Lezione 13: IPC FIFO

Fin qui siamo stati in grado di scambiare i dati tra processi legati tra loro: tali processi sono avviati da un antenato comune. Questa è una limitazione poichè talvolta è necessario che processi tra loro non in relaizone siano in grado di scambiarsi i dati. Questo è possibile farlo con l'uso delle pipe con nome o FIFO.

13.1 Pipe con nome: mkfifo()

Possiamo crearèuna FIFO dalla linea di comando o da proframma, dalla linea di comando: `mkfifo filename` da programma:

```
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
```

la funzione mkfifo crea una nuova fifo e resituisce un errore EEXIT se la fifo già esiste, se non é desiderata la creazione di una fifo é necessario invocare `open()` anzichè mkfifo().

Per aprire una fifo esistente o crearèuna nuova FIFO se questa non esiste:

si invoca `mkfifo` si controlla un eventuale errore `EEXIST` e se questo si verifica si apre open.

13.1.1

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(){
    int res = mkfifo("/tmp/my_fifo", 0777);
    if (res == 0)
        printf("FIFO creata\n");
    exit(0);
}
```

Il programma usa la funzione `mkfifo` per crare in file speciale, sebbene impostiamo i permessi 0777 questi sono filtrati dalla maschera dell'utente umask, così come avviene nella normale creazione di file. I permessi risultanti saranno 755 se ad esempio`umask è 022.

Possiamo rimuovere la FIFO come un file convenzionale o da linea di comando o da codice usando `unlink()`

13.2 Accedere ad una FIFO

Una carattesitica utile delle pipe con nome è che risiedendo nel file system per cui le possiamo usare in comandi dove normalmente utilizziamo nomi di file. Proviamo a leggere dalla fifo vuota.

Ora scriviamo nella FIFO usando un altro terminale, infatti il primo comando si sospende in attesa di dati scriti nellàfifo.

SI vedrà che l'out appare dal comando cat. Se non si invia alcun dato alla fifo, il comando cat si sospenderà finop a che non lo si interrompe.

Possiamo fare entrambe le cose ponendo il primo comando in backgroung

13.3 Aprire una FIFO con open()

La principale restrizione quando si apre una FIFO è che un programma non può aprirla per leggere e scrivere nella modalità `O_RDWR` in tal caso il risultato è indefinito.

Questa è una restrizione relativa,poichè usiamo le FIFO per passare i dati in una singola direzione e dunque non c'è necessità di aprira in quella modalità.

Quando scriviamo scriviamo in coda e a leggere si legge dall'inizio

Un'altra differenza nell'aprire una FIFO rispetto ad un file regolare è l'uso dell'argomento oflag, uno di questi valori é `O_NONBLOCK`

l'utilizzo di questa modalità di apertura non solo cambia il modo in cui la chiamata open viene elaborata ma cambia anche il modo in cui sono elaborate le richieste.

```
open(const char *path, O_RDONLY);
```

in questo caso, la chiamata ad open si bloccherà; non ritorna fino a che un processo apre la
stessa FIFO per scrittura

```
open(const char *path, O_RDONLY | O_NONBLOCK);
```

la chiamata ad open ha successo e ritorna immediatamente, anche se la FIFO non è stata
aperta in scrittura da alcun processo

```
open(const char *path, O_WRONLY);
```

la chiamata ad open si bloccherà fino a che un processo apre la stessa FIFO in lettura

```
open(const char *path, O_WRONLY | O_NONBLOCK);
```

ritorna sempre immediatamente, ma se nessun processo ha la FIFO aperta in lettura, open
ritornerà un errore, -1, e la FIFO non sarà aperta. Se un processo ha aperto la FIFO in lettura,
il descrittore di file restituito può essere usato per scrivere al suo interno

13.3.1 Esempio

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[]){
    int res;
    int open_mode = 0;
    int i;
    if (argc < 2) {
        printf("Uso: %s <combinazioni di O_RDONLY O_WRONLY O_NOI
        exit(1);
    }

    /* Impostiamo il valore di open_mode dagli argomenti. */
```

```
    for(i = 1; i < argc; i++) {
        if (strncmp(*++argv, "O_RDONLY", 8)==0) open_mode |= O_R
        if (strncmp(*argv, "O_WRONLY", 8)==0) open_mode |= O_WR
        if (strncmp(*argv, "O_NONBLOCK", 10)==0)open_mode |= O_N
    }

    /* Se la FIFO non esiste la creiamo. Poi viene aperta */
    if (access(FIFO_NAME, F_OK) == -1) {
        res = mkfifo(FIFO_NAME, 0777);
    if (res != 0) {
        printf("Non posso creare la FIFO %s\n", FIFO_NAME);
        exit(1);
    }

    printf("Processo %d apre la FIFO\n", getpid());
    res = open(FIFO_NAME, open_mode);
    printf("Risultato processo %d: %d\n", getpid(), res);
    sleep(5);
    if (res != -1) close(res);
        printf("Processo %d terminato\n", getpid());
        exit(0);
}
```

13.4 Leggere e scrivere in una FIFO bloccante

Se la fifo è aperat in mod bloccante, una read() su una fifo vuota, aspetterà fino a che è disponibile qualche dato da leggere, sela FIFO è aperta in scrittura, restituisce 0, se la fifo non è aperta in scrittura.

Una write() su una FIFO aspetterà fino a che i dati possono essere scritti, se la FIFO é aperta in lettura. Genera SIGPIPE se la FIFO non è aperta in lettura.

13.5 LEggere dalle FIFO non bloccanti

L’utilizzo della mod O_NONBLOCK influisce sul comportamento delle chiamate read() sulle FIFO, una read() su una FIFO vuota non bloccante, restituisce un errore sel la fifo è aperta in scrittura. REstituisce 0 se la fifo non è aperta in scrittura.

Sulla write invecr genera una SIGPIPE se la FIFO non è apertàin lettura. Se la fifo è aperta in lettura, se il numero di byte da scrivere è ≤ di PIPE_BUF. Se c’è spazio per il numeròdi byte specificato sono trasferiti tutti i byte. Se non c’è spazio per tutti i byte ritorna EAGAIN

Se è superiore allora se c’è almeno 1 byte nella FIFO il kernel trasferisce tanti byte quanto spazio c’è e restituisce quello che ha scritto altrimenti torna EAGAIN

13.6 Esempio: comunicazione tra due processi con FIFO

```
#include <unistd.h>
…
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1000 * 10)
int main(){
int pipe_fd;
int res;
int open_mode = O_WRONLY;
int bytes_sent = 0;
char buffer[BUFFER_SIZE];
if (access(FIFO_NAME, F_OK) == -1) {
res = mkfifo(FIFO_NAME, 0777);
if (res != 0) {
```

```
printf("Could not create fifo %s\n", FIFO_NAME);
exit(-1);}
}
printf("Process %d opening FIFO O_WRONLY\n", getpid());
pipe_fd = open(FIFO_NAME, open_mode);
printf("Process %d result %d\n", getpid(), pipe_fd);
if (pipe_fd != -1) {
while(bytes_sent < TEN_MEG) {
// supponiamo che buffer sia riempito altrove
res = write(pipe_fd, buffer, BUFFER_SIZE);
if (res == -1) {
printf("Write error on pipe\n");
exit(1);
}
bytes_sent += res;
}
close(pipe_fd);
}
else
{
exit(1);
}
printf("Process %d finished\n", getpid());
exit(1);
}
```

13.7 Client Server

Un utilizzo delle FIFO consiste nell'inviare i dati tra un client ed un server. Se abbiamo un server che è contattato da numerosi client, ogni client può scrivere la sua richiesta ad una FIFO che il server crea. Poiché possono esserci multipli scrittori per la FIFO le richieste inviate dai client al server devono essere minori di `PIPE_BUF` byte di dimensione.

Vogliamo un singolo processo server che accetta richieste, le elabora e restituisce i dati risultanti alla parte richiedente ovvero il client.

Vogliamo consentire a processi client multipli di inviare dati al server. Per semplicità assumiamo che i dati da elaborare siano spezzati in blocchi, ciascuno più piccolo di i `PIPE_BUF` byte.

Poichè il serve elaborerà solo un blocco di informazione per volta, consideriamo un'unica FIFO che é letta dal srver e scritta da ciascun client. Aprendo la FIFO in modo bloccante, il server ed il client saranno automaticamente sincronizzati come richiesto.

Restituire i dati ai client è leggermente più difficile. Abbiamo bisogno di organizzare unàseconda pipe, una peècliente, peèi dati ritornati passando l'identificatore di processo del client nei dati originiali invaiti al server entrambe le parti possono usarlo per generare il nomi unico per la fifo di ritorno.

Il server crea la sua FIFO in modalità solo lettura e si blocca, ciò avviene fino a che il primo client si connette aprendo la stessàFIFO in scrittura. A quel punto il processo del server si sblocca ed è eseguita una sleep in modo che le scritture dei client si accodano (la sleep non è usata nelle app reali) nel frattempo dopo che il client ha aperto la fifo del server, esso crea la propria FIFO identificativa univocamente peèleggere i dati provenienti dal server. Solo allora il client scrive i dati al server e dopo si blocca su una letturàdella propria FIFO in attesa della risposta.

Ricevendo i dati dal client, il server li elabora, apre la FIFO del client per la scrittura e scrive i dati. Ciò sblocca il client, quando il client é sbloccato esso può leggere dalla sua FIFO i dati scritti in essa dal server.

L'intero processo si ripete fino a che l'ltimo client chiude la pipe del server, provocando un fallimento della lettura del server poichè nessun processo ha la

pipe del server aperta in scrittura. Se questa fosse un processo server reale che necessita di aspettare altri client, dovremmo modificalo per:

- aprire la propria FIFO in scrittura, in modo che la lettura si blocca sempre piuttosto che ritornare 0 oppure chiudere e riaprire la FIFO del server quando `read()` restituisce 0 byte così il processo server si blocca con la oper in attesa di un client.

Esercizi per casa