli mandera' solo stringhe e quindi dobbiamo sostituire il terminatore di linea con il terminatore di stringa\*/*

*mess[MSGSIZE-1]='\0';*

*write (piped[1], mess, MSGSIZE);*

*j++; (il figlio cosi stampa quanti messaggi ha scritto nella pipe)*

*}*

*printf("Figlio %d scritto %d messaggi sulla pipe\n", getpid(), j);*

*exit(0);*

*}*

*/\* padre \*/*

*close (piped [1]); /\* il padre CHIUDE il lato di scrittura perché CONSUMA \*/*

*printf("Padre %d sta per iniziare a leggere i messaggi dalla pipe\n", getpid());*

*j=0; /\*il padre inizializza la sua variabile j per verificare quanti messaggi*

*ha mandato il figlio \*/*

*while (read ( piped[0], inpbuf, MSGSIZE))🡨* **dato che il processo scrittore figlio ad un certo punto termina, la primitiva read tornerà 0 e quindi il processo lettore terminerà il while!**

*{*

*/\* dato che il figlio gli ha inviato delle stringhe, il padre le puo' scrivere direttamente*

*con una printf \*/*

*printf ("%d: %s\n", j, inpbuf);*

*j++;*

*}*

*printf("Padre %d letto %d messaggi dalla pipe\n", getpid(), j);*

*/\* padre aspetta il figlio \*/*

*pidFiglio = wait(&status);*

*if (pidFiglio < 0)*

*{*

*printf("Errore wait\n");*

*exit(5);*

*}*

*if ((status & 0xFF) != 0){*

*printf("Figlio con pid %d terminato in modo anomalo\n", pidFiglio);*

*}*

*else{*

*ritorno=(int)((status >> 8) & 0xFF);*

*printf("Il figlio con pid=%d ha ritornato %d (se 255 problemi!)\n", pidFiglio, ritorno);*

*}*

*exit (0);*

*}*

Guardare la presentazione di questo esempio di codice .

# OSSERVAZIONE:

DUE PROCESSI CHE CONDIVIDONO UNA PIPE POTREBBERO ***ENTRAMBI*** SCRIVERE O LEGGERE SULLA/DALLA PIPE

|  |  |  |
| --- | --- | --- |
| **processo**  | piped[0] | * **processo** |
| **padre**  | piped[1] |  **figlio** |

# IMPORTANTE X DETERMINARE IL VERSO:

OGNI PROCESSO CHIUDE UN LATO DELLA PIPE E USA SOLO L'ALTRO

La pipe è creata dal processo padre

i file descriptor sono i primi liberi, il 3 e 4

**PADRE FIGLIO**

**(B)**



**Processo PADRE**

**Processo FIGLIO**

**(A)**

**Processo**

**FIGLIO**

**Processo**

**PADRE**

**(B)**

0

0

0

0

1

**piped**

1

1

1

2

2

2

**piped**

3

4

3

4

3

4

2

3

4

**4**

**3**

**4**

**3**

|  |
| --- |
| stdin |
| stdout |
| stderror |
| **piped[0]** |
| **piped[1]** |
|  |
|  |

|  |
| --- |
| stdin |
| stdout |
| stderror |
| **piped[0]** |
| **piped[1]** |
|  |
|  |

|  |
| --- |
| stdin |
| stdout |
| stderror |
| **piped[0]** |
| **piped[1]** |
|  |
|  |

|  |
| --- |
| stdin |
| stdout |
| stderror |
| **piped[0]** |
| **piped[1]** |
|  |
|  |

Il processo figlio eredita per copia la TFA del padre e quindi i file descriptor del padre (A)

Il padre chiude la parte di scrittura della pipe (effetto SOLO su TFA del padre) ed il figlio chiude la parte di lettura (effetto SOLO su TFA del figlio) (B)

* si chiudono tutti i file descriptor non necessari

**(segue OSSERVAZIONE)**

Vediamo di illustrare che informazioni mantiene il S.O. (in particolare il KERNEL di UNIX) relativamente ad una pipe:

1. Chiaramente, il kernel mantiene una zona di memoria (della lunghezza vista precedentemente) che viene gestita in modalità FIFO, cioè i primi dati che vengono scritti sono anche i primi che vengono letti (e quindi estratti) dalla pipe
2. Inoltre il kernel mantiene una tabella che riporta i PID dei processi che possono operare su una pipe, sia come lettori che come scrittori; questa tabella, chiaramente, si crea contestualmente alla pipe e si modifica in seguito ad una fork, a delle close e alla morte dei processi!

Facciamo un esempio di questa tabella per la pipe creata dal padre dell’esempio:

* 1. Dopo la creazione della pipe, la situazione è questa

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | Yes |

* 1. Dopo la creazione del figlio, la situazione è questa

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | Yes |
| Pid-figlio | Yes | Yes |

* 1. Dopo le chiusure operate sia dal padre che dal figlio, la situazione è questa

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | No |
| Pid-figlio | No | Yes |

Quando il figlio termina, la tabella associata alla pipe diventa:

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | No |

quindi la pipe non risulterà più avere scrittori e quindi il ciclo di lettura del padre avrà termine e quindi potrà poi terminare anche il padre!

# PROBLEMA: PIPE SENZA SCRITTORE

Vediamo che cosa succede ad un processo CONSUMATORE/RECEIVER se **MUORE** il PRODUTTORE/SENDER

(cioè secondo la terminologia della pipe lo SCRITTORE) prima dell’invio di tutti i messaggi che avrebbero dovuto mandare

**NOTA BENE:** come caso limite, facciamo terminare il processo figlio (che nel nostro esempio è lo scrittore!) senza mandare alcun messaggio, ma il discorso vale anche se lo scrittore morisse dopo aver mandato qualche messaggio!

#include <stdio.h>

…

int main (int argc, char \*\*argv)

{ …

if (pid == 0)

{

**/\* figlio \*/**

int fd;

close (piped [0]); /\* figlio CHIUDE il lato lettura \*/

…

printf("Figlio %d sta per iniziare a scrivere …);

**/\* IL FIGLIO TERMINA ===> PIPE SENZA SCRITTORE \*/**

exit (0);

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | No |
| Pid-figlio | No | Yes |

}

/\* padre \*/

close (piped [1]); /\* padre CHIUDE lato scrittura \*/

…

while (read ( piped[0], inpbuf, MSGSIZE))

{ … }

if (j != 0) …

else { puts("NON C'E' SCRITTORE"); exit(4); }

… }

* la **read** ritorna **0** se non ci sono dati e non ci sono processi scrittori, per non BLOCCARE INDEFINITAMENTE il processo lettore (nel nostro caso il processo padre!)

 NOTA BENE: è quello che succede anche in condizioni normali!

# PROBLEMA: PIPE SENZA LETTORE

Vediamo il problema complementare e cioè che cosa succede ad un processo PRODUTTORE/SENDER se **MUORE** il CONSUMATORE/RECEIVER (cioè secondo la terminologia della pipe il LETTORE) prima della ricezione di tutti i messaggi che avrebbero dovuto essere mandati

**NOTA BENE:** come caso limite, facciamo terminare il processo padre (che nel nostro esempio è il lettore!) senza leggere alcun messaggio, ma vale anche se il lettore morisse dopo aver letto qualche messaggio!

#include <stdio.h>

…

int main (int argc, char \*\*argv)

{ …

if (pid == 0)

{ /\* figlio \*/ int fd;

close (piped [0]); /\* figlio CHIUDE il lato lettura \*/

…

printf("Figlio %d sta per iniziare a scrivere …); while (read(fd, mess, MSGSIZE))

{ mess[MSGSIZE-1]='\0';

write (piped[1], mess, MSGSIZE); j++; } printf("Figlio scritto %d messaggi sulla pipe\n",j); exit (0);

}

/\* padre \*/

close (piped [1]); /\* padre CHIUDE lato scrittura \*/ printf("Padre %d sta per iniziare a leggere …);

**/\* il padre termina ===> PIPE SENZA PIU'LETTORE \*/**

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | No |
| Pid-figlio | No | Yes |

exit (0);

}

* il sistema spedisce il segnale **SIGPIPE** al processo scrittore per avvisarlo che non ci sono più processi lettori: questo segnale, *di default*, provoca la terminazione del processo figlio

 **N.B.** questo evita che la pipe si saturi e che quindi il processo scrittore si blocchi indefinitamente

Esempi interattivi🡪 senza lettore e scrittore

# PIPING DI COMANDI

Si consideri ora come il sistema UNIX potrebbe gestire il **piping** di due comandi da SHELL

Si noti l’uso della primitiva **dup** ===>

un nuovo file descriptor viene associato allo stesso file cui fa riferimento il file descriptor fornito come parametro

# PRIMITIVA DUP

Per duplicare un elemento della tabella dei file aperti di processo:

**DUP:** retval = **dup**(fd);

int fd, è il file descriptor da duplicare int retval;

L’effetto di una **dup** è copiare l’elemento di indice fd della Tabella dei File Aperti del processo nella prima posizione libera (quella con l’indice minimo tra quelle disponibili)

Restituisce il nuovo file descriptor (del file aperto copiato) cioè il nuovo indice, oppure -1.

La primitiva dup può essere usata in generale per duplicare qualunque file descriptor: in particolare, viene usata per duplicare i file descriptor di una pipe e realizzare il piping di comandi

***Funzionamento programma shellpipe (che deriva da shellpipe.c)***

***Nel disegno che segue tralasciamo il processo della shell di partenza: quindi il processo P1 è il***

***processo, creato dalla shell, che esegue il programma shellpipe (e che nelle nostre intenzioni***

***simula la shell); il processo P2èil processo figlio di P1 che (come nella Bourne Shell) deve***

***occuparsi della gestione dei comandi in piping; quindi P2, crea la pipe e poi crea il processo P3; P3***

***si occupa, dopo aver fatto le operazioni necessarie al collegamento del suo standard output con il***

***lato di scrittura della pipe, di eseguire com1, mentre P2, si occupa, dopo aver fatto le operazioni***

***necessarie al collegamento del suo standard input con il lato di lettura della pipe, di eseguire com2***Immagine che contiene testo

Descrizione generata automaticamente

**ESEMPIO**: Piping di comandi (si riporta un codice semplificato)

#include <stdio.h>

…

int **join** (char \*com1[], char \*com2[])

{ int status; int pid;

int piped[2];

/\* **processo P1: simulazione del processo shell** \*/

/\* **creazione** del figlio per eseguire il comando in pipe \*/ switch ( fork ()) {

case -1: /\* errore \*/ return (1);

case 0: /\* figlio ===> **processo P2** \*/ break; default: /\* padre P1: attende il figlio \*/

wait (&status); return (status); }

**/\* il figlio P2 tratta il comando intero: crea la pipe \*/**

if (pipe (piped) < 0 ) { exit(-1); }

/\* CREAZIONE di un nuovo FIGLIO: **processo P3** \*/ if ((pid = fork()) < 0) { exit(-1); }

else

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **P1** | | | | | | |
|  | | | | | | |
| ***stdin***  | **P3** |  |  |  | **P2** | * ***stdout*** |
|  |  | ***pipe*** | | |  |  |

if (pid == 0) **/\* processo P3: figlio del figlio P2 \*/**

{ close(1); /\* lo std output va messo sulla pipe \*/

**dup(piped[1]);**

close(piped [0]); close(piped [1]); /\* NON servono più \*/

**execvp(com1[0], com1);**

exit(-1); /\* errore in caso si ritorni qui \*/

}

else **/\* processo P2 (padre di P3) \*/**

{ close (0); /\* std input va preso dalla pipe \*/

**dup(piped[0]);**

close(piped [0]); close(piped [1]); /\* NON servono più \*/

**execvp(com2[0], com2);**

exit(-1); /\* errore in caso si ritorni qui \*/

}}

main (int argc,char \*\* argv)

{ int integi, j, i;

char \*temp1 [10], \*temp2 [10];

/\* si devono fornire nella linea comandi due comandi distinti, separati dal carattere **!** (punto esclamativo). Non si usa direttamente il **|**, perche’ questo viene direttamente interpretato dallo shell come una pipe \*/

if (argc > 3) /\* ci devono essere almeno 3 stringhe \*/

{ for (i=1; i < argc && **strcmp (argv[i], “!”)**; i++) temp1[i-1] = argv[i];

temp1[i-1] = (char \*)0; i++;

for ( j = 1; i < argc ; i++ , j++) temp2[j-1] = argv[i];

temp2[j-1] = (char \*)0; /\* terminatore \*/

}

else { printf(“Errore nel numero parametri“); exit(3); } integi = join(temp1, temp2);

exit(integi);

}

## PRIMA DI DUP E CLOSE DOPO DUP E CLOSE



**Processo P2**

**Processo P3**

**Processo P3 Processo P2**

0 0

**Processo P2 Processo P3**

stdin stdout

1 1



0 stdin 0

 1 **piped[1]** 1

stdin stdout

stdout

**piped[0]**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | stderror |  | 2 | stderror |  |  | 2 | stderror |  | 2 | stderror |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | **piped[0]** |  | 3 | **piped[0]** |  |  | 3 |  |  | 3 |  |
| 4 | **piped[1]** |  | 4 | **piped[1]** |  |  | 4 |  |  | 4 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |









# ESEMPIO CHE CONSENTE DI DETERMINARE LA LUNGHEZZA

**DI UNA PIPE (uso di alarm**  **segnali):**

#include <stdio.h>

…

#include <signal.h>

int count;

**void alrm\_action()**

{ printf (”write bloccata dopo %d caratteri\n”, count);

exit(0);

}

main()

{ int p[2]; char c = ’x’;

if (pipe(p) < 0) { printf(”Errore\n”); exit(1); }

**signal(SIGALRM, alrm\_action);**

for (count = 0;;)

{ **alarm(20)**; /\* settiamo l’allarme \*/ write(p[1], &c, 1); /\* scrittura sulla pipe \*/ **alarm(0);** /\* resettiamo l’allarme \*/

if ((++count % 1024) == 0)

printf (”%d caratteri nella pipe\n”, count);

}

}

## OSSERVAZIONE: NON dobbiamo abortire l’esecuzione di questo codice!

**Esempio di lunghezza di una pipe su:**

LINUX (in particolare macchina virtuale lx04)

 **65536** (64 KByte)

**PROBLEMA: PIPE SENZA LETTORE (bis)**

Vediamo lo stesso problema di prima, ma dove lo scrittore (nel nostro caso, il figlio), cattura il segnale SIGPIPE e quindi termina in modo normale.

#include <stdio.h>

…

int flag = FALSE;

void Azione(int sig)

{ printf("Arrivato segnale # %d\n", sig); flag = TRUE;

}

int main (int argc, char \*\*argv)

{ …

if (pid == 0)

{ /\* figlio \*/ int fd;

**signal(SIGPIPE, Azione); /\* si aggancia Azione per trattare il segnale SIGPIPE \*/**

close (piped [0]); /\* figlio CHIUDE il lato lettura \*/

…

printf("Figlio %d sta per iniziare a scrivere …); while (read(fd, mess, MSGSIZE))

{ mess[MSGSIZE-1]='\0';

write (piped[1], mess, MSGSIZE); j++; } printf("Figlio scritto %d messaggi sulla pipe\n",j); exit (0);

}

/\* padre \*/

close (piped [1]); /\* padre CHIUDE lato scrittura \*/ printf("Padre %d sta per iniziare a leggere …);

**/\* il padre termina ===> PIPE SENZA PIU'LETTORE \*/**

exit (0);

|  |  |  |
| --- | --- | --- |
| ***PID*** | ***Lettore*** | ***Scrittore*** |
| Pid-padre | Yes | No |
| Pid-figlio | No | Yes |

}

* il sistema spedisce il segnale **SIGPIPE** al processo scrittore per avvisarlo che non ci sono più processi lettori: alla ricezione del segnale non si esegue l’azione di default (la terminazione anormale) ma il segnale viene catturato e gestito in modo opportuno dalla funzione Azione() che scrive su std output una frase di commento e poi fa terminare il processo in modo normale (con una exit)!

# PIPE CON NOME ===> FIFO

Le pipe hanno due **SVANTAGGI**:

1. consentono la COMUNICAZIONE ***SOLO*** fra processi in relazione di parentela;
2. **non** sono PERSISTENTI

Non risulta possibile tramite le pipe, progettare un processo **GESTORE** di una risorsa che deve poter ricevere messaggi di richiesta da un qualunque processo nel sistema

**SOLUZIONE**: PIPE CON NOME dette anche FIFO

Una FIFO si comporta come una PIPE cioè:

rappresenta un canale UNIDIREZIONALE di tipo first-in first- out (si usano SEMPRE le operazioni **read** e **write**)

ma, con queste ***differenze***:

1. possiede un nome UNIX  permane nel sistema

ha un proprietario, un insieme di diritti ed una lunghezza

1. deve essere creata con una primitiva diversa dalla *pipe()*

retval = **mknod**(path, mode); int retval, mode;

char \*path;

## NOTA:

Serve anche per creare i direttori e i file speciali (dispositivi)

* dopo si deve usare la open

# ESEMPIO:

un programma RECFIFO.c rappresenta il SERVITORE un altro programma SENDFIFO.c rappresenta i clienti

**/\* file RECFIFO.c \*/** #include <fcntl.h> #include <stdio.h> #include <string.h> #define MSGSIZ 60

main()

{ int fd; char msgbuf[MSGSIZ+1];

/\* Apertura FIFO \*/

if ((**fd = open (”fifo”, O\_RDWR)**) < 0)

{ printf(”Errore in open\n”); exit(1); }

/\* Ricezione messaggi \*/

for (;;) /\* processo ciclico \*/

{ if (read(fd, msgbuf, MSGSIZ+1) < 0)

{ printf(”Errore in lettura\n”); exit(2); }

printf(”Messaggio ricevuto: %s\n”, msgbuf);

}

}

## NOTA:

La FIFO viene aperta in lettura e scrittura nel SERVITORE. Infatti se fosse stata aperta solo in lettura, quando i processi clienti terminano, la *read* ritornerebbe 0 poichè non esisterebbe il processo scrittore e si avrebbe un inutile loop nel SERVITORE. In questo caso, invece, la read risulta bloccante poichè lo stesso servitore viene riconosciuto come scrittore.

**ESEMPIO** (segue)

**/\* file SENDFIFO.c \*/** #include <fcntl.h> #include <stdio.h> #include <string.h> #define MSGSIZ 60

main(int argc, char \*\*argv)

{ int fd, i, nwrite; char msgbuf[MSGSIZ+1];

if (argc < 2) { printf (”Errore nel numero parametri\n”); exit(1); }

/\* Apertura FIFO \*/

if ((**fd = open (”fifo”, O\_WRONLY | O\_NDELAY)**) < 0)

{ printf(”Errore in open\n”); exit(2); }

/\* Spedizione messaggi \*/ for (i = 1 ; i < argc; i++)

{ if (strlen(argv[i]) > MSGSIZ)

{ printf(”Errore messaggio troppo lungo\n”); continue; } strcpy(msgbuf, argv[i]);

if ((nwrite = write(fd, msgbuf, MSGSIZ+1)) <= 0)

{ printf(”Errore in scrittura\n”); exit(3); }

}

exit(0);

}

## NOTA:

La FIFO viene aperta in scrittura usando il flag O\_NDELAY per non bloccare il processo se non esiste un processo che abbia aperto la stessa FIFO in lettura (cioè il corrispondente SERVITORE)  torna -1

## Esempi di uso:

$ mknod fifo p

$ RECFIFO& PID = ...

$ SENDFIFO "messaggio numero 1" "messaggio numero 2" Messaggio ricevuto: messaggio numero 1

Messaggio ricevuto: messaggio numero 2

$ SENDFIFO "messaggio numero 3" Messaggio ricevuto: messaggio numero 3

**ALTERNATIVA** PER LA CREAZIONE DELLA FIFO

* DA PROGRAMMA

/\* file CREAFIFO.c \*/ main()

{

if (mknod(”fifo”, 010600) < 0) printf(”Errore\n”);

}